

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P.  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p><b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b></p>					
1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)			
13-10-2011	Final Report	October 2010 - October 2011			
4. TITLE AND SUBTITLE			<p>META Adaptive, Reflective, Robust Workflow (ARRoW) Phase 1b Final Report</p>		
			5a. CONTRACT NUMBER	HR0011-10-C-0108	
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S)			5d. PROJECT NUMBER		
Bankes, Steven; Challou, Daniel; Cooper, David; Haynes, Todd; Holloway, Hillary; Pukite, Paul; Tierno, Jorge; Wentland, Christopher			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)			8. PERFORMING ORGANIZATION REPORT NUMBER		
BAE Systems U.S. Combat Systems 4800 East River Road Minneapolis, MN 55421-1498			TR-2742		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)		
Defense Advanced Research Project Agency 3701 North Fairfax Drive Arlington, VA 22203-1714			DARPA		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT					
Distribution Statement B: "Distribution authorized to U.S. Government agencies only due to the inclusion of proprietary information and to prevent Premature Dissemination of potentially critical technological information. Other requests for this document shall be referred to DARPA Technical Office via email at tio@darpa.mil."					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
The goal of the META program is to reduce development cycle time for complex cyber-physical systems such as aircraft, rotorcraft, and ground vehicles by a factor of 5x over current cycle times. Our approach employs model-based methodologies for revolutionizing design and verification processes currently used in industry. The objective is to develop a new set of metrics and flows that use this model-based approach and then define and develop the new infrastructure needed and make this technology available to industry.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:		17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE	SAR	David F. Adams	
U	U	U		19b. TELEPHONE NUMBER (Include area code) 763-572-6181	

Standard Form 298 (Rev. 8/98)  
Prescribed by ANSI Std. Z39-18

## Table of Contents

1.	Summary .....	1
1.1	Task Objectives.....	2
1.2	Technical Challenges.....	2
1.3	General Methodology .....	3
1.4	Technical Results .....	3
1.5	Important Findings and Conclusions .....	5
1.6	Significant Hardware Development.....	6
1.7	Special Comments.....	6
1.8	Implications for Further Research .....	6
2.	Introduction.....	7
3.	Methods, Assumptions, and Procedures.....	8
3.1	Achieving Program Goals .....	8
3.2	Simulation's Role in IFV Analysis.....	9
3.2.1	Illustrative Example.....	12
3.2.2	Enabling Infrastructure.....	13
4.	Results and Discussion .....	14
4.1	ARRoW Tool Chain/Workflow Overview.....	17
4.2	Stages in an ARRoW Facilitated Design Process .....	18
4.2.1	Requirements Ingestion.....	18
4.2.2	Initial Decomposition.....	19
4.2.3	Initial Test Case Generation .....	20
4.2.4	Initial Requirements Reasoning .....	20
4.2.5	System Conceptualization .....	21
4.2.6	System Composition .....	22
4.2.7	Mixed Initiative Design Exploration .....	23
4.2.8	System Detail Engineering Design .....	25
4.2.9	System Operational Assessment.....	26
4.3	ARRoW Foundation Infrastructure .....	27
4.3.1	AMIL – ARRoW Model Interconnection Language .....	27
4.3.2	Component Model Library .....	28
4.3.3	Verification Methods.....	29
4.3.4	Metrics.....	29
4.4	Notional Demo System Application.....	31
5.	Conclusions.....	32

6. Recommendations .....	32
6.1 Integration of Additional Tools .....	32
6.2 Application to Other Domains .....	35
6.3 Achieving Industry Reform and a 5x Compression in System Development Time .....	35
6.4 The Hybrid Approach to Democratizing Design .....	36
7. Appendices .....	37
7.1 System Engineering and Architecture .....	37
7.2 Tool Design .....	37
7.3 Modeling Language .....	37
7.4 Library Requirements .....	37
7.5 System Demonstration .....	37
7.6 Advanced Reasoning and Applications of ARRoW Technology .....	37
7.7 Metrics Developed by Team Member (BBN) .....	37
7.8 Spatial Design Exploration (BBN) .....	37
7.9 RMPL (MIT) .....	37
7.10 Verification (MIT) .....	37
7.11 Programmatic .....	37

## List of Figures

Figure 1. Elements of the ARRoW Tool Chain .....	5
Figure 2. Tool&Model Abstraction Levels, Integration, and Verification Relationships .....	10
Figure 3. Structural Dynamics Supports Mobility Analysis .....	13
Figure 4. ARRoW Tool Chain – From Requirements to Manufacture .....	17
Figure 5. ECTo .....	22
Figure 6. Tool Flexibility in Early Design Phases .....	25
Figure 7. “Heavyweight” Analysis Tools with Potential “Lightweight” Alternatives .....	26
Figure 8. Example Dashboard Configuration .....	31
Figure 9. Multiphysics Levels of Difficulty/Maturity and Relevance to IFV Development .....	33
Figure 10. Overlap Among Multiphysics Modeling and Analysis Topics .....	35

## List of Tables

Table 1. The Components of ARRoW .....	14
--	----

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
AIDE	ARRoW Integrated Development Environment
AMIL	ARRoW Model Interconnection Language
ARRoW	Adaptive Reflexive Robust Workflow
CAD	Computer-Aided Design
CML	Component Model Library
CONOPS	Concept of Operations
DSE	Design Space Exploration
ECTo	Early Concepting Tool
ESKER	Expert-System Knowledgebase Evaluation Reasoner
IFV	Infantry Fighting Vehicle
ITAR	International Traffic in Arms Regulations
MBE	Model Based Engineering
MM	Master Model
NRMM	NATO Reference Mobility Model
PCC	Probabilistic Certificate of Correctness
PoC	Probability of Correctness
QML	Qualitative Modeling Language
RAS	Requirement Archetype Sets

## 1. Summary

ARRoW (Adaptive Reflexive Robust Workflow) is a software infrastructure, which facilitates the design of complex cyber-physical systems by supporting computational exploration of alternative designs with continuous test and verification using:

- Multiple models at various level of abstraction
- Integrated heterogeneous specialized reasoners
- Libraries of components, design patterns, and workflows

Developed for DARPA's META program intended to accelerate the development process of Combat Vehicle Systems by a factor of 5, ARRoW includes models, reasoners, and libraries for the design of Infantry Fighting Vehicles (IFVs). However, the infrastructure is readily extensible to other design problems by providing content appropriate for the specific cyber-physical system being addressed. The ARRoW infrastructure facilitates more aggressive use of computation, reducing the workload on engineers while allowing for mixed initiative exploration for good solutions. Where existing model based engineering approaches are often constrained by the use of isolated computational models, ARRoW allows these isolated models to be joined, greatly enhancing their value.

ARRoW supports distributed collaboration, allowing not only the use of geographically distributed information and computational resources, but also collaboration among engineers, extensible to a crowd-based development paradigm. This allows for the verification of early design concepts to accelerate the design process while providing for early detection of problems where they can be addressed at reduced cost. This same capability could provide for improved interactions between customers and performers, illuminating design tradeoffs, allowing customers to more clearly understand their options, and avoiding many of the problems currently associated with communication through requirements. In providing mechanisms to automate routine tasks currently performed by engineers, ARRoW facilitates the utilization of engineering resources where they are most valuable, promotes faster completion of design tasks, and has the potential to promote the democratization of design.

The following sections summarize our approach and results. The remainder of this summary covers task objectives, technical challenges, general methodology, technical results, important findings and conclusions, and implications for future research. An introduction to the main body of the report is followed by sections on:

- Methods, Assumptions, and Procedures
- Results and Discussion
- Conclusions, and
- Recommendations.

Accompanying this report is a series of appendices that provide greater technical detail:

1. ARRoW System Engineering and Architecture
2. Tool Design
3. Modeling Language
4. Library Requirements
5. System Demonstration
6. Advanced Reasoning and Applications of ARRoW Technology

7. Metrics Developed by Team Member BBN
8. Spatial Design Exploration (BBN)
9. RMPL (MIT)
10. Verification (MIT)
11. Programmatic

## 1.1 Task Objectives

The Phase 1b task objectives were:

1. Develop a detailed design for ARRoW's integrated system design, verification and validation toolset.
2. Develop syntax for the modeling language and requirements for the structure and content of the model library, which is accessed by the ARRoW toolset.
3. Implement the ARRoW software toolset.
4. Demonstrate ARRoW's suitability for synthesizing a notional vehicle and providing traceability from the development process to the program objectives regarding schedule.

These objectives all serve the overall program goal of achieving a 5-fold reduction in the time to design complex cyber-physical systems.

## 1.2 Technical Challenges

Designing and producing today's complex aerospace and defense vehicle systems requires engineering a labyrinth of complex systems, each with numerous states, many subsystems, and thousands of unique components, resulting in a multitude of subsystem interactions, myriad lines of software code, and large numbers of requirements and metrics at tension with one another. This problem is challenging, not only due to the number of components that make up an infantry fighting vehicle, but in the number of different aspects to the performance of a given design that must be analyzed in order to assess design options in order to ultimately execute an efficient development process to create a successful, producible, and militarily relevant weapon system.

Thus, the primary technical challenge that must be met to achieve the goals of the META program is to jointly utilize that diversity of problem aspects, modeling formalisms, tools, and specialized reasoners that are involved in the design process and avail this system to an extended community of system developers. The existing state of the practice of model based engineering (MBE) is to use high fidelity and high value models and tools when they are available, and to rely on highly skilled engineering personnel manage the design process using these tools. This can result in islands of computational modeling connected by human-mediated process. These high-fidelity, specialized reasoners (for example tools that compute design details) and solvers (tools that compute behavior from properties expressed as a model) are critical to achieving high quality and successful designs, especially when the use of physical prototyping and testing must be limited. Replacing the high value specialized tools with homogeneous modeling environments would greatly reduce the quality and even more significantly increase the development and production risk of the designs produced. Consequently, improving upon existing practice requires relieving engineers of routine tasks, using computational evaluations of alternatives to enhance human design expertise, and facilitating communication where the need for joint decisions slows existing processes. This

means in particular that automation must bridge across model-based computational islands by enabling joint use of heterogeneous computational resources.

### 1.3 General Methodology

Phase 1b of the META project was focused on implementing the notional ARROW Architecture that emerged from Phase 1a via a spiral development process. We proceeded by implementing key software components, and testing and demonstrating the utility of the overall system concept. We demonstrated our results at the bi-monthly PI meetings starting in January and reviewed them with the DARPA program management team.

Specifically, at the January PI meeting we presented our concept of the ARRoW System for initial review; at the March PI meeting we presented the ARRoW Concept of Operations on the design of a Ramp for an IFV. At the May PI meeting we presented an end to end version of the ARRoW System that instantiated all key technologies used to implement the ARRoW architecture including key underlying technologies of the ARRoW Architecture - and demonstrated the ARRoW System on the design and verification of key use-cases of an IFV Ramp. At the July PI meeting we presented the entire ARRoW Toolset, reviewed the Tool Design, Modeling Language and Library requirements, and demonstrated the use of the ARRoW toolset in designing an IFV ramp. Finally, in September we presented a full version of the ARRoW and demonstrated its use in developing a concept for an entire IFV.

In summary, in seeking a design for ARRoW, we assessed the state of engineering and development practice, along with our extensive experience base, for combat vehicles to derive requirements for primary ARRoW functionality. We then implemented theoretically inspired concepts founded on these proven practices. These implementations were then iteratively tested on a series of challenge problems drawn from the design of infantry fighting vehicles. The resulting approach combines advanced concepts from computer science with insights and experience of the systems and domain expert engineering community.

### 1.4 Technical Results

In order to reduce development times for complex cyber-physical systems by a factor of 5, ARRoW encodes the expertise and methods of professional systems engineers and domain experts and provides a capability to:

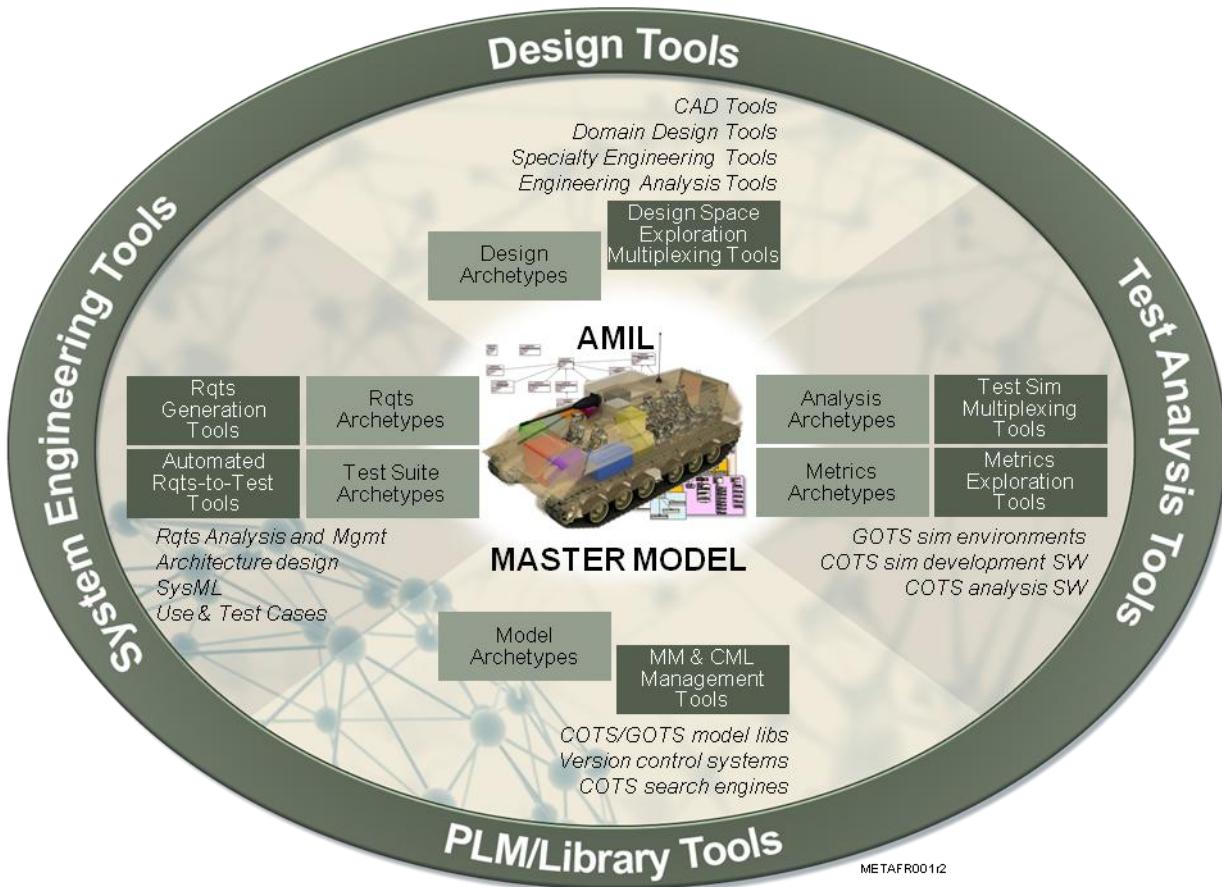
- a. Support multiple, asynchronous workflows
- b. Enable continuous design evaluation
- c. And provide for integrated data using a heterogeneity of tools.

Our approach is built upon a software infrastructure using a collection of repositories and services such that the overall system is extensible, evolvable, and can be applied to a wide range of design problems by populating the libraries appropriately. ARRoW can be deployed across a distributed computing environment (including publicly accessible services supported by cloud computing), allowing multiple independent designers to invoke design and verification tools and make choices that lead to correct by construction designs ready for manufacture. As illustrated in **Figure 1**, the major components in this tool chain are listed below.

- Robust, holistic Model-Based Systems Engineering environment—Capture requirements analysis, categorization, and decomposition through Use and Test cases to establish complete traceability across engineering domains. Facilitate integration of

automated requirements decomposition, test case generation, and architecture and topology design mechanisms.

- Transactional Master Model—Contains a version tree of designs, including an evolving root or baseline design, alternative designs being explored, models of design performance at multiple abstraction levels or domains of performance, and an audit trail of design changes leading to the current state.
- Design Exploration Tools—Components that algorithmically search across design space. Some contain interfaces for human designers allowing them to efficiently and effectively explore design alternatives.
- Verification Tools—Readily extensible verification approaches, including both general and customized methods, are used to assess design choices. These are invoked where appropriate, using Test Cases and resources in the master model to calculate metrics and probability of correctness of design options.
- Patterns and Workflow Archetypes—Multiple collections of established system engineering knowledge in the form of design and analysis patterns or workflows. In order to be applicable to a wide range of tools, models, levels of abstraction, and designs, these patterns and workflows are expressed as archetypes – networks of constraints and requirements that must be satisfied for an instance of a pattern or workflow to be useful.
- Component Model Library (CML)—Repository of versioned artifacts including functional design and specific component models; design and validation environment models; design patterns, and analytic archetypes organized and implemented so as to facilitate the design process. The CML can contain the product of previous design efforts using ARRoW to design systems, sub-systems, or components, as well as archetypes captured from experience with past design efforts.
- Metrics Library—An extensible, continuously maintained collection of metrics deployed in a dynamic, tightly integrated framework. Attributes associated with metrics facilitate the automated matching of metrics to test cases.
- Interconnection Infrastructure—Repository of relationships among all of the tools, models, and design elements active in the system at any point in time. This repository is expressed in ARRoW Model Interconnection Language (AMIL).



**Figure 1. Elements of the ARRoW Tool Chain**

## 1.5 Important Findings and Conclusions

There are a number of elements of existing systems engineering and development practice that can potentially have significant impact on the cost and schedule for combat vehicle development. Much of the current processes and mechanisms lack automation, requiring a high level of touch labor. Still other areas are currently only effectively managed by a relatively small handful of experts, much of whose knowledge is not recorded and is passed on only through personal interaction. Because great expertise of this type is developed over time and extensive experience, the holders of such knowledge tend to develop processes and employ specialized tools, which can make the introduction of new tools, systems, or processes a difficult and costly endeavor for development organizations. One of the tenets of the META program is that development processes can be successful without relying so heavily on these small pockets of deep knowledge by leveraging much larger bodies of broader knowledge through crowd sourcing, open source tools, and extensive use of model-based engineering and model reuse.

Finally, it is known that a primary cause for extended system development time is the result of weak and often conflicting source requirements. This, especially when combined with weak conceptualization analysis, results in system development with a high risk of late-discovery

issues, where they cause the most significant cost and schedule impact. Proactive and intelligent requirements analysis and negotiation, combined with system conceptualization capable of extended design look-ahead mitigates risk to ensure successful system development and provide acceleration.

Technologies developed for the ARRoW system provide an infrastructure facilitating computational exploration of designs with continuous test and verification using multiple models, multiple specialized reasoners, libraries of components, design patterns, and workflows. A foundational element of ARRoW, AMIL provides an executable graphical database that supports relationship maintenance among diverse models, designs, reasoners, patterns, and workflows, allowing either local or distributed execution of computations. The innovation of archetypes, allowing design patterns and workflows to be applied across a broad range of designs and stages in the design process, provides an improved means of capturing and automating engineering practice in order to join previously isolated MBE islands and support much more robust early requirements and concept analysis.

This infrastructure facilitates more aggressive use of computation, while reducing the workload on experts, allowing for mixed initiative exploration for good solutions. It provides an extensible basis for automation, allowing new models and tools to be incrementally introduced, and existing tools to be replaced as better alternatives emerge. The resulting infrastructure provides the means to achieve the acceleration in the design of complex systems. And by both enabling distributed design and automating where possible existing systems engineering knowledge, ARRoW provides an opportunity to democratize design, allowing a wider range of individuals contribute their creative insights.

## 1.6 Significant Hardware Development

None.

## 1.7 Special Comments

None.

## 1.8 Implications for Further Research

Complex system design problems have general features across broad ranges of engineering domains, from integrated circuits to complex cyber-mechanical systems. These general features include:

- The centrality of key abstractions results in a stack of abstraction layers across which the design process must operate.
- Design exploration is conducted using models of relatively high abstraction.
- Design verification is conducted using high fidelity models, often specialized for aspects of the design problem.
- Design refinement requires the ability to add details to an abstract design consistent with abstract properties.
- The use of multiple specialized tools is driven by the need to address multiple aspects of design challenges.

ARRoW has pioneered an approach and a suite of tools that address these general properties of complex systems design; the tools have been demonstrated on a series of challenges drawn from the design of IFVs. However, these tools have not yet been used to support an actual design process with users who can provide feedback on needed improvements. Similarly, the system architecture is library centric, allowing for use across a wide range of design challenges. At this point there has not been an application outside of IFV design that would allow assessing the level of challenge in applying ARRoW to alternate design domains. Opportunity to assess ARRoW in the context of live design problems with users not associated with its development and for novel design domains would provide improved understanding of ARRoW utility and identify areas for further development.

Additionally, some aspects of the ARRoW tool set have been shown to be feasible, but require further development to be readily applicable across a wide range of problems. Notable in this regard is the innovation of the analytic archetype where further technology is needed to allow a broad range of archetypes to be automatically instantiated into workflows appropriate and applicable to specific design states, and available tools, reasoners, and models. In particular, automated integration of very high fidelity models is possible, but presents increasing levels of substantive and software integration challenge. As the modeling tools become increasingly specialized, the volume and format eccentricity of data increases significantly. Replacing labor of skilled engineers is achievable with AMIL and Archetypes, but will require extension of currently implemented software methods.

## 2. Introduction

The goal of the META program was to reduce the development cycle time for complex cyber-physical systems (particularly aerospace and defense systems such as aircraft, rotorcraft, and ground vehicles) by a factor of 5 over current cycle times. In order to achieve the >5x metric, a new model-based methodology—Adaptive, Reflective, Robust Workflow (ARRoW)—was implemented based on a novel concurrent design, testing, and validation workflow.

The approach we have taken in developing ARRoW features an architecture designed to allow for flexible interoperation of heterogeneous tools. This data driven infrastructure provides for flexible configuration of the systems as needed for specific purposes. In particular, alternate libraries can allow for International Traffic in Arms Regulations (ITAR) controlled, proprietary, or open versions of the development system. Further, novel models and tools can be incrementally incorporated, and application to new design domains can be accomplished by introducing alternative libraries.

This architecture is built for extensibility by allowing manipulation and enhancement of collections of:

- Component Models
- Design Patterns
- Analytic Workflows
- Metrics
- Specialized Design Tools
- Verification Tools

These libraries and archetypes not only capture system design patterns and architectures, but also capture industry experience and expertise, facilitating both greater automation in service of a 5x reduction in development times, and potentially the democratization of design.

In the remainder of this document, we analyze the characteristics of the IFV design problem, describe each of the elements of the ARRoW system, and discuss the infrastructure that supports their use.

## 3. Methods, Assumptions, and Procedures

### 3.1 Achieving Program Goals

An analysis of industry performance in developing complex cyber-physical systems suggests that there are three main ways that advanced technology could provide speed-ups:

- The early conceptual stages of the design process require a significant amount of communications between customers and engineers, and between engineers with different backgrounds and disciplines. Conceptual design work can involve significant face-to-face meetings and is often characterized by multiple false starts and roll-backs until a concept satisfactory to all interests is finally discovered.
- Problems at a systemic level (emerging “from the seams” between subsystems or engineering/analytic domains) are often not detected until late in the design and development process, when addressing them is most costly in both dollars and schedule impact.
- Many different computational tools are needed to address all aspects of the design problem. Significant skilled labor is invested in migrating results between these tools. The manual execution of these analytic workflows slows the latter stages of design.

The approach taken in creating ARRoW explicitly addresses all three of these opportunities by providing the following:

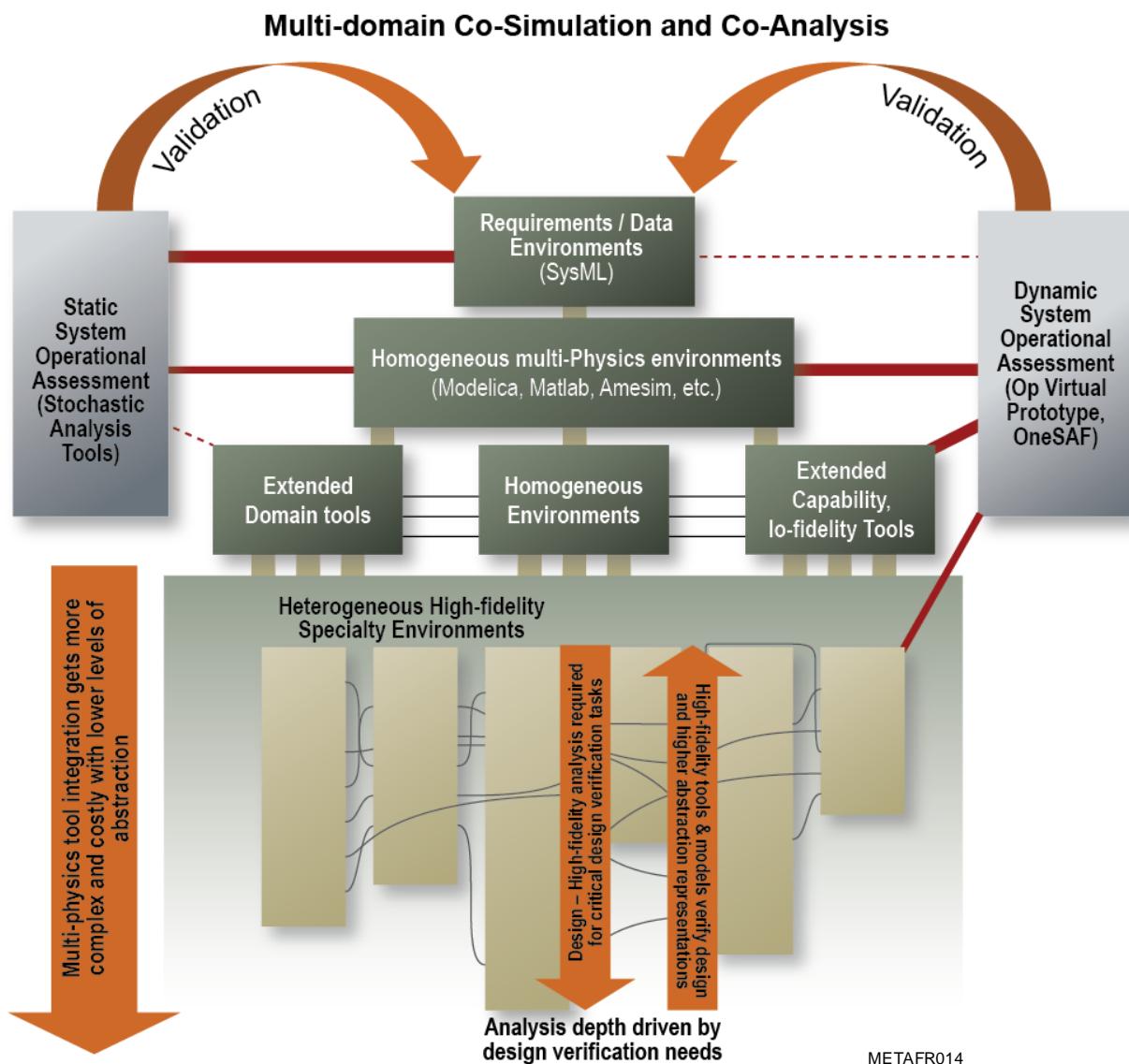
- Explicit support for models and design elements across a range of levels of abstraction. Combined with direct support for design refinement, this allows ARRoW to utilize abstract functional models and design patterns to support conceptual design. (See in particular the description of the ECTo tool, Section 4.2.5 below.) Conceptual designs provide a scaffolding (i.e. derived constraints such as space and weight claims by subsystem) that allow designers to move to increasingly detailed design work without leaving ARRoW.
- Support for continuous testing from abstract through detailed design. Continuous testing provides for detection of problems and their correction much earlier than would otherwise be true. This includes the analysis of requirements, conceptual design at high abstraction levels, as well as integrating the data of the high fidelity multi-domain design environment.
- Capture of the workflow patterns that connect design tools to produce system level tests, diagnosis, and analysis. ARRoW supports workflow capture that can automate routine tasks that otherwise slow design progress and distract expert resources from higher valued tasks. Analytic workflows are abstracted to produce *analytic archetypes* that can be compiled to create an instance appropriate for a given stage of the design process. This allows the resulting system level tests to be consistently available throughout the design process.

Additionally, the ARRoW infrastructure organically supports geographically distributed design. By facilitating interoperation between the many aspects of the design environment, these mechanisms allow elements of the system to be located anywhere accessible via the internet (see description of AMIL, Section 4.3.1 below). Thus, computational resources, models, data, tools, designers and analysts can all interoperate from distributed locations, providing significant potential efficiencies.

### **3.2 Simulation’s Role in IFV Analysis**

Many different models addressing multiple domains of physics and varying levels of abstraction are needed in the designing Infantry Fighting Vehicles (IFVs). High fidelity simulation models capture multiphysics effects that can be missed in more abstract representations and are critical to mitigating development risk and ensuring a design will achieve its requirements. But very high fidelity models can have extreme computational requirements (i.e. multiple supercomputer days per case) which together with the high dimensionality of their input and parameter spaces prohibit their use for design exploration or stochastic estimation of the implications of uncertainties. Models at varying levels of abstraction must be used jointly: abstract models achieve the benefits of rapid design exploration and stochastic verification while high fidelity simulations provide the ability to check for abstraction leakage (problems arising due to effects not seen at abstract levels of representation) and effects due to the interaction of multiple domains of physics. High fidelity simulations at specific design points are used to check for constraint violations and to recalibrate abstract models.

Verification requires the use of high-fidelity, specialty analysis tools. High fidelity models are especially crucial in a paradigm of reduced physical testing. These high-fidelity analyses verify that the design produced is sound and, at the same time, verify that the abstractions in higher levels are correct and accurate. However, in order to produce a balanced design and to capture as much domain and abstraction leakage as possible, system-level co-analysis and co-simulation is necessary, as well. This is critical to support a “continuous validation” development paradigm. Multiphysics/multi-domain system-level analysis and simulation is simplest, most readily achievable, maintainable and executable (for continuous validation) at higher levels of abstraction. It becomes computationally expensive and has diminishing returns at lower levels of abstraction (higher fidelity).



**Figure 2. Tool&Model Abstraction Levels, Integration, and Verification Relationships**

Figure 2 illustrates the relationships across abstraction levels and the flows of information during analysis. Design refinement descends to increasing levels of detail, while the results of detailed simulation flow back up to abstract levels. Even though designs are refined to increasing detail allowing higher fidelity analysis, abstract models remain useful throughout the design process.

The nature of simulation based design work, and the utility of specific tools, varies significantly at different levels of abstraction:

- Relatively abstract multiphysics co-analysis/co-simulation occurs in integrated, homogeneous environments like Matlab, Modelica, Amesim, etc. In the ARRoW system, AMIL provides the system, context, and test case data, and a very simple internal analytic archetype provides the “wiring” connecting models of this type. Multiphysics tool integration is accomplished within the native environment.

- At moderately greater levels of detail, higher fidelity models are required. They can be supported by comprehensive modeling environments, but must be augmented with domain specific and extended capability tools. Supporting the integration of data, tool, and workflow is more complicated here, but readily attainable through development of archetypes and AMIL mapping capabilities.
- Automated integration of very high fidelity models is possible, but presents increasing levels of substantive and software integration challenge. As the modeling tools become increasingly specialized, the volume and format eccentricity of data increases significantly. Replacing labor of skilled engineers is still achievable with AMIL and Archetypes, but requires extension of currently implemented software methods.

Abstract models represent a layer of fidelity that supports system conceptualization and early design refinement. Due to their low to moderate level of fidelity, these models can be represented in a homogeneous language (such as Modelica), and yet cover a significant breadth of design space and domains. As a result of the common, model-based language deployment, the models are readily integrated and contracts can be monitored to lend automation to their integration. When the design progresses past the applicability of these abstract models, requiring domain-specific technologies, methodologies, and high-fidelity required to support detailed design and analysis, the abstract models are retained and maintained to support ongoing system level trades and analyses where the long runtimes of the deep models prove prohibitive. Additionally, these models are retained to support high-level interfaces to and data manipulation from the higher fidelity analysis tools.

The validation of design alternatives also requires models and tools across a range of abstraction levels:

- Static System Operational Analysis employs primarily requirements data and abstract sizing models to answer questions like “how many rounds must this system carry to win this battle” and “a system with these general capabilities has this probability of mission success in this type of scenario”. This type of analysis generally presents no need for high fidelity modeling.
- Dynamic System Operational Analysis employs some low fidelity models, but mostly moderate levels of abstraction and domain specialty models. Customized high-fidelity models are required for design-challenged and/or high-risk areas. This environment requires explicit Computer-Aided Design (CAD)-defined system design, prototype or tactical control and software systems, and a time-managed runtime environment.
  - One of the most crucial applications of these environments is to support and validate warfighter-centric design. Capturing and designing for humans to operate and survive within the system is one of the most challenging but critical design aspects of an IFV.

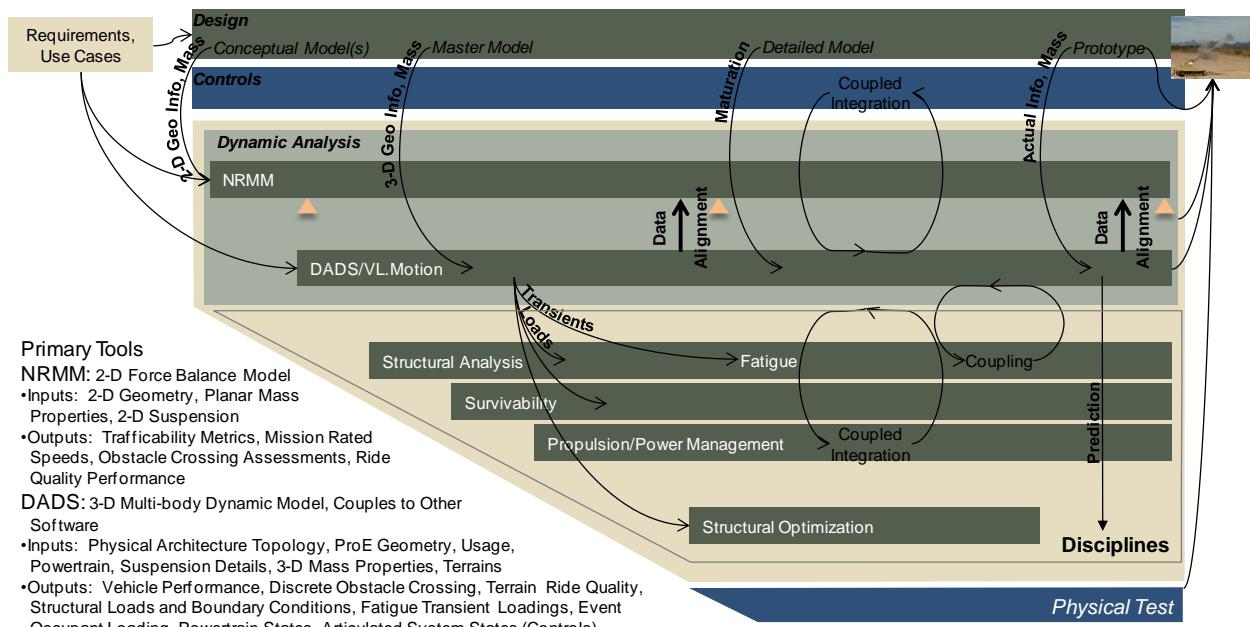
Addressing the full range of multiphysics problems via a small set of models or tools is not possible in part because the phenomena of interest span a wide range of spatial and temporal scales (including frequency ranges) which generally prohibits using a single numerical solver. Regardless of representational approaches, accurate simulation across a thorough range of scales is computationally intractable. When multiple physics with diverse ranges of dynamics are simulated in a single model, abstraction must be used, even in high fidelity models. The extent of abstraction necessary to combine all domains and physics required for the development of an IFV into a single or a small number of solvers is directly in conflict with the level of verification and validation (correct by construction) required to develop these systems

with a minimum of physical prototyping and testing. Thus, for different problems, different abstractions are useful and necessary, implying the need for multiple models, tools, and solvers. Similarly, for co-simulation, different adapters between models of computation are appropriate for different uses. Regardless, no universal modeling framework capable of solving all aspects of IFV design can be expected soon.

### **3.2.1 Illustrative Example**

To provide an example of the use of high fidelity simulation, **Figure 3** diagrams the levels of analysis for mission rated mobility and obstacle crossing. For this problem, powerplant and drivetrain performance is tightly coupled with mission rated speed for traversing terrains and the crossing of discrete obstacles. The coupling identifies the power-limited speed crossing relatively smooth terrains and discovers the balance between power-limits and driver-limits for moderate to severe terrains. Simulations capturing the coupling between these domains can suffice to determine power-limited and driver-limited speeds. By extending those multiphysics integrations to include the structural domain, further goals and requirements can be evaluated. This is achieved by the coupling of the mobility loads to the structural models.

A domain oriented view of this coupling for mobility dynamic analysis is illustrated in **Figure 3**. Within the dynamic analysis domain, two general levels of fidelity tools are illustrated. The NATO Reference Mobility Model (NRMM) provides a relatively abstract representation of mobility as a function of terrain and vehicle design. It provides support for concept exploration and trade studies, without requiring detailed 3-D geometry. Given this lower fidelity tool's level of abstraction, load extraction for structural performance is not possible. Consequently, a higher fidelity tool is necessary, and that higher fidelity tool facilitates the coupling to other domains. These various levels of abstraction complement each other in the design process. Preliminary design choices made based on NRMM are embodied as moderately detailed design, facilitating higher resolution modeling, whose results both provide a higher fidelity check of performance and a basis for adjusting NRMM to incorporate the detailed modeling results. As design refinement proceeds, increasingly detailed structural models provide a basis for high fidelity simulation, while the more abstract model provides a context for other analyses, further trade studies, probabilistic verification against uncertainties, and other services.



**Figure 3. Structural Dynamics Supports Mobility Analysis**

### 3.2.2 Enabling Infrastructure

Current practice solves IFV design problems by using multiple models and multiple modeling tools. Skilled engineers perform chains of analysis, moving data between these formalisms. Thus, a useful approach for solving the pervasive and multivariate multiphysics problems that arise in an IFV design is to capture and systematize the expert engineering knowledge of how to perform chains of analysis like those described above. The captured engineering patterns and workflows inform and constrain data transport and computation, enabling intelligently focused (and thereby computationally tractable) multiphysics calculations. There are multiple ways of viewing the innovations in such an approach, but one that we find illuminating is that this approach tractably computes multiphysics by dynamically and intelligently introducing the necessary abstractions for a given multiphysics analysis. We in effect augment co-simulation with co-analysis, providing a flexible means for capturing important coupled effects. That is, we solve the multiphysics problem by capturing work and data flows between heterogeneous specialty tools. Each of these tools captures specific domains and phenomena for multiphysics coupled simulation. The resulting combination of tools is used to address the systemic requirements constraining IFV design.

ARRoW provides the means to optimize the employment of these powerful specialized tools to realize significant acceleration of the development process. Data is centrally contained within the Master Model and distributed between tools based on relationships described as an AMIL graph. Complex and cooperative engineering analysis processes are captured in Analytic Archetypes. These archetypes facilitate the automated distribution of data, work, and tool flow to minimize engineer labor, accelerate design processes, and eliminate wasted or duplicated effort. This support for analysis tool application takes an objective, forms initial conditions, applies boundary conditions, and executes an analytic process in order to assess a design in terms of one or more metrics. System operational assessment takes processes and relaxes some of the unnecessarily (or unnaturally) rigid boundary conditions to form tests closer to the anticipated use.

Combining these various techniques provides the means for probabilistic verification. Stochastic methods applied to models at suitable levels of abstraction can produce probabilistic estimates of constraint violation where more efficient formal verification techniques cannot be applied. As detailed designs are created, increasingly higher fidelity models can be used to assess the accuracy of, and as necessary, recalibrate more abstract models. Specification of the appropriate analytic workflows is captured in Analytic Archetypes, which provide a basis for automating data transport and co-simulation.

ARRoW provides the software architecture that enables these concepts. Software development principles that have guided the development of the architecture and the prototype implementation of ARRoW include:

- A totally data-driven approach, with no logic specific to the IFV design example contained in any of the tools
  - This requires that data repositories be crafted to ARRoW-specific internal structure, and access methods that facilitate use by ARRoW
- Concealing implementation details within tool boundaries, exposing only the minimal information necessary at the interfaces
- Definitions of tools and services that provide for distributed and parallel design and testing processes, especially ones that facilitate parallel work among geographically distributed designers.

The following section provides detailed descriptions of all the components of ARRoW. Further depth and representative examples can be found in the appendices accompanying this report.

## 4. Results and Discussion

This section describes the various components of ARRoW in terms of both their design and how they operate together to support the design of complex systems. Table 1 lists the major components, together with the function they provide, their underlying technology, and the dates of the Principal Investigator meetings where they were demonstrated.

**Table 1. The Components of ARRoW**

Tool Component	Function Provided	Underlying Technology	Demos (2011)
ARRoW Integrated Development Environment (AIDE) Interface and Dashboard	Designer's graphical interface, providing means to make design choices, invoke design and verification tools, and visualize properties of design alternatives.	Eclipse/SpringHTML, Maven, Subversion, Tomcat, Java	Jan, Mar, May, Jul, Sep
Metrics Library	An extensible set of metrics than can be incorporated in test sets, displayed in AIDE, be the subject of constraints on acceptable designs, or targets design exploration tools can attempt to optimize.	AMIL (also see metrics documentation)	Jan, Mar, May, Jul, Sep

Tool Component	Function Provided	Underlying Technology	Demos (2011)
CAD (Pro/Engineer plug-in)	Provide interface and access between ARRoW tools and Pro/Engineer	Pro/Engineer API, C++	Mar, May
AMIL	Heterogeneous model and tool interconnect	Neo4j, Java/Prolog/C++ API, persistence and caching control. Has associated AMIL graph viewer.	May, Jul
Galileo Test &Verification Tools	Collection of specialized design and verification tools. Responsible for computing Probabilistic Certificate of Correctness and other Diagnostics.	Monte Carlo and Importance Sampling, Context Models, PDF's and Ratio distributions, Reach-set Analysis (MIT), K-means clustering, Expert systems	May, Jul
ESKER	Look-ahead and Design Space Exploration, Adaptability via set-based concurrent engineering, Levels of Abstraction, Language	Expert system state expansion and search, Rule-based design structure matrix, AMIL-aware, variable fidelity modeling, partial decomposition, subjective/qualitative rankings	May, Jul
Envisioner	Qualitative Simulation. Can be used either for design exploration or to efficiently calculate Probabilistic Certificates of Correctness (PCCs)	Lisp	May
SysML (MagicDraw plug-in)	Requirements capture.	MagicDraw API, AMIL-interconnected	May, Jul
CML/Master Model	Repository of design patterns and component models. Provides for Design refinement and Component Reuse	Ontology-based search, Maven/Artifactory delivery mechanism	Jul, Sep
ECTo	Vehicle-level conceiving and prototyping	C++ object hierarchy of generic domain models	Jul, Sep
Metrics Infrastructure and Dashboard	Integrated metrics analysis and calculation services, role and interest-configurable graphical user interface	AMIL-integrated service architecture using Java Standard Object Notation (JSON text metric definition files)	Jul, Sep
Generative Archetype Reasoning (GEAR)	Synthesis, Component Reuse, Domain-specific reasoners for design exploration and analysis.	Semantic Web technologies such as OWL, Description Logic and Declarative Logic Programming, Lisp, SPARQL, Protégé	Sep

Tool Component	Function Provided	Underlying Technology	Demos (2011)
Cloud Deployment	Provide mechanisms to support wide distribution and crowd participation.	Amazon Cloud deployment mechanisms	Sep

In Section 4.1, we present a notional flow through the ARRoW architecture, describing how the various components and tools interoperate to support design processes. Section 4.2 goes through this tool chain a step at a time, providing greater detail. Section 4.3 covers aspects of foundation components of ARRoW that are potentially active throughout the workflow. Finally, Section 4.4 describes notional demo systems that were used to test and motivate ARRoW design and development.

An overview diagram of the ARRoW tool chain is provided in Figure 4. In this depiction we have emphasized the progression that begins with requirements and ends with a design sent to be manufactured at the iFab. Such a depiction by its nature does not reveal the iterative aspect of developments, with many cycles of design exploration and verification. In order to make the diagram interpretable we have also chosen to minimize the number of crossing lines by representing two software entities at multiple locations. Metrics are computed and the CML is accessed at multiple places in the workflow, and these components are represented at a number of locations in the diagram, where in fact there is a single metrics library and a single CML that supports this. However, the fundamental aspects of our tool chain are readily evident in the figure – an open and readily extensible infrastructure that supports asynchronous and parallel processes, distribution of functionality, and integration of heterogeneous tools and reasoners throughout the development chain.

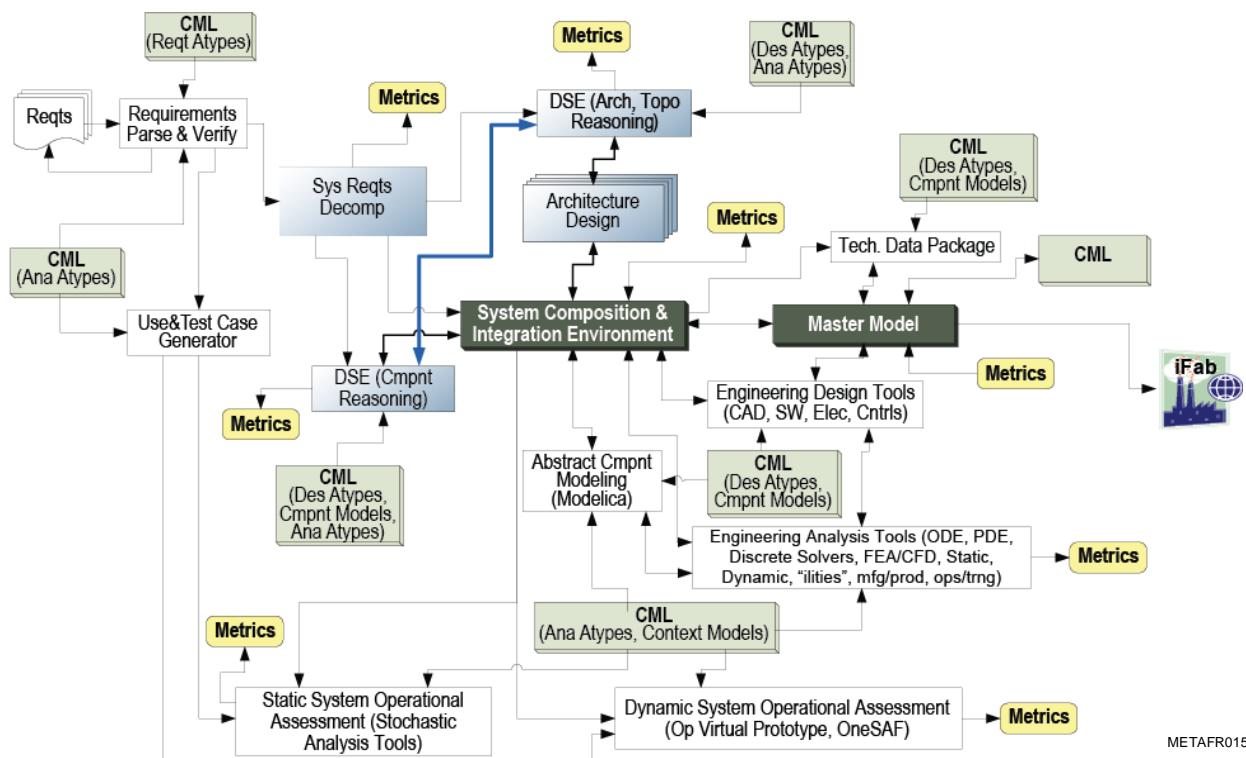


Figure 4. ARRoW Tool Chain – From Requirements to Manufacture

#### 4.1 ARRoW Tool Chain/Workflow Overview

The ARRoW tool chain is founded upon the notion of heterogeneous tool and technology integration and lightweight, unobtrusive data integration mechanisms. This approach enables the employment of fast and automated abstract design methodologies, but retains the capability to access high-fidelity domain-specific tools and capabilities where needed and appropriate for the development of a complex weapons platform like a combat vehicle.

Coordination of the workflows and data constructs are captured and fed through entities known as Archetypes served out of a CML, which not only enable development-accelerating, model and pattern-based design efficiencies and automation, but also facilitate participation by novice designers in local domains. Metrics are generated and captured throughout the system, facilitating asynchronous “continuous verification and validation” to support ongoing design decisions, and to provide for computation of deep high-level system development metrics like Probability of Correctness (PoC) of specific requirements. (Associated with PoCs are documentation for a given probability value or PCC). Finally, the tool chain is ideally suited to the integration of, and augmentation by, advanced reasoning systems to provide further system development acceleration. In particular, several mechanisms have been implemented under the META program, including Knowledge-Based Reasoners, to facilitate more efficient and intelligent design space exploration, and standardized ontologies throughout the system to optimize data and logic access and flow.

This approach offers significant acceleration through all phases of the development process, including the Requirements phase, which contains the most significant potential for acceleration since robust, well understood requirements enable focused and efficient

development activities, while mitigating program-breaking risks and lengthy redesign activities.

A language for expression of relationships among design elements (AMIL) provides a foundation on which distributed tools interact to facilitate the ARRoW vision. This provides for a loose (and hence flexible and readily extensible) coupling among ARRoW tools and models, allowing a broad range of workflows to be supported. AMIL links connect the components models, facilitating model execution for test and verification, enabling design space exploration.

Multiple verification tools and methods can be used simultaneously for aspects of the design problem for which they are suitable. The combination of test cases and design elements contained within the master model determine which verification methods are appropriate to test any particular constraint or design goal. This includes model checking and rigorous model composition methods where they exist and are appropriate. By mapping from test cases to suitable verification methods, various metrics can be produced, including estimates of the probability of correctness. The set of verification tools is easily extensible. At a given time, multiple tools can simultaneously be running test cases against multiple designs, and a given verification tool could also potentially be running multiple test cases in parallel, using the elastic compute bandwidth of the cloud.

Specific verifiers may be appropriate for only certain tests. For example, qualitative simulation may be possible only if there is a relevant model in Qualitative Modeling Language (QML). Some verification methods may only be appropriate for conceptual exploration stages, others may be customized to particular technical tests (e.g. setting up and running a computational fluid dynamics model). Specific tests will be applied through appropriate verification tools, using tool associated attributes that describe their range of utility.

## 4.2 Stages in an ARRoW Facilitated Design Process

ARRoW has the flexibility to support a wide range of workflows. For example, it could be used to analyze the impacts of alternative requirements, or to support modification of an existing design. However, for descriptive purposes, it is useful to go through the canonical process that begins with requirements, goes through multiple iterations of design exploration, verification and refinement, leading eventually to a complete design ready for manufacture. In this section, we describe the steps in this process in greater detail.

- Requirements Ingestion
- Initial Decomposition
- Initial Test Case Generation
- Initial Requirements Reasoning
- System Conceptualization
- System Composition
- Mixed Initiative Design Exploration
- System Detail Engineering Design
- System Operational Assessment

### 4.2.1 Requirements Ingestion

The workflow begins with a set of raw requirements for the System of Interest (SoI), which are generally provided by the SoI customer. These requirements are then ingested into the ARRoW Integrated Development Environment (AIDE), where they are captured in a SysML

tool-readable format. Specifically, when we refer to the “ingestion” of requirements into ARRoW, we mean the process of:

- Importing the raw source requirements from specification documents or other media into the ARRoW IDE Master Model (MM), and
- Capturing the text of each requirement into a unique SysML “requirement” model element if the source requirement is not already in that format.

#### **4.2.2 Initial Decomposition**

After requirements are ingested, they must be processed (“digested”) to be usable in the ARRoW environment. This process begins by searching the CML for comparable Requirement Archetypes. Requirements Archetypes are abstractions of typical requirements constructed for the purpose of reusability. Examples of Requirements Archetypes can be found in the “ARRoW System Engineering and Architecture” appendix (Appendix 7.1) under the section labeled “Requirements and Requirement Archetypes”, and in the table labeled “Sample Requirement Archetype Text” in that same appendix. A possible mechanism for this search process might be to scan the raw requirements for known keywords, and then using the keyword “hits”, to search the CML for Requirements Archetypes containing those keywords.

Requirements Archetypes in the CML are associated with both Requirement Archetype Sets (RASs) and Design Archetypes. Requirements Archetype Sets are logical groupings of Requirements Archetypes. These sets can be related to MIL-STDs, for example “full up” System Performance Specifications, system functions, or common design constraints such as allowable material types, transportability requirements, best practice design standards, product structures, etc. Design Archetypes include reference architectures and/or specific design components. They are abstractions of integrated designs or components constructed for reusability in new systems. Just as decomposed requirements are typically allocated to lower level product structure elements in finished systems, decomposed Requirement Archetypes are pre-allocated to Design Archetypes in the CML.

The AIDE helps the developer to select which Requirement Archetype Sets are most appropriate for the raw requirements imposed on the system. Once the Requirement Archetype Sets are imported (copied into) the Master Model, the raw system-level requirements are allocated to system-level Requirement Archetypes to establish traceability from the ARRoW derived Requirement Archetypes to the requirements provided by the customer. This allocation process could be aided by ARRoW based on keyword associations. Gap analyses are then performed between the raw requirements and the Requirement Archetype Sets to determine if raw requirements are missing or overly constraining. Note that in the event that the raw requirements are originally derived from existing Requirement Archetype Sets in the CML, this process can be quite straight-forward.<sup>1</sup> This process establishes the system-level requirements baseline for the SoI.

Once the requirements baseline is established, the AIDE will assist the developer to select an initial design baseline. Since Requirement Archetypes are allocated to Design Archetypes in the CML, the aforementioned discovery of Requirement Archetypes in the CML that

<sup>1</sup> The AIDE is open to integration of mechanisms that in-process natural language requirements, but that technology was not sufficiently mature to leverage for this phase of the META program.

correspond to customer requirements can be used as a basis to present candidate design solutions to the developer. The developer, in concert with AIDE design mechanisms, then selects and imports the Design Archetypes initially deemed most appropriate for the SoI. When these Design Archetypes are imported, lower level decomposed Requirement Archetypes, pre-allocated to corresponding Design Archetypes and parent Requirement Archetypes, are automatically imported. Design Archetypes are then refined within the Master Model to create specific instances of a design. This refinement process might involve, for example, manipulation of Design Archetypes with design tools or assignment of constant values to design parameters. Completion of the Design Archetype import and refinement process establishes the initial design baseline.

This initial decomposition process provides a significant acceleration of the design process by automating many of the conventionally tedious and manual processes employed for requirements decomposition. This is accomplished by leveraging a fully populated CML to facilitate automated requirements decomposition and to establish an initial design baseline. It additionally provides the structure to facilitate test case generation, and reasoning over the requirements to reduce the Design Space Exploration required in later phases of the development process.

#### **4.2.3 Initial Test Case Generation**

A Test Case is an executable that configures and orchestrates the testing of, and stimulates the inputs of a design component for, the purpose of verifying one or more requirements levied against that component or a product structure parent of that component. Each Requirement Archetype that is allocated to a Design Archetype in the CML has a corresponding Test Case Archetype. When the Design Archetype is imported into the AIDE Master Model, the appropriate Test Case Archetype is also imported automatically.

Test Case Archetypes might be composed of pseudo-code, parameterized functions/services expressed in a general purpose language, Modelica or other solvers, Simulink® or other simulator blocks, SysML parametric diagrams, or any form of expression that can provide a template for the logic of a Test Case. Test Cases are created by refining Test Case Archetypes consistent with the design choices made when design components are refined from Design Archetypes. Test Case Archetypes are abstracted such that the form of their abstraction clearly corresponds to the form of the Design Archetype abstraction.

#### **4.2.4 Initial Requirements Reasoning**

Once requirements have been transformed into interpretable form, designers can begin the process of conceptual design exploration. Alternatively, automated reasoners could also be employed at this point to perform further decomposition of requirements and initial Design Space Exploration (DSE). Relatively simple processing of the requirements can drastically reduce the potential design space that must later be assessed at lower levels of abstraction with slower-running, higher fidelity tools. Examples of this reasoning include:

- If a vehicle's specified top speed exceeds X, it must be a wheeled vehicle (exceeds known limitations of tracked mobility systems).
- If a vehicle is required to "swim" in this marine environment, it must have an aquatic propulsion system.

- If a vehicle must protect X #crew and Y #squad, at *this* level of protection, and be able to traverse *this* terrain, then it will likely weigh in excess of Z and therefore require tracks.

Through this simple logical processing of the requirements, the range of feasible vehicle Design Archetypes (e.g. “wheeled combat vehicle”, “tracked amphibious combat vehicle”, etc.) can be significantly reduced (or even singularly identified), before any significant and time-consuming human analysis is required. The same mechanisms could also (if, for example, one were to treat the above examples as a set), identify non-viable and/or high-risk requirements for immediate customer feedback. Further, information produced by this process can significantly prune the space of possible system components to be explored in subsequent phases of the development process. Information such as this is instantiated in the Master Model for the system being developed and passed via the AMIL to the data management system for continued development.

In the course of the ARRoW project various design space exploration tools have been developed. Approaches tested include use of the parametric capabilities of the SysML tools and the Knowledge-Based Reasoning framework employed in our Expert-System Knowledgebase Evaluation Reasoner (ESKER) design space reasoning tool.

#### **4.2.5 System Conceptualization**

While automated tools can narrow the range of choice, systems engineers and others experts will typically wish to be involved in resolving tradeoffs among alternative conceptual design options. ARRoW provides a graphical tool supporting this exploration, the Early Concepting Tool (ECTo). ECTo is a major component of the AIDE. As illustrated in **Figure 5**, ECTo provides a graphical user interface for system composition and exploration, including mechanisms to browse the CML for design archetypes and components, automatic updating of design by choices from the CML, tracking of high level metrics (in particular cost, weight, and spatial dimension impacts of design choices), and an interactive 3-dimensional representation of the system. ECTo is fully integrated with AMIL, and is capable of either functioning as a downloadable “App”, working exclusively from a local database and local executables, or as a user interface to AIDE, communicating all data and executable functionality via AMIL. Archetypes established during requirements reasoning are communicated, also via AMIL, directly into ECTo, ready for immediate manipulation. Additionally, changes made in other tools (for example a change to a requirement) will be reflected seamlessly in ECTo displays. Design choices made in ECTo will be retained in the Master Model where they will be available to other ARRoW tools (for example the Metrics dashboard, described later).

Once one or more conceptual designs are identified, further refinement of the system can proceed, with multiple, different groups of engineers potentially working in parallel. The conceptual design choices made at this stage provide a scaffolding (derived constraints) that guide later design steps. Should abstract level decisions need to be reconsidered due to more detailed design analysis, this conflict can be automatically detected, minimizing the potential disruption.

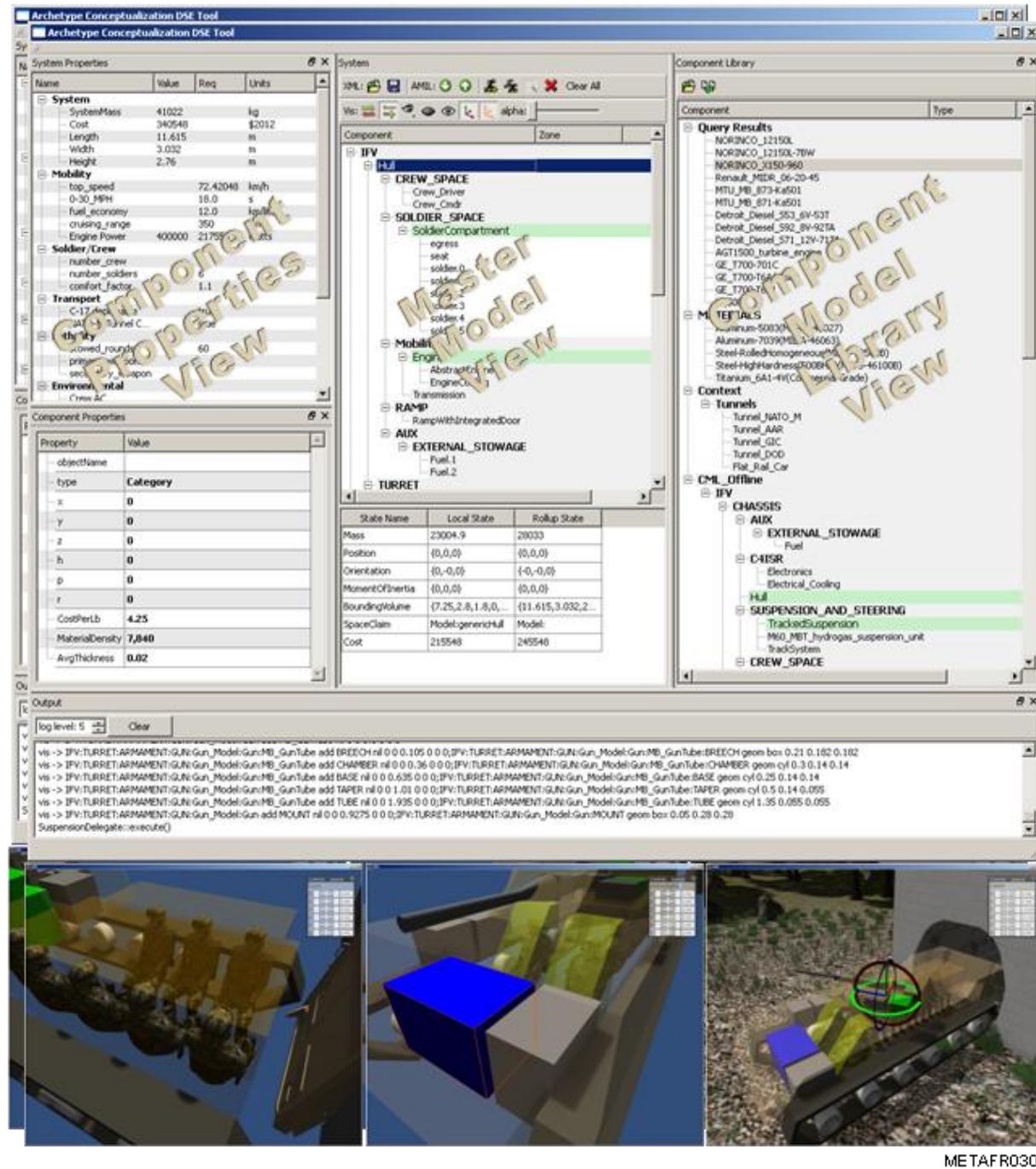


Figure 5. ECTo

#### 4.2.6 System Composition

Beginning with a down selected number of System Archetypes chosen during conceptual design, ARRoW allows a user to rapidly compose the system to begin exploration of more refined concepts. ECTo automatically performs vehicle sizing estimates based on data from the requirements and derived Archetypes, including the System Archetype(s), numbers of crew and squad, levels of required protection, lethality subsystems, and propulsion system. Metrics for vehicle size and weight are continuously monitored and maintained. From this baseline, the

designer can rapidly explore system configuration options and begin the design space exploration activities to refine the design. Overall system configuration information is sent to the architecture engineering system, and the two environments work in parallel to rapidly establish system architectures and component/subsystem topologies.

Within the Design Archetypes are associated high-level abstract models of major subsystems, enabling the designer to begin the process of identifying rough notions of required subsystem capabilities. For example, a “tracked combat vehicle” Design Archetype will contain abstract models of track propulsion system efficiencies, context models, and engine sizing models. This will allow the designer to determine notions of subsystem and component requirements to aid in the reasoning over and selection of components. In other words, the designer will have the capability at this point to determine that a vehicle of *this* approximate size and configuration is required to meet the system requirements, and with *this* type of propulsion system over *this* terrain, will require engines of *this* power range to achieve the required on- and off-road mobility requirements. Note that archetypes and the associated reasoning systems are not prescriptive, and don’t restrict any creativity by the designers. In addition to designing facilitated by the set of archetypes in the CML, new ones are freely created, and the set of archetypes can be extended at any time.

This abstract conceptual development will narrow the design space, essentially establishing requirements (or requirement ranges) of subsystem or component performance to support component search and selection. Elements of the CML will have associated meta-data attributes (related to attributes of design elements in the master model), allowing the discovery of candidate members of the CML for a possible design revision to be achieved through search. The CML is structured through attributes and relationships, not directories and hierarchical typologies. This allows our CML structure to be distributable and dynamic, not monolithic and stagnant. The organizing principles develop according to use, which can take direct advantage of the benefits of crowd-sourcing. Candidate design archetypes or components, deemed feasible given the current state of the design, can be identified using search and discovery (implemented with map-reduce algorithm, for example). In general, the link and relationship-centric design of ARRoW (akin to the architecture of the semantic web) allows both the CML and the Master Model to be segmented and geographically distributed. Consequently, it is possible to make component models available for incorporation into designs and to test these designs without making the models themselves public.

#### **4.2.7 Mixed Initiative Design Exploration**

Once a conceptual design has been selected, more refined design details can be addressed. The scaffolding provided by a design at a given level of abstraction allows parallel design activities to be pursued on subsystems. Previous design choices impose constraints on future design choices, and this information is readily stored as relationship information in the AMIL graph that represents the design. For instance, pre and post conditions on a given design element can be interpreted as assume-guarantee contracts. By this means, ARRoW facilitates the early pruning of design choices based on such constraints, as well as early detection of constraint violation through either static (emergent violation of contracts) or dynamic (constraints violated in a simulation based verification test) means.

These parallel activities can be driven by human designers or algorithmic search algorithms. Architectural optimization leverages the system requirements and previously down selected system Design and Requirements Archetypes, along with information about design options

stored in the CML, to perform its design exploration and refinement process. Algorithmic design exploration will extract component models, or subsystem archetypes (with associated contract patterns) from the CML, filtered by constraints in the current design or design criteria set by either human or algorithmic means. Alternative feasible design choices can be assembled (either by users or automated tools) into candidate architecture refinements that satisfy the constraints, and then tested against computational models and metrics drawn from the metrics library. Various algorithms can be used to iteratively test feasible options seeking designs that optimize multiple objectives. Collaboration with human designers can often best be achieved by search tools that produce trade sets rather than single recommendations. Consistent with ARRoW's overall approach, multiple such reasoning mechanisms can be supported and utilized when appropriate.

Finally, ARRoW is designed to be easily extensible, and new design space exploration algorithms or user design tools can readily be incorporated. Algorithms and user interfaces facilitating specific aspects of the design problem are likely to be of use, and the same infrastructure that supports identifying feasible components for a stage of design can also identify the appropriate tool, reasoner, or solver for a stage of the design process. One reasoning design exploration tool that was created during the development of ARRoW was the ESKER. ESKER is an example of a tool that can incorporate design rules to “break ties”, automating decisions between multiple components that satisfy all requirements but differ in forecasted multi-attribute performance. Such tools can introduce reasoning based on designer intent, such as qualitative preferences for faster vehicles, or lighter vehicles, or systems that can be fielded quickly, or with minimum lifecycle cost.

Facile interoperation between user interfaces, design elements, heterogeneous models and design tools, design space exploration algorithms, the component model library and the metrics library is made possible by the web of connections provided by AMIL. Known relationships and connections are explicitly represented. In particular, access to solvers is provided via AMIL connections to both ECTo and higher fidelity engineering analysis tools. This concept is illustrated in **Figure 6**, with identified commercial and alternate lightweight free or open source tools that could usefully support ARRoW-based design work.

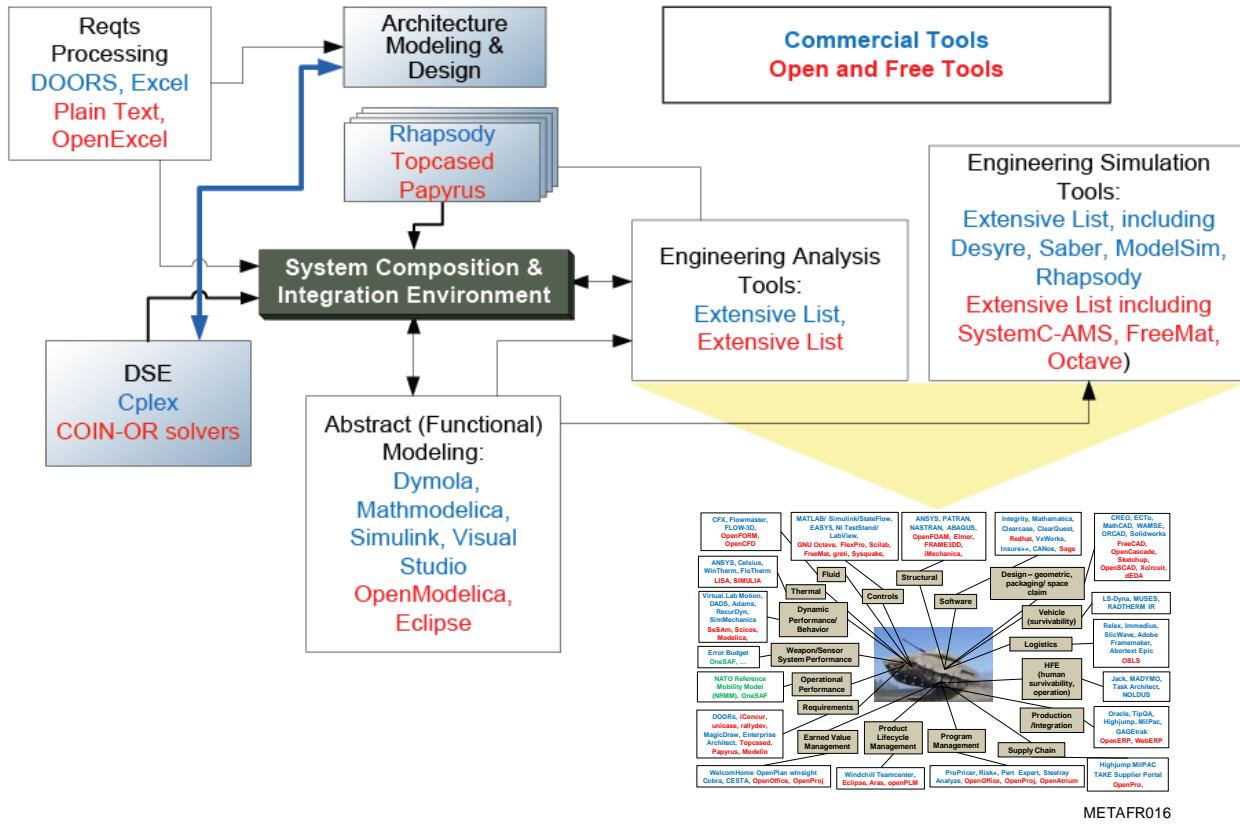


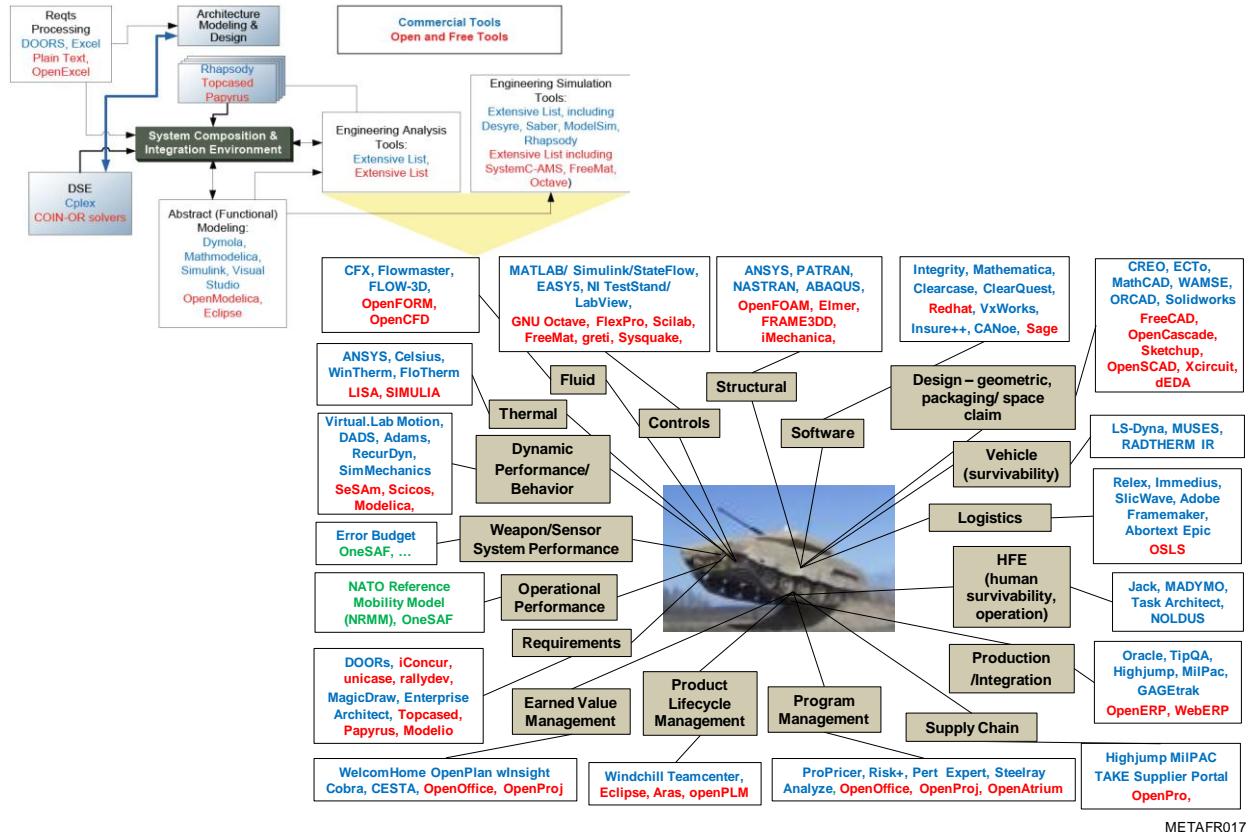
Figure 6. Tool Flexibility in Early Design Phases

#### 4.2.8 System Detail Engineering Design

Ultimately, the conceptual design produced by the largely automated conceptualization tools will need to be refined, analyzed and verified using higher-powered tools:

1. To complete design of any sub-systems that are developed specifically for the this (new) system (e.g. the chassis/hull, component interface and integration mechanism, system level software, etc.).
2. To ensure that potential abstraction leakages that have survived early analysis are discovered and mitigated, and
3. To prepare the system for production and ultimate fielding.

This phase of the tool chain consists of the conventional “heavyweight” engineering design and analysis tools, as they provide what more abstract representations cannot—the necessary domain richness, diversity, depth, and accuracy necessary to ensure that a producible, functional, survivable, and operationally meaningful and compliant system is generated. **Error!** **Reference source not found.** Figure 7 provides an overview of a representative set of these tools, identifying both commercial and lighter-weight free or open source alternative in many of the domains.



**Figure 7. “Heavyweight” Analysis Tools with Potential “Lightweight” Alternatives**

Engineering analysis tools consist of a large number of domain-specific, high-fidelity analysis tools required to ensure that the design is producible, safe, sound, and effective. These include structural, dynamic, fluids, and thermal computational fluid dynamics and finite elements analysis codes; Human Factors analysis tools; and tools for the planning and prediction of manufacturing and production, reliability, logistics, lethality, survivability, etc.

Use of these “best-in-breed” tools identified in **Error! Reference source not found.****Figure 7** is critical to the development of realizable and effective complex combat systems. However, the tool chain provides a means to optimize the employment of these powerful tools to achieve significant acceleration of the development process. Data is centrally contained within the Master Model and distributed between tools via AMIL, and the multitude of complex and cooperative engineering analysis processes are captured in Analytic Archetypes. Essentially, Analytic Archetypes function as recorded “macros” of the deep engineering and analytic processes. Once recorded, they facilitate the automated distribution of data, work, and tool flow to accelerate the processes and eliminate wasted or duplicated effort.

#### **4.2.9 System Operational Assessment**

System Operational Assessment tools provide context model environments to support system-level performance assessment and metric production to ensure that the developing system

meets its system-level operational requirements. These tools support the notion of continuous system validation, ensuring that the integrated system will meet its objectives, mitigating risks associated with abstraction leakage and design/engineering “stovepipe” issues.

#### **4.2.9.1 Static**

Static System Operational Assessment consists of tools designed to operate on abstract system representations, beginning with requirements. These tools are generally stochastic analysis environments and support reasoning systems that are employed to decompose and refine system, subsystem, and component requirements; support early abstract system conceptualization analysis; and system concept refinement (e.g., functional architecture development and component down select and configuration).

#### **4.2.9.2 Dynamic**

Dynamic System Operational Assessment consists of tools designed to operate on explicit system concepts, or those that consist of integrated fully defined components, functioning according to operational use and test cases, driven by software and potentially simulated or real human operators. The OneSAF war-gaming environment is an example of this type of environment. These environments are time-based, and again support continuous system validation as the developing system transitions from a concept to a mature system design. These environments also support system-level human-cyber-mechanical trades and optimization, and provide the development environment for the system being developed Concept of Operations (CONOPS), and operational and training procedures.

### **4.3 ARRoW Foundation Infrastructure**

There are several components of ARRoW that are used throughout its operation, and have not been addressed in depth in the previous walkthrough of the design process. This section describes each in turn:

- AMIL
- The Component Model Library
- Verification Methods
- Metrics

#### **4.3.1 AMIL – ARRoW Model Interconnection Language**

The purpose of AMIL is to automate in a rigorous fashion the joint use of tools, solvers, and reasoners that are specialized for different parts of the design challenge and have very different, possibly incompatible, syntax and semantics. AMIL is based on the same philosophy as the Web in that the key concept is links between information that is not replicated. Links provide information about relationships between models as well as computational dependencies and only need to be established where they are needed.

AMIL links capture the relationships between design elements and can be annotated as needed to capture salient information. In particular, where formal assumptions and guarantees are available for component models this information can be carried in the AMIL graph, and the operations of ARRoW can be guided and constrained by this knowledge. In similar fashion, ARRoW can accommodate a range of formalized approaches to representing component semantics.

The use of AMIL links to represent both design detail and system engineering knowledge provides a number of important benefits. As these links can cross platforms and domain boundaries, this approach allows the master model of the system under development to be distributed, so long as those pieces it is composed of each support the services that allow ARRoW to traverse the design graph. AMIL describes the data that is communicated between the models, which can also be annotated with information about the security that needs to be imposed on each link. Thus, some component models contained in the master model could reside on Windows platforms behind firewalls, while others could run on Linux systems in the cloud. This feature provides opportunities for addressing Intellectual Property, licensing, ITAR, and potentially even security level and classification issues by enabling models to participate in testing without being made publically available.

#### **4.3.1.1 AMIL Implementation**

AMIL provides the data and process communication mechanism upon which other components of the ARRoW system are built. It assumes only that external models and modeling tools provide some sort of access interface and that they are capable of exporting unique identifiers for modeling elements that are relevant across multiple models. AMIL creates proxies for these elements as nodes in an attributed graph. These nodes can be associated with related nodes (proxies of other model elements) with attributed links. The approach represents only the proxies that are required and only introduces the interconnections that are useful. The underlying attributed graph semantics provides a general and extremely flexible foundation in support of the model interconnection semantics. Both nodes and edges of the interconnection network contain lists of key-value pairs that can be extended without arbitrary limit. New relationship types can be introduced on demand without compromising performance.

The physical data store AMIL is built on scales to billions of nodes and relationships. The existing AMIL interpreter can be embedded, or AMIL statements can be executed by invoking the AMIL Web Service. The data transport that we selected follows a standard format and is thus easy to use.

Existing tools can be integrated into AMIL through the use of plug-ins. For example, we have implemented a plug-in for the CAD tool Pro/Engineer (Pro/E and recently renamed Creo), that allows us to utilize Pro/E as a geometry server within ARRoW. The Pro/E plug-in dynamically provides parametric information for generating information about a design such as mass and moments of inertia. A similar plug-in was implemented for the SysML tool Magic Draw, allowing for information (capturing requirements, for example) to be represented and manipulated in SysML and seamlessly used by other tools by through the intermediary of the AMIL database.

#### **4.3.2 Component Model Library**

The Component Model Library (CML) has two primary purposes. First, it is a repository that stores technological knowledge and facilitates its sharing and communication between work threads and components. Second, it encourages re-use of artifacts and makes it easy to do so in a reliable and consistent manner. A centralized component library supports distributed design, because it is available anywhere, and facilitates design evolution, because it is always available.

Design exploration and verification is greatly facilitated by having a Component Model Library (CML) that contains data about available options. Heuristic or combinatorial search can be used for aspects of design that can be framed as a fixed topology in which components from the

library be combined. While developing and populating a complete CML for IFV design was out of scope for the project, we developed requirements, a design, and implemented a prototype CML in order to facilitate our development and demonstration of ARRoW. (See Appendix 7.4.)

In the context of ARRoW, the CML advantageously contains information beyond models and data of physical components. In particular, it is useful to also have archetypes stored there, and to include all relationship information that is available regarding archetypes, components, and models at varying levels of abstraction. As a consequence, the CML is not a hierarchical data store, but also contains a complex web of relationships among its elements. These relationships can greatly facilitate design exploration. Information on how this information can be used can be found in appendices 3, 4, and 6.

Key to using the CML is the ability to search for artifacts that met specific needs. AMIL provides a formal ontology and supports semantic searches of it using a standard language. This is utilized in supporting search of the CML for components or models meeting particular needs. For example, engines meeting specific size and performance requirements, whether in terms of torque, power output or fuel efficiency.

#### **4.3.3 Verification Methods**

Formal verification methods offer sophisticated means of assuring that a design satisfies requirements that can, for some issues, provide significant efficiencies in computing probabilistic assessments of correctness over basic Monte Carlo approaches. Various candidate methods have been investigated during this project (see appendices 3, 9, and 10), and others are being developed by other groups. All of the formal verification methods currently available impose restrictions on models and design and so are currently applicable to only a limited subset of IFV related design issues. Consistent with the overall approach of taking advantage of specialized reasoners, ARRoW provides for the use of formal verification methods where they apply, without requiring that all models or design formalisms employed obey a limiting framework. Conditions for appropriate use of these methods, as with the use of other specialized reasoners, can be captured as analytic archetypes, and included in test sets for routine execution.

#### **4.3.4 Metrics**

Metrics are central to the analysis of design alternatives. Some may be specified by system requirements, others may be selected by designers seeking to better understand properties of their designs. Some metrics can be directly calculated from available models. Others, notably complexity and adaptability metrics and Probabilistic Certificates of Correctness (PCCs), can require substantial additional computation. In order to facilitate ease of use of needed metrics, ARRoW includes:

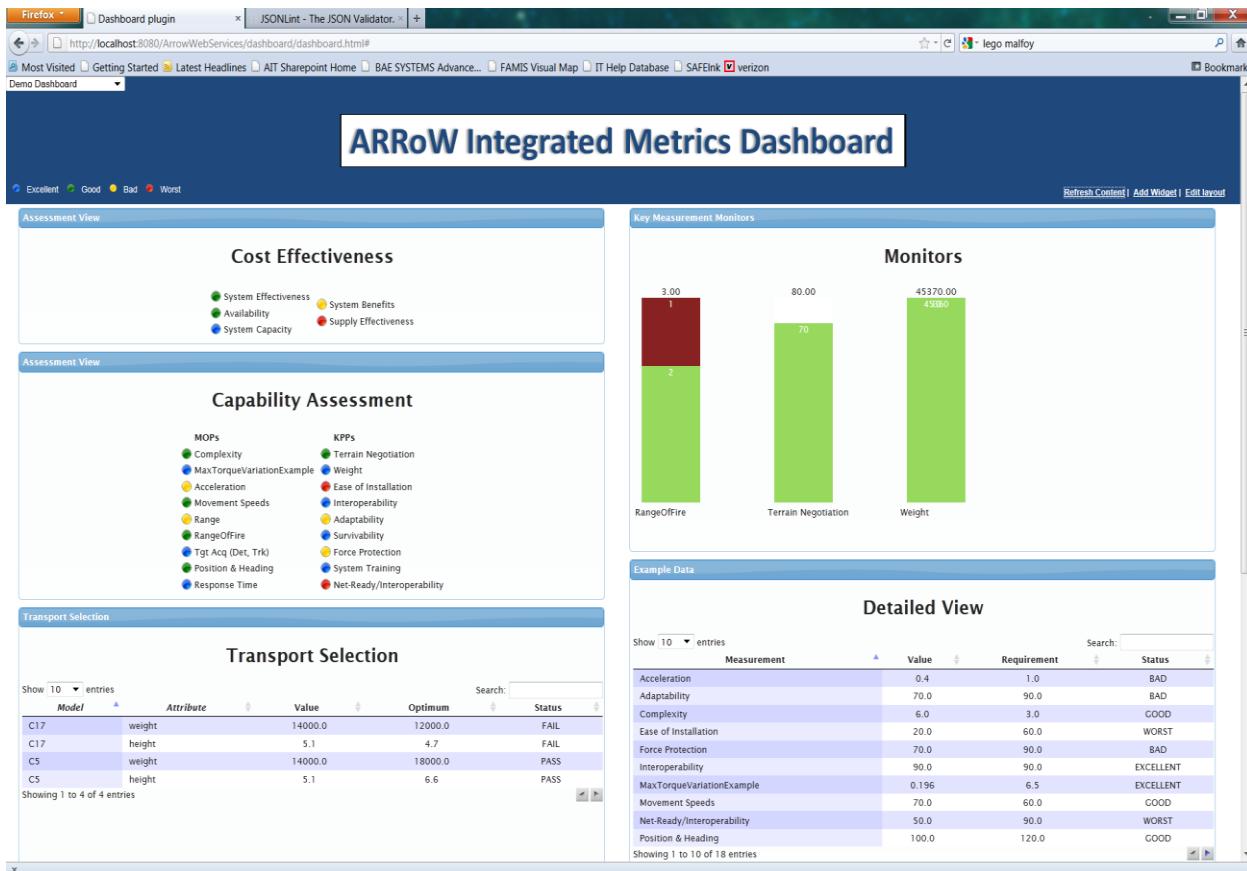
- A metrics library containing algorithms for computing metrics such as robustness and adaptability
- A generic metric model that would allow for ease of metric development and integration by outside users
- An extensible metrics framework to support selection and evaluation of metrics, with supporting infrastructure allowing metrics to pull data from AMIL as needed
- A metrics dashboard to provide continuous graphical display of selected metric values as alternative designs are explored

The ARRoW metrics framework seamlessly integrates metrics with simulation and design tools, hides the back-end AMIL details, is easily re-configurable and supports straight-forward metric creation. The metrics framework is an extension of AMIL, and the same mechanism for integrating 3rd party tools into AMIL is applied to metrics. Consequently, the metrics framework provides the ability to rapidly prototype and integrate new metrics.

An example of the Demo dashboard is shown in **Figure 8**. The various panels within the dashboard are provided by the different views that have been developed, and include: metrics values, assessment of metrics values (excellent, bad, worst), comparison of design alternatives against requirements, and monitoring graphics.

Metrics themselves have many consumers, so the dashboard is easily configurable to support those consumers. Top sponsor leadership and company management may be interested in cost and system effectiveness, and capability and gap analysis, whereas the upper project management are more interested in the health of the design and risk mitigation metrics. Direct line management will be interested in reviewing tracking book metrics.

To support the varying needs of the different metrics consumers, three demonstration dashboards were implemented during ARRoW development: Demo, Design, and Requirements. The Demo dashboard is used to demonstrate a sample of a top level cost analysis and capability assessment. The Design dashboard relies on the outputs of an ECTo design exported to AMIL and presents the results of a mobility model and other design metrics such as calculated total cost and total weight. The Requirements dashboard provides a table of integrated Signal complexity results.



**Figure 8. Example Dashboard Configuration**

Appendix 7.2 contains details about metrics implementation in ARRoW along with brief descriptions of the metrics implemented during META ARRoW Phase 1b. A much longer exposition on promising complexity and adaptability metrics can be found in the META ARRoW Phase 1a Final Report.

#### 4.4 Notional Demo System Application

The ARRoW system development used a series of design challenge problems in order to illustrate key capabilities in the context of a non trivial problem that exists in the combat vehicle design space today. The most thoroughly investigated challenge problem was that of designing the egress system for an IFV squad.

The Ramp challenge problem:

- Considered alternative solutions for the ramp assembly and its supporting subsystems and components to enable subsequent selection of solutions for further design exploration
- Considered trade-offs of multiple inter-related subsystems/component alternatives in order to realize an optimal solution for a given set of criteria
- Used realistic requirements in order to emulate a real design process and in order to calculate correctness of solution alternatives

The various versions of the IFV Ramp challenge problem allowed ARRoW developers to address a highly scalable problem containing contributions from every design domain, including cyber-physical subsystems, and an operational context. These problems were addressed across a broad range of abstractions, including high fidelity simulations of ramp dynamics and mechanical deformations and drive response to operational stimuli including the impacts of footfalls of soldiers running down the ramp. Together with conceptual design problems demonstrated with ECTo, these problems span a range of realistic problems, establishing the viability of ARRoW to support the design of an entire system at scale, given the necessary models, archetypes, and a complete component model library.

## 5. Conclusions

Technologies developed for the ARRoW system provide an infrastructure facilitating computational exploration of designs with continuous test and verification using multiple models, multiple specialized reasoners, libraries of components, design patterns, and workflows. AMIL provides an executable graphical database that supports relationship maintenance among diverse models, designs, reasoners, patterns, and workflows, allowing either local or distributed execution of computations. The innovation of archetypes, allowing design patterns and workflows to be applied across a broad range of designs and stages in the design process, provides an improved means of capturing and automating engineering practice in order to join previously isolated MBE islands.

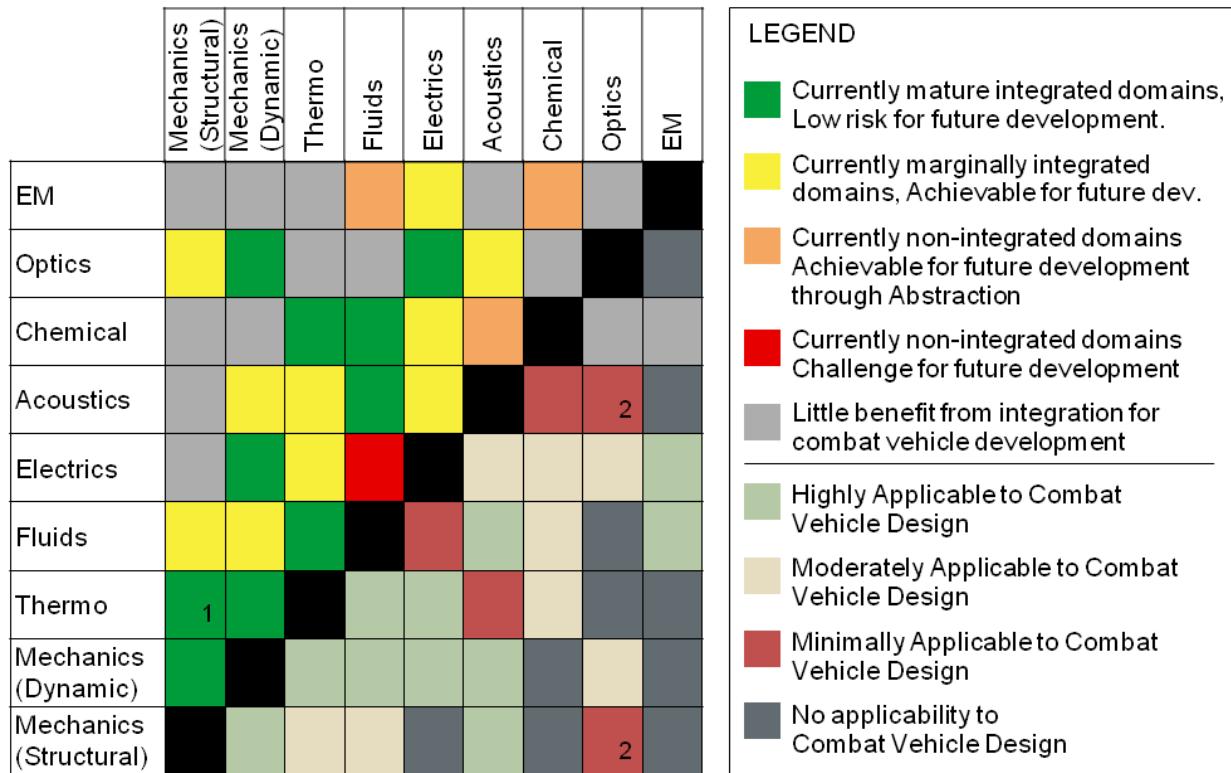
This infrastructure facilitates more aggressive use of computation, while reducing the workload on experts, allowing for mixed initiative exploration for acceptable solutions. It provides an extensible basis for automation, providing more opportunity for innovation and allowing new models and tools to be incrementally introduced, and existing tools to be replaced as better alternatives emerge.

## 6. Recommendations

### 6.1 Integration of Additional Tools

The ARRoW infrastructure provides a foundation in which multiple models, solvers, and reasoners can be jointly used to solve difficult design and verification problems. In order to exploit this foundation to achieve the goal of 5x reduction in times from requirements to first operational prototype, a large number existing specialized design and analysis tools will need to be connected to ARRoW. The benefits of doing so go beyond the automation of analytic processes currently in routine use, as continuous testing and validation can detect problems much earlier (when they are cheaper to fix, and have less schedule impact), and can potentially facilitate the discovery of superior designs.

At its current level of maturity, ARRoW can usefully support requirements analysis, conceptual design exploration, and the use of models of moderate abstraction to refine designs. Incorporating very high fidelity models and design tools presents additional challenges, both in terms of software engineering, and capturing required systems engineering knowledge. The investment to continue ARRoW development to include such tools could provide improved ability to capture combinations of phenomena. **Figure 99** conveys a unique subjective assessment by engineers at BAE Systems Land & Armaments of both the degree of difficulty and importance for 36 combinations of physical effects in the design and verification of an IFV.



1. Highly integrated at a deep analysis level, but there are problems with deriving accurate abstractions.
2. Structural motion/rigidity greatly affects optical performance, but not vice versa.

METAFR031r3

**Figure 9. Multiphysics Levels of Difficulty/Maturity and Relevance to IFV Development**

Among the classes of multiphysics phenomena portrayed in **Figure 9**, various tradeoffs could be made in prioritizing their implementation. In order to facilitate making these choices, we have identified the most significant deep-physics analytical activities necessary for the development of combat vehicles. The following list synopsizes the most important analyses that involve high fidelity multiphysics simulation and analysis for IFV design.

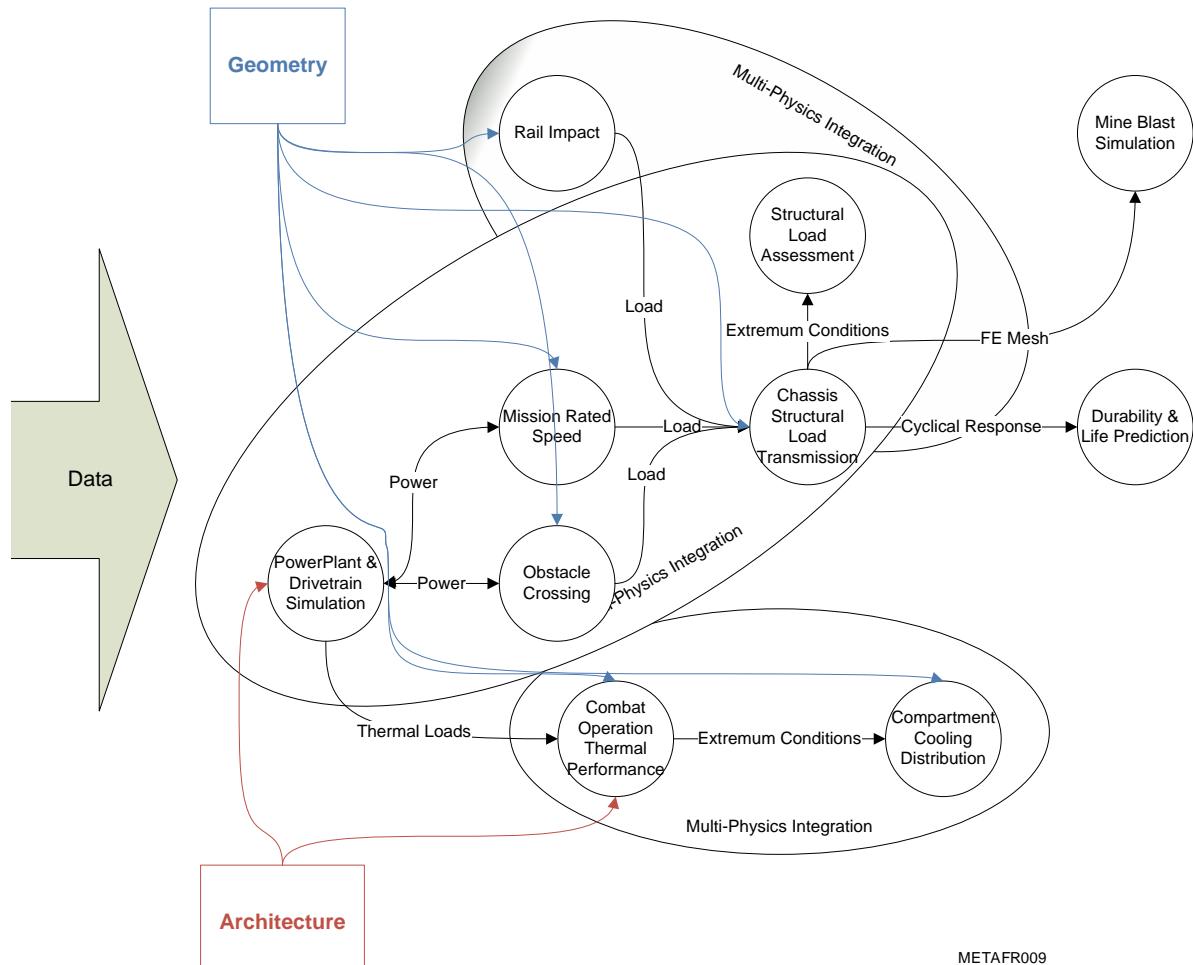
#### Significant deep-physics analytical domains and activities. (Not rank ordered)

- Structural:
  - **Load Assessment.** Assess structural integrity for static load conditions
  - **Load Transmission.** Determine transfer function through chassis to mounted components and subsystems from external shocks and vibrations
  - **Durability and Life Prediction.** Accumulate and assess typical operational loading for the combat vehicle on the structure(s)
- Survivability:
  - **Mine Blast Simulation.** Predict and assess the chassis structural performance when exposed to under-belly explosion

- **Armor:** Prediction of armor recipe performance
- Automotive – Land:
  - **Static Attitude.** Ensure ground clearance and vehicle attitude/wheel loading under various Gross Vehicle Weight (GVW), lift load, grade and slope conditions
  - **Mission Rated Speed.** Predict the maximum speed, governed by crew acceleration and vibration, crossing a given terrain
  - **Obstacle Crossing.** Simulate the go/no-go performance for the vehicle crossing mobility obstacles
  - **Drivetrain/Powerplant Simulation.** Simulate powerplant and drivetrain for mobility and obstacle events for performance assessment
- Automotive – Aquatic:
  - **Water Speed.** How fast can the vehicle travel (and maneuver).
  - **Buoyancy.** Buoyancy reserve and self-righting
- Thermal:
  - **Combat Operation Thermal Performance.** Simulate the thermal exposure and response for crew and components during a typical combat mission
  - **Compartmental Cooling.** Assess airflow and temperature distribution for an extreme thermal loading condition.
  - **Powerplant Cooling.** Simulate the thermal load and ambient heat rejection for various combat missions
- Towing and Transport:
  - **Rail Impact.** Simulate the vehicle rail transportability scenario of a “Hump” test
  - **Transport Tie Down.** Longitudinal pull load
  - **Towing.** Performance towing similar/like kind vehicles
  - **Towed.** Performance assessment under tow

The subjective estimate of our engineers is that this list captures 80+% of the deep, high-fidelity multiphysics based analytical work needed to support design and verification of an IFV. The list is dominated by system level analysis, as this level of performance typically involves multiple sub-systems, and consequently a large number of domains. Description at this level is most readily done in terms of system level mechanics, but note that for each member of the list there are multiphysics effects, and multiple design domains, frequently including control systems and electronics.

Note that the classes of verification listed above have overlaps in the models and kinds of information they would use. **Figure 10** diagrams some of the overlaps and commonalities we will exploit in implementing this list of analytic tests. The shading of the ellipses indicates the relative ease of coupling. Darker shades indicate less challenging integration. Topics in the ellipses of lighter shades provide more of a challenge, but we believe these also can be met with additional investment.



**Figure 10. Overlap Among Multiphysics Modeling and Analysis Topics**

## 6.2 Application to Other Domains

ARRoW has been designed to support design of any complex system for which the necessary libraries are provided. However, testing and development to date have all been specific to IFV design problems. Testing ARRoW's applicability to other domain would both confirm the achievement of this goal, and potentially reveal architectural revisions that would improve its utility.

## 6.3 Achieving Industry Reform and a 5x Compression in System Development Time

In order to achieve significant improvements in the nation's ability to rapidly design and field new weapon systems, innovation is needed in both the processes and supporting tools used in designing these systems. By providing a means to connect what are presently isolated applications of model based engineering, ARRoW is an important innovation of this kind. Capturing the promise ARRoW presents will require combining it with existing commercial tools, open source and academic innovations, and mechanisms, allowing private models and reasoners to be used in a more open context. This combination will provide a means for defense companies, whose greatest expense is their engineering talent, to get more leverage from this

resource and subsequently become more agile, faster, and cost effective, while at the same time, tapping the innovation potential of the “crowd” through facilitating democratized design. By establishing the competitive advantage of reducing their footprint and creating more value sooner, the industry can become an active agent in its own reform.

Additionally, the existence of software platforms that also allow interoperation of open source and commercial tools will bring market forces to bear, leading over the longer term to discovery of the best balance between open source and commercial design tools. As with the historical case of Electronic Design Automation (EDA), it is the positive engagement between systems of open innovation and commercial interests that is needed to achieve the long-term goals of Adaptive Vehicle Make (AVM).

The restructuring of defense and software tool industries will involve processes that will continue beyond the end of the AVM program. The use of hybrid approaches for the duration of AVM can promote these processes and lay a foundation that can guide this evolution towards desired long term goals.

#### **6.4 The Hybrid Approach to Democratizing Design**

ARRoW’s open architecture, support for distributed design, the automation of systems engineering practices, together present a possibility of using ARRoW to enable a wider range of individuals to contribute complex design enterprises. Access for a wider range of design talent can be very helpful in discovering innovative solutions. Conversely, professional engineering expertise can facilitate crowd innovation, by providing the context and assistance with highly technical aspects of design problems. ARRoW could be used to promote democratization of design in multiple ways:

- Proven design patterns can be captured and stored in the CML. These patterns provide both the relationship graph of the components in a particular design and the constraints these components impose on each other. These patterns also capture domain expertise and experience, providing for beneficial, potentially innovative contribution from non domain experts.
- Analytic workflows used to calculate important metrics can similarly be captured, providing significant assistance in automating the testing and verification of designs, especially to assess system level properties.
- Dual-use development of user interfaces and tools to serve both professional engineers and the crowd can ensure that the crowd tools are sufficiently powerful to support successful designs, and in particular militarily relevant designs.
- Flexible infrastructure allowing users scalability in the choice of commercial design tools (frequently expensive) and equivalent, albeit often more limited and/or not supported, low-cost or free tools, promotes both outreach to novice participants and the involvement of professionals, who can in collaboration with other crowd users provide important design ideas and insights.
- The participation of professional engineers in the management of crowd sourcing exercises can monitor the course of design activities, potentially providing feedback, assistance, and verification of any system properties not adequately addressed by automated means.

A crowd-sourcing design exercise would illuminate this opportunity further.

## **7. Appendices**

Appendices are included with this report as separate, external documents due to the volume of material. The following is provided as title and number reference summary.

- 7.1 System Engineering and Architecture**
- 7.2 Tool Design**
- 7.3 Modeling Language**
- 7.4 Library Requirements**
- 7.5 System Demonstration**
- 7.6 Advanced Reasoning and Applications of ARRoW Technology**
- 7.7 Metrics Developed by Team Member (BBN)**
- 7.8 Spatial Design Exploration (BBN)**
- 7.9 RMPL (MIT)**
- 7.10 Verification (MIT)**
- 7.11 Programmatic**

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.1 – ARRoW System Engineering and Architecture**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.1 ARRoW System Engineering and Architecture.....</b>	<b>1</b>
<b>7.1.1 ARRoW Systems Analysis.....</b>	<b>1</b>
<b>7.1.1.1 ARRoW IDE (AIDE) Brainstorming Results .....</b>	<b>2</b>
<b>7.1.1.2 Typical Requirements Quality Issues and Corrective Actions .....</b>	<b>3</b>
<b>7.1.1.3 Hard Vs. Soft Requirements.....</b>	<b>13</b>
<b>7.1.1.4 ARRoW Behavioral Analysis.....</b>	<b>16</b>
<b>7.1.1.5 Example Development Metrics .....</b>	<b>37</b>
<b>7.1.1.6 Design Concept Discriminator Analysis.....</b>	<b>48</b>
<b>7.1.1.7 META Project Product Breakdown Structure.....</b>	<b>50</b>
<b>7.1.1.8 Notional IFV System of Interest.....</b>	<b>53</b>
<b>7.1.1.9 IFV Reference Architecture SysML Model.....</b>	<b>92</b>
<b>7.1.2 ARRoW Architecture.....</b>	<b>92</b>
<b>7.1.2.1 Archetypes .....</b>	<b>92</b>
<b>7.1.2.2 ARRoW Context Diagram.....</b>	<b>103</b>
<b>7.1.2.3 AIDE Entities .....</b>	<b>104</b>
<b>7.1.2.4 ARRoW Master Model .....</b>	<b>111</b>
<b>7.1.2.5 ARRoW Library Elements.....</b>	<b>113</b>
<b>7.1.2.6 ARRoW Requirements to Test Case Flow Architecture.....</b>	<b>124</b>
<b>7.1.3 Bibliography .....</b>	<b>130</b>

## List of Figures

Figure 7.1-1. ARRoW Actors .....	18
Figure 7.1-2. AIDE Top Level UCs–Actors .....	34
Figure 7.1-3. Rqmts & TC Use Cases .....	35
Figure 7.1-4. RTTC Flow Use Case .....	36
Figure 7.1-5. Design UCs .....	37
Figure 7.1-6. Example Project Scorecard for a Notional IFV .....	38
Figure 7.1-7. Example Individual Metric Template Chart.....	39
Figure 7.1-8. Example Order of IFV Criteria Measures for Formal Decisions/Trade Studies	48
Figure 7.1-9. SysML Package Structure .....	52
Figure 7.1-10. Subversion Repository Structure.....	53
Figure 7.1-11. Example Allocation of Requirements to Ramp Assembly .....	64
Figure 7.1-12. Notional IFV PBS .....	67
Figure 7.1-13. Notional IFV Reference Architecture Components and Properties .....	75
Figure 7.1-14. Notional IFV Mounted Operations Use Cases Diagram .....	82
Figure 7.1-15. Notional IFV Actors Diagram .....	83
Figure 7.1-16. Example Operate IFV Ramp Use Cases Diagram .....	86
Figure 7.1-17. Example Details On Operate Ramp Use Case.....	87
Figure 7.1-18. Operate Ramp before Squad Dismounts Activity Diagram.....	88
Figure 7.1-19. Operate Ramp before Squad Dismounts Interaction Diagram .....	89
Figure 7.1-20. Operate Ramp while Squad Dismounts Activity Diagram .....	90
Figure 7.1-21. Operate Ramp while Squad Dismounts Interaction Diagram .....	91

Figure 7.1-22. Sample Requirement Archetype with Metadata.....	95
Figure 7.1-23. Physical Envelope Verification.....	101
Figure 7.1-24. Operational Virtual Prototype Verification.....	101
Figure 7.1-25. AIDE Context Diagram.....	104
Figure 7.1-26. AIDE .....	105
Figure 7.1-27. Master Model .....	112
Figure 7.1-28. Archetype Library (Notional).....	113
Figure 7.1-29. RTTC Entities - Structure View .....	126
Figure 7.1-30. RTTC Entities - Run-Time Interfaces View .....	126
Figure 7.1-31. ARRoW RTTC Entities .....	127

## List of Tables

Table 7.1-1. List of AIDE Brainstorm Ideas .....	2
Table 7.1-2. Requirement Quality Factor Definitions .....	5
Table 7.1-3. Example Rationale for Requirement Quality Issues .....	6
Table 7.1-4. Examples of Requirements Quality Issues and Corrective Actions .....	8
Table 7.1-5. Requirement "To Be" Key Words .....	14
Table 7.1-6. Sample Forbidden Requirement Words and Phrases .....	14
Table 7.1-7. Examples of Requirements Priority Assignments.....	15
Table 7.1-8. Examples of Requirement Bi-Directional Traceability .....	16
Table 7.1-9. ARRoW Actor Descriptions .....	19
Table 7.1-10. ARRoW Requirements .....	31
Table 7.1-11. Example Metrics for Trade Criteria and Measures .....	40
Table 7.1-12. MRL Definitions.....	43
Table 7.1-13. Example Ranges for Mobility Performance.....	49
Table 7.1-14. META Product Breakdown Structure Elements .....	50
Table 7.1-15. Example of Notional IFV Requirements in a SysML Model.....	55
Table 7.1-16. Additional Examples of Notional IFV Requirements.....	56
Table 7.1-17. Notional Ramp Assembly Requirements in a SysML Model .....	62
Table 7.1-18. Notional IFV PBS Elements.....	68
Table 7.1-19. Initial Example of Mobility Components and Properties.....	74
Table 7.1-20. Power Package/Power Train Subsystem Components and Properties .....	75
Table 7.1-21. Steering & Braking Subsystem Components and Properties.....	76
Table 7.1-22. Suspension Subsystem Components & Common Properties .....	77
Table 7.1-23. Suspension Subsystem Component Unique Properties.....	78
Table 7.1-24. Example Actor Descriptions.....	84
Table 7.1-25. Sample Requirement Archetype Text .....	96
Table 7.1-27. AIDE Block Descriptions .....	106
Table 7.1-28. Master Model Descriptions .....	112
Table 7.1-29. Archetype Library Descriptions .....	115
Table 7.1-30. ARRoW RTTC Entities Description .....	128

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
AFSC	Air Force Systems Command
ACAT	Acquisition Category
AIDE	Arrow IDE
Ao	Operational availability
AoA	Analysis of Alternatives
APG	Aberdeen Proving Grounds
ARL	Army Research Lab
AROC	Army Requirements Oversight Council
BCT	Brigade Combat Team
BII	Basic Issue Items
C2	Command and Control
CDD	Capability Development Document
CDR	Critical Design Review
CFV	Cavalry Fighting Vehicle
CG	Commanding General
CG	Commanding General
CML	Component Model Library
CONOP	Concept of Operations
CPD	Capability Production Document
CRM	Customer Relationships Management
CSCI	Software Configuration Item
CSSV	Combat Service Support Vehicle
CSV	Combat Support Vehicle
CV	Combat Vehicles
DC	Design Component
DCSCD	Deputy Chief of Staff for Combat Developments

Symbol, Abbreviation, Acronym	Definition
DMI	Defense Material Item
DoD	Department of Defense
DRACAS	Defect Reporting and Corrective Action System
DT&E	Development Test & Evaluation
DTLOMS	Doctrine, Training, Leader Development, Organization, Materiel and Soldier
EMC	Electromagnetic Capability
EMD	Engineering and Manufacturing Development
EMI	Electromagnetic Interference
EVMS	Earned Value Management System
FCC	Federal Communication Commission
FIR	Field Incident Report
FRP	Full Rate Production
FSR	Field Service Representative
GPS	Global Positioning System
GUI	Graphical User Interface
HFE	Human Factors Engineering
HLA	High Level Architecture
HSI	Human System Integration
HVAC	Heating, Ventilation, and Air Conditioning
HW	Hardware
IAT&C	Integration, Assembly, Test, and Checkout
ICA	Industrial Capabilities Assessment
ICD	Initial Capabilities Document
ICT	Integrated Concept Team
IDE	Integrated Development Environment
IFV	Infantry Fighting Vehicle
IM	Insensitive Munitions

Symbol, Abbreviation, Acronym	Definition
INCOSE	International Council on Systems Engineering
IPR	In-Process Review
ISR	Intelligence, Surveillance, Reconnaissance
JCIDS	Joint Capabilities Integration & Development System
JROC	Joint Requirements Oversight Council
Kph	Kilometers per Hour
KPP	Key Performance Parameter
KSA	Key System Attribute
LCC	Life-Cycle Cost
LRIP	Low Rate Initial Production
MANPRINT	Manpower and Personnel Integration
MATDEV	Material developer
MBE	Model Based Engineering
MBSE	Model Based System Engineering
MGV	Manned Ground Vehicle
MM	Master Model
MMBF	Mean Miles Between Failures
MOE	Measure of Effectiveness
MOM	Measure of Merit
MOP	Measure of Performance
MOS	Military Occupational Specialty
MOU	Measure of Usage
MRA	Manufacturing Readiness Assessment
MRL	Manufacturing Readiness Level
MSA	Materiel Solution Analysis
MTBF	Mean Time Between Failure
MTBSA	Mean Time Between System Aborts

Symbol, Abbreviation, Acronym	Definition
MTTR	Mean Time to Repair
NATO	North Atlantic Treaty Organization
NCO	Non Commissioned Officer
NRE	Nonrecurring engineering
O&O	Organization and Operation
ORD	Operational Requirement(s) Document
OT&E	Operational Test & Evaluation
PBS	Product Breakdown Structure
PLM	Product Lifecycle Management
PM	Program Manager
PMO	Program Management Office
POC	Point of Contact
QA	Quality Assurance
R&D	Research and Development
RD	Requirements Design
RFP	Request for Proposal
RoF	Rate of Fire
RPG	Rocket-Propelled Grenade
RTTC	Requirements to Test Case
SAC	System Analysis & Control
SBS	System Breakdown Structure
SD	System Design
SE	Software Engineering
SME	Subject Matter Expert
Sol	System of Interest
SVN	Subversion
SVS	Surface Vehicle System

<b>Symbol, Abbreviation, Acronym</b>	<b>Definition</b>
SW	Software
TBD	To be determined
TBR	To be reviewed
TBS	To be supplied
TC	Test Case
TCA	Test Case Archetype
TD	Technology Development
TDP	Technical Data Package
TOE	Tables of Organization and Equipment
TPM	Technical Performance Measure
TRA	Technology Readiness Assessments
TRADOC	Training and Doctrine Command
TRL	Technology Readiness Level
TSM	TRADOC System Management
TTP	Tactics, Techniques, and Procedures
UGS	Unmanned Ground Vehicles
UI	User Interface
UML	Unified Modeling Language
V&V	Validation & Verification

## 7.1 ARRoW System Engineering and Architecture

### 7.1.1 ARRoW Systems Analysis

This section describes the systems analysis performed in META Phase 1b that provides an analytical foundation for the Adaptive, Reflective, Robust Workflow (ARRoW) Integrated Development Environment (IDE) architecture described in section 7.1.2. Our goal was to develop an AIDE system architecture that supports:

- Faster delivery of adaptable systems that are trusted, assured, reliable and interoperable
- New processes, methods and tools to build adaptable Defense Material Items (DMIs)
- Early Concept Engineering
- Model-Driven Design, Model-Based Engineering, and Model Based Systems Engineering methodologies
- An open, virtual, realistic environment for validation and manufacturing
- Scalability from today's manually driven development tools and processes to integration of tomorrow's automation techniques, algorithms, and applications
- An infrastructure that is tool agnostic: it does not prescribe particular tool choices, but provides a framework that supports heterogeneous ARRoW design implementations

The purpose of the ARRoW Systems Analysis effort was to develop ideas to explore and develop for ARRoW technologies, architecture, design, and proof of concept capabilities and tools in the areas of:

- Automated requirements development for a DMI
- Automated selection of a DMI PBS
- Automated requirements allocation to DMI PBS elements
- Automated reduction of design space exploration
- Automated selection of preferred design alternative(s)
- Automated metrics for formal decisions and trade studies
- Automated verification preparation
- Automated reporting and recording of verification results

The ARRoW Systems Analysis on cardinal aspects of Combat Vehicle Development (Requirements Analysis, System Design, Systems Analysis and Control, Verification & Validation) reveals examples of:

- Product development systemic issues (e.g., poor quality requirements, lack of requirements templates and reuse, lack of PBS templates and reuse, lack of development metrics templates and reuse, lack of design concept discriminator templates and reuse)
- Potential product development enablers (e.g., model-based work products, reuse, templates, patterns, archetypes, reference architectures, libraries, automation)

The ARRoW Systems Analysis section includes:

- AIDE brainstorming results
- Typical requirements quality issues and corrective actions

- Hard vs. Soft requirements
- ARRoW Behavioral analysis
- Example development metrics
- Design concept discriminator Analysis
- META Project Product Breakdown Structure
- Notional IFV System of Interest
- IFV reference architecture SysML model

#### 7.1.1.1 ARRoW IDE (AIDE) Brainstorming Results

A series of brainstorming meetings were held that challenged the participants with the following:

***“Create a Revolutionary Approach to Combat Vehicle Development”***

The full list of ideas offered in these brainstorming sessions can be found in the accompanying reference document whose filename is “ARRoW IDE Brainstorming.docx”. Many of the ideas in that document have influenced the analysis and architecture of ARRoW during META Phase 1b.

Table 7.1-1 provides a synopsis of some of the more forward thinking items taken from brainstorming ideas that might influence future development.

**Table 7.1-1. List of AIDE Brainstorm Ideas**

No.	Idea
1	AIDE continuously integrates an acquisition customer User Interface throughout development effort. AIDE should support continuous monitoring of customer satisfaction (validation).
2	AIDE optimizes reporting to management and receiving management approval. AIDE supports continuous monitoring of the health of the project and the design.
3	Develop Expert Systems for every engineering discipline/process (e.g., Safety, Maintainability, Configuration Management specialist, System Engineering, Reliability, Testability). Expert System Agents run in background to continuously assess master model (like a spell checker).
4	Create models (physical and cognitive) of the user for automated trials and feedback.
5	AIDE continuously integrates the Warfighter User Interface throughout the development effort: how is the Warfighter executing the mission, operating the equipment and making use of the system capabilities.
6	Think of iFAB as additionally part of the early development process. AIDE facilitates automated prototyping of hardware, supporting early test so as to detect emergent behavior of hardware not accounted for in software-based models as well as continuous model validation.

No.	Idea
7	Programmable, automated test rigs are integrated with the master model. For example, vibration and temperature test environments can be automatically configured based on system requirements.
8	AIDE will automatically generate engineering charts and diagrams so humans can interpret patterns. AIDE additionally has automated agents that look for these patterns.
9	AIDE will provide a single log-on interface. Based on user roles (e.g., manager, curator, professional developer, crowd source, etc.), AIDE automatically configures the user interface to provide appropriate privileged access to data and applications (e.g., ITAR data, licensed applications, etc.). Within the AIDE, security needs are transparently managed. All models know their security levels and simulations can be run in appropriate environments and with appropriately authenticated/need-to-know personnel.
10	In general, relevant information is available to the developer. All DoD data is accessible by appropriately cleared personnel, facilities, and organizations. All DoD programs share data with each other.
11	Actual Sustainment statistics (e.g., Ai, MTTR, MTBSA, MTBF), Field Incident Reports (FIR), Operations & Support reports) feed into CML and AIDE elements. The AIDE Verification environment generates similar metrics to Sustainment statistics and will be able to directly compare to field maintenance and usage reports.
12	Designers, integrators, Field Service Representatives (FSRs), and end-users have access to a “Review” Web-site to comment on and rate products created by or used by AIDE (ala Amazon.com). This product rating data is associated with CML elements and is readily available in the AIDE to influence future designs. Designers and integrators have access to a “Review” Web-site to comment on and rate Requirements, Use Cases, Test Cases, Component Models, Archetypes, or in general any library element. This rating data is associated with the respective library elements and is readily available in the AIDE to influence future designs. AIDE interfaces with social networks (e.g., Customer Relationships Management [CRM]) to influence the design process.
13	AIDE will allow the user to define an objective function and then, via automated design exploration and optimization, create a design that realizes that function.
14	AIDE can roll-back to any prior point in the development (robust versioning control).
15	AIDE maintains pedigrees of components such as: TRL and supporting evidence, history of demonstrations, manufacturing safety critical audits, model accreditations, etc.
16	AIDE supports “Read Only” test results.

### 7.1.1.2 Typical Requirements Quality Issues and Corrective Actions

A diverse range of quality issues can exist for requirements at the beginning and end of a development phase and during the transition from development to production. Typical DMI

programs such as one for a combat vehicle or an Infantry Fighting Vehicle (IFV) undertake daunting time consuming and labor intensive requirements analysis and definition tasks during the development phase. Requirements engineering can be an extremely complex discipline because of the numbers of stakeholders involved in the development process and the precious use of resources. Although requirements gain quality as the development phase progresses, some requirements do not gain significant ground in maturity.

“Poor quality requirements are costly. Some statistics to illustrate the point:

- 50% of product defects are actually due to requirement errors
- 80% of rework is on fixing those errors
- 30% of devices ship with 50% or fewer of the originally specified features

Gunnar Hofmann, a researcher in the requirements management area, found that ‘successful projects typically allocate 15 to 30% of resources to requirements management activities.’”<sup>[HOF11]</sup>

Successful DMI development programs conduct early and often validation of requirements to enable building and delivering a “right” quality product. Cardinal ingredients to validating requirements are measuring and tracking the quality of individual requirements. Table 7.1-2 defines an example set of factors that have been used to determine the quality and issues of requirements. Table 7.1-3 indicates that multiple quality issues can exist for an individual requirement more often than not. Table 7.1-4 identifies candidate corrective actions to overcome requirement quality issues.

Section 7.1.1.3 proffers a requirements maturity roll-up metric within the “Problem Domain Understanding quadrant” on a sample Combat Vehicle Development Project Scorecard that is a summary compilation of requirement quality issues. Requirements maturity roll-up analysis and requirement quality measurement and tracking provide key project visibility mechanisms into requirements validation progression.

Requirements quality measurement and tracking tools should:

- Address requirements quality issues before the beginning of relevant design phases
- Resolve requirement quality issues before formal decisions are made on design
- Ensure steady progression on requirement quality and maturation
- Early and often project visibility into requirements validation
- Ensure that the right product is built

Table 7.1-2 identifies an example set of requirement quality factors used to determine the quality issues/maturity of a requirement.

**Table 7.1-2. Requirement Quality Factor Definitions**

<b>Example Requirement Quality Factors</b>		
A requirement is immature when it lacks one or more of the following quality factors.		
	<b>Title</b>	<b>Definition</b>
1	<b>Necessity</b>	The requirement specifies an essential capability, characteristic or quality. Unjustified or “nice to have” requirements add cost to the system.

Example Requirement Quality Factors		
A requirement is immature when it lacks one or more of the following quality factors.		
	Title	Definition
2	<b>Conciseness</b>	The requirement states only what must be done. Explanations, justification and definitions go in the rationale attribute.
3	<b>Measurability</b>	The requirement is stated in qualitative, quantitative, or probabilistic terms. If stated qualitatively, it specifies the standard for comparison. If stated quantitatively, it specifies tolerances of quantity or a range of acceptability, not an absolute. If stated probabilistically, it specifies confidence levels.
4	<b>Clarity and Unambiguous</b>	The requirement is stated in terms that are specific and have only one interpretation.
5	<b>Implementation/Design Freedom</b>	The requirement is stated in terms of what is required, not how it will be met, either directly or by implication.
6	<b>Attainability/Feasibility</b>	The requirement can be achieved by one or more concepts within defined program constraints such as cost, schedule or risk.
7	<b>Completeness and Stand-alone</b>	The requirement needs no further amplification for understanding when separated from the other requirements.
8	<b>Consistency</b>	The requirement does not contradict the other requirements at the same level or any requirement above its level. Terminology is used the same way throughout the requirements.
9	<b>Verifiability</b>	The requirement is stated in quantified terms such that it can be verified in one or more of five methods: analysis, modeling & simulation, inspection, demonstration or test. However, requirements should not be specified as tests. The verification is put in the appropriate verification attributes.
10	<b>Singularity</b>	Each requirement is a single thought. Only one “shall” per requirement should be used.
11	<b>Uniqueness</b>	The requirements do not duplicate or overlap other requirements at the same level.
12	<b>Proper Level</b>	The requirement is written for the proper level in the PBS/System Breakdown Structure (SBS).
13	<b>Positivity</b>	Each requirement is written as a positive statement rather than a negative one (i.e., avoid the use of “shall not”).

Table 7.1-3 represents an example of the multitude of quality issues associated with a given requirement that is typically experienced during the development of a complex cyber-physical combat vehicle system.

**Table 7.1-3. Example Rationale for Requirement Quality Issues**

Example Rationale for Requirement Quality Issues		
Performance		

Unless otherwise specified, performance requirements in the following paragraphs shall be met with the Vehicle at maximum weight, resting on a flat, hard, level surface, and over the range of environmental conditions specified herein.

Requirements relating to personnel shall apply to males in the 5th through 95th percentile in stature wearing cold weather gear.

Quality Issue	Definition	Analysis
Conciseness	The requirement states only what must be done. Explanations, justification and definitions go in the rationale attribute.	<p>The 1<sup>st</sup> requirement statement beginning with “Unless otherwise specified ...” is not concise (e.g., a product performance requirement). It is a:</p> <ul style="list-style-type: none"> <li>• test condition for a to be determined (TBD) set of product requirements</li> <li>• generalization statement</li> </ul> <p>This statement is best suited for either section 4 (Quality Assurance Provisions – e.g., the test requirements/approach for product requirements) of a specification or a Verification Plan and associated test procedure.</p>
Measurability	The requirement is stated in qualitative, quantitative, or probabilistic terms. If stated qualitatively, it specifies the standard for comparison. If stated quantitatively, it specifies tolerances of quantity or a range of acceptability, not an absolute. If stated probabilistically, it specifies confidence levels.	<p>The 1<sup>st</sup> requirement statement beginning with “Unless otherwise specified ...” is not explicitly measurable as stated.</p> <p>It does not state qualitative, quantitative, or probabilistic terms.</p> <p>This statement needs to explicitly state the qualitative, quantitative, or probabilistic terms for the combinations of test conditions for each relevant product performance requirement.</p>
Clarity & Unambiguous	The requirement is stated in terms that are specific and have only one interpretation.	<p>The 1<sup>st</sup> requirement statement beginning with “Unless otherwise specified ...” is not specific.</p> <p>It does not identify specific:</p> <ul style="list-style-type: none"> <li>• product performance requirements</li> <li>• combinations of test conditions</li> </ul> <p>This statement needs to explicitly state a finite list of product performance requirements and an exact combination of test conditions for each relevant product performance requirement.</p>
Completeness & Stand-alone	The requirement needs no further amplification for understanding when separated from the other requirements.	<p>The 1<sup>st</sup> requirement statement beginning with “Unless otherwise specified ...” is not complete and standalone. It by itself means nothing unless the other product performance requirements are included.</p> <p>This statement needs to explicitly state a finite list of product performance requirements and an exact combination of test conditions for each relevant product performance requirement.</p>

Example Rationale for Requirement Quality Issues		
Performance		
Quality Issue	Definition	Analysis
Consistency	The requirement does not contradict the other requirements at the same level or any requirement above its level. Terminology is used the same way throughout the requirements.	<p>The 2<sup>nd</sup> requirement statement relating to Human System Integration (HSI)/Human Factors Engineering (HSE) is not consistent with requirements in the HSI/Human Engineering section.</p> <p>This statement needs to be deleted to eliminate contradiction with the HSI/Human Engineering section.</p>
Verifiability	The requirement is stated in quantified terms such that it can be verified in one or more of five methods: analysis, modeling & simulation, inspection, demonstration or test. However, requirements should not be specified as tests. The verification is put in the appropriate verification attributes.	<p>The 1<sup>st</sup> requirement statement beginning with “Unless otherwise specified ...” cannot be explicitly verified as stated.</p> <p>Viable verification methods (analysis, modeling &amp; simulation, inspection, demonstration or test) cannot be identified because the statement does not identify specific:</p> <ul style="list-style-type: none"> <li>• product performance requirements</li> <li>• combinations of test conditions</li> </ul> <p>This statement needs to explicitly state a finite list of product performance requirements and an exact combination of test conditions for each relevant product performance requirement to ascertain viable verification methods for each relevant product performance requirement.</p>
Uniqueness/ Duplication	The requirements do not duplicate or overlap other requirements at the same level.	<p>The 2<sup>nd</sup> requirement statement relating to HSI/HSE overlaps requirements in the HSI/Human Engineering section.</p> <p>This statement needs to be deleted to eliminate duplication with the HSI/Human Engineering section.</p>

Table 7.1-4 provides further examples of typical of requirements quality issues that need requirements refinement and validation. This example set of requirement quality issues includes potential corrective actions to improve the quality of the requirements.

**Table 7.1-4. Examples of Requirements Quality Issues and Corrective Actions**

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
<b>Performance</b>	Unless otherwise specified, performance requirements in the following paragraphs shall be met with the Vehicle at maximum weight, resting on a flat, hard, level surface, and over the range of environmental conditions specified herein. Requirements relating to personnel shall apply to males in the 5th through 95th percentile in stature wearing cold weather gear.	<ul style="list-style-type: none"> <li>• Conciseness</li> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Completeness and Stand-alone</li> <li>• Consistency</li> <li>• Verifiability</li> <li>• Uniqueness/Duplication</li> </ul>	<p>1) Derive and flow down driving design constraint requirements to vehicle and lower level product specifications to ensure respective designs can be used under pertinent operating environmental conditions (i.e., components on hard surface and muddy surface).</p> <p>2) Eliminate duplication and/or consistency issues with the ingress and egress requirement in the HSI/Human Engineering section.</p> <p>3) Derive and flow down HSI/HFE requirements to respective components of the vehicle PBS.</p>
Acceleration	The Vehicle at maximum capacity weight shall accelerate from a standing start with the engine idling to 40 mph in not more than 20 sec under nominal conditions. The vehicle, at curb weight, shall accelerate from 0 to 40 mph in not more than 15 sec.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Completeness and Stand-alone</li> <li>• Verifiability</li> </ul>	<p>1) Derive a vehicle acceleration loads (TBD g's) requirement to flow down to the Rear Egress/Ingress assembly such it retains its closed position while the vehicle is under maximum forward acceleration.</p>
Threat Ballistic Protection	The Vehicle shall provide protection against 14.5 mm machine gun and RPG-7 threats.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Completeness and Stand-alone</li> <li>• Verifiability</li> <li>• Uniqueness/Duplication</li> </ul>	<p>1) Determine threat protection quantification factors (e.g., friend-to-threat range, munitions energy level) to quantify the degree of threat protection by threat type.</p>

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
Rear Egress/ Ingress Assembly	The time required for the Rear Egress/Ingress assembly to fully open or close with the engine running shall not exceed 10 sec. The Rear Egress/Ingress Assembly lock mechanism shall permit single hand locking and unlocking.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity and Unambiguous</li> <li>• Implementation/ Design Freedom</li> <li>• Verifiability Proper Level</li> <li>• Positivity</li> </ul>	<p>1) Develop requirements that are clear, measurable &amp; verifiable, implementation free, stated in a positive manner, and at the right level of the PBS.</p> <p>2) Derive and flow down requirement to respective components (e.g., Rear Egress/Ingress assembly, Rear Egress/Ingress latch) of the vehicle PBS.</p>
Interior Lighting	All interior lights, except the turret panel and turret drive power lights, shall extinguish automatically when either the Rear Egress/Ingress Assembly or the Rear Egress/Ingress Assembly door is opened.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Implementation/ Design Freedom</li> <li>• Verifiability Proper Level</li> </ul>	<p>1) Develop requirement that is clear, measurable &amp; verifiable, implementation free, and at the right level of the PBS.</p> <p>2) Derive and flow down requirement to respective components depending on architecture and solutions (e.g., chassis, Rear Egress/Ingress assembly) of the vehicle PBS.</p>
Driver's Switches & Indicators	The Vehicle shall provide the following analog functions and indicators: a. Rear Egress/Ingress Up/Down switch and unlocked indicator	<ul style="list-style-type: none"> <li>• Clarity &amp; Unambiguous</li> <li>• Implementation/ Design Freedom</li> </ul>	<p>1) Develop vehicle level requirement that is clear and implementation free.</p> <p>2) Derive and flow down requirements (e.g., performance, and internal interfaces) to respective PBS components depending on architecture and solutions (e.g., chassis, driver controls, passenger personnel controls, Rear Egress/Ingress assembly).</p>

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
Failure Handling	<p>After a failure is detected and acknowledged (if required), failure handling shall disable only the functionality affected by the detected failure, and the remaining Vehicle shall continue operation. The Vehicle shall be capable of handling occurrence of multiple failures by disabling the summation of impacted functions. When a failure occurrence impacts the Vehicle functionality, the Vehicle shall inform the personnel of the loss. The Vehicle shall not create conditions that may present an unacceptable risk to personnel or result in serious damage to equipment. The Vehicle shall transition to a safe mode as required. The Vehicle shall remain in a safe mode until crew acknowledgement is received from failure handling Pop-up. The Vehicle shall allow the crew to display the malfunction advisory list.</p>	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Implementation/ Design Freedom</li> <li>• Completeness &amp; Stand-alone</li> <li>• Verifiability</li> <li>• Proper Level</li> <li>• Positivity</li> </ul>	<p>1) Develop failure handling requirements set that are clear, measurable &amp; verifiable, implementation free, and at the right level of the PBS.</p> <p>2) Derive and flow down requirements to respective components depending on architecture and solutions (e.g., electronics, SW, user interface (UI), Rear Egress/Ingress assembly) of the vehicle PBS.</p>
Emergency Operation	<p>The Vehicle shall provide an emergency operation capability to drive the vehicle in the case of electronics failures. Functions required to support driving the vehicle include:</p> <p>a. Rear Egress/Ingress Assembly up/down</p>	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Verifiability</li> </ul>	<p>1) Develop vehicle level emergency operation requirements set that is clear, measurable &amp; verifiable.</p> <p>2) Derive and flow down requirements to respective components depending on architecture and solutions (e.g., electronics, SW, user interface (UI), Rear Egress/Ingress assembly) of the vehicle PBS.</p>

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
Embedded Diagnostics	The Vehicle shall perform embedded diagnostics functionality sufficient to eliminate special diagnostic equipment.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Completeness &amp; Stand-alone</li> <li>• Verifiability</li> </ul>	<p>1) Develop vehicle level embedded diagnostics requirements set that is clear, complete &amp; stand-alone, and measureable &amp; verifiable.</p> <p>2) Derive and flow down requirements to respective components depending on architecture and solutions (e.g., electronics, SW, user interface (UI), Rear Egress/Ingress assembly) of the vehicle PBS.</p>
Climate	The Vehicle shall be capable of operating under the conditions specified in AR 70-38, for the climatic categories hot and basic without a cold start aid, and categories cold and severe cold with an aid.	<ul style="list-style-type: none"> <li>• Clarity &amp; Unambiguous</li> <li>• Completeness &amp; Stand-alone</li> <li>• Singularity</li> <li>• Uniqueness</li> </ul> <p>Traceability Issues:</p> <ul style="list-style-type: none"> <li>• AR 70-38 requirements</li> </ul>	<p>1) Restate into complete &amp; standalone vehicle level operating climatic environmental design constraint requirements that are clear, singular, and unique.</p> <p>2) Flow down the operating climatic environmental design constraint requirements to the Rear Egress/Ingress assembly.</p>
Missing Environmental Requirements	Natural: Ambient Pressure, Temperature Shock, Solar Radiation, Salt Fog, Rain & Hail, Ice & Snow, Winds, Lightning (Direct & Indirect), Sand & Dust, Induced: Weapon/gun firing environmental loads: (shock, vibration, thermal, & blast), non-firing thermal loads, vehicle movement shock and vibration	N/A	<p>1) Develop and/or derive vehicle level natural and induced environmental design constraint requirements.</p> <p>2) Flow down pertinent natural and induced environmental design constraint requirements.</p>

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
MTBF		<p>The MTBF was a missing reliability requirement.</p> <p>Candidate - The Vehicle Mean Time Between Failures (MTBF) shall be greater than 120 hours (Threshold) and 168 hours (Objective).</p>	<p>1) Derive a vehicle level reliability (MTBF) design constraint requirement to enable reliability budget allocations to components on the vehicle PBS.</p> <p>2) Flow down the allocated reliability budget (MTBF) requirement to the Rear Egress/Ingress assembly.</p>
General Safety	The Vehicle shall ensure the highest degree of safety and health consistent with mission requirements throughout its life cycle.	<ul style="list-style-type: none"> <li>• Conciseness</li> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Attainability &amp; Feasibility</li> <li>• Completeness &amp; Stand-alone</li> <li>• Verifiability</li> <li>• Uniqueness</li> </ul>	<p>1) Develop the complete &amp; stand-alone vehicle level safety requirements set (e.g., environmental, material, equipment motion, weapon firing and munitions handling, emitter usage, software, failure modes, electrical, mechanical, explosive).</p> <p>2) Flow down the pertinent safety requirement to the Rear Egress/Ingress assembly.</p>
Transportability	.TBP	Missing other Transportability environmental requirements: e.g., Shock and vibration requirements associated with Air, Sea, & Land modes of transportation	<p>1) Develop the complete &amp; stand-alone vehicle level transportability environmental requirements set (e.g., shock, vibration for air, sea, and land transportation modes).</p> <p>2) Flow down the pertinent transportability environmental requirement to the Rear Egress/Ingress assembly.</p>

Typical Requirements			
Title	Statement	Quality Issues	Corrective Actions
Materials, Processes, and Parts Selection	All materials, parts, and processes selected for use in the Vehicle construction shall be compatible with the safety, performance, and environmental requirements as specified herein.	<ul style="list-style-type: none"> <li>• Measurability</li> <li>• Clarity &amp; Unambiguous</li> <li>• Attainability &amp; Feasibility</li> <li>• Completeness &amp; Stand-alone</li> <li>• Verifiability</li> <li>• Singularity</li> </ul>	<p>1) Develop the complete &amp; stand-alone vehicle level materials design constraint requirements.</p> <p>2) Flow down the pertinent material design constraint requirements to the Rear Egress/Ingress assembly.</p>

### 7.1.1.3 Hard Vs. Soft Requirements

Timely progressive requirements maturation drives success for both project management and systems engineering. Requirements maturity measures the endurance or in other words the hardness or softness of a requirement. A “hard” or long lasting requirement possesses all excellent quality factors, a verb form that obligates commitment to deliver, absence of forbidden words, a priority of importance that the product meets the requirement at delivery, and bi-directionally traceable to substantiated rationale, and higher level and/or lower level requirements. The four cardinal attributes of a requirement that can be used to determine the maturity of a requirement (hardness or softness) are:

- Quality factors
- Verb forms of “To be”
- Forbidden words and phrases
- Prioritization
- Bi-Directional traceability

One or more quality factors that are less than excellent determine that a requirement is soft or not enduring. Refer to table 7.1-6 for quality factor attributes of a requirement.

Verb tense and mood of the verb forms of “To be” used in a requirement statement dictate whether a requirement is hard or soft and to the extent in which a requirement is hard or soft. A requirement verb form of “To be” obligates a commitment that is mandatory, desirable, or optional.

Table 7.1-5 provides requirement “To Be” key word definitions consistent with the International Council on Systems Engineering (INCOSE) Systems Engineering Handbook.[INC10]

**Table 7.1-5. Requirement "To Be" Key Words**

Requirement Verb Forms of “To Be” Error! Bookmark not defined. <sup>2</sup>		
No.	Verb Tense and Mood	Description
1	<b>Shall</b>	A mandatory requirement that originates from a stakeholder. Requirements are demands upon the designer or implementer and the resulting product. The verb “shall” is the imperative form of the verb “to be”. The verb “shall” identifies requirements and requires verification.
2	<b>Should</b>	A statement that conveys a desirable requirement or capability by the customer, compliance is not required. The verb “should” is an indefinite form of the verb “to be”. When developing specifications minimal use of the verb “should” is expected. Use the “Should” statements are not requirements.
3	<b>Must</b>	A customer desire, or possibly a goal, but not a requirement and does not require verification. If “Shall” and “Must” are both used in a requirements specification, there is an implication of difference in degree of responsibility upon the implementer.
4	<b>May</b>	An optional requirement or a statement relating to how the mandated requirements can be achieved.
5	<b>Will</b>	A statement of intent or a statement relating to something outside the scope of the product to be developed, but that is relevant to the product under consideration. A statement containing “will” can be used to identify a future happening or convey an item of information, explicitly not to be interpreted as a requirement.

Forbidden words or phrases lead to ambiguity and determine whether a requirement is soft or not enduring. Table 7.1-6 provides a sample of forbidden words or phrases that is consistent with the International Council on Systems Engineering (INCOSE) Systems Engineering Handbook. [INC10]

**Table 7.1-6. Sample Forbidden Requirement Words and Phrases**

Sample Forbidden Words and PhrasesError! Bookmark not defined. <sup>3</sup>		
No.	Type	Examples
1	<b>Superlatives</b>	Supreme, excellent, fullest, least, outstanding, highest, greatest, best, most, worst, unparalleled, unrivalled, peerless, matchless, unsurpassed, of the highest order, poor, ordinary
2	<b>Subjective Language</b>	user friendly, easy to use, efficient, effective, cost effective, good, readable, seamless, visible, ideal, assist, quick, correct, practicable, consistent, necessary, near, clear, intended, capable
3	<b>Vague Pronouns</b>	he, she, this, that, they, their, who, it, its, which

Sample Forbidden Words and PhrasesError! Bookmark not defined. <sup>3</sup>		
No.	Type	Examples
4	<b>Ambiguous Adverbs and Adjectives</b>	all, full, low, adequate, applicable, appropriate, almost always, better, significant, maximum, minimal, minimum, timely, real-time, precisely, appropriately, approximately, various, multiple, many, few, limited, accordingly, some, high, bad, rapid, easy, complete, incorrect
5	<b>Open-ended Non-verifiable Terms</b>	provide support, but not limited to, as a minimum, sufficient, give, do, provide
6	<b>Comparative Phrases</b>	better than, higher quality, like, equivalent, in order to, includes but shall not be limited to, between
7	<b>Loopholes</b>	if possible, as appropriate, as applicable, however, relevant, could, possible, consider, must, may,
8	<b>Other Indefinites</b>	etc., and so on, to be determined (TBD), to be reviewed (TBR), to be supplied (TBS), and/or, shall not, will be required, would, is

Priority assignment determines whether a requirement is soft or not enduring. Table 7.1-7 provides examples of the priority types:

**Table 7.1-7. Examples of Requirements Priority Assignments**

Examples of Requirement Priority Assignments		
No.	Type	Examples
1	<b>Mandatory</b>	A requirement that is deemed to be imperatively fulfilled by the product.
2	<b>Desirable</b>	A requirement that is deemed to be worth being fulfilled by the product.
3	<b>Optional</b>	A requirement that is deemed to be electively fulfilled by the product.
4	<b>Regulatory or Legislative</b>	A requirement that is deemed to control or governed fulfillment by the product.
5	<b>Tradable</b>	A requirement that is deemed to be partially or zero fulfilled by the product.

The “bi-directional traceability” of a requirement determines whether a requirement is hard or soft. A requirement should have traceability to substantiated rationale or an analytical foundation, and upward or downward traceability to requirements. A low-level, detailed requirement without traceability to a parent requirement is potentially a requirement with no basis for existence (gold plating). A customer or higher-level PBS requirement that does not yield lower level requirements that are either derived, decomposed, or allocated are potentially irrelevant, unrealizable, not having been fulfilled or implemented, or not testable.

Table 7.1-8 provides examples of downward traceability requirements.

**Table 7.1-8. Examples of Requirement Bi-Directional Traceability**

Examples of Requirements Bi-Directional Traceability		
No.	Type	Examples
1	<b>Derivation</b>	Operate vehicle => conserve fuel, fuel carrying capacity Vehicle dash speed (acceleration) => Power Package/Power Train power and torque => Engine power and torque, Power Transport (Transmission) power and torque Transportability => military lift cargo weight limits, airlift and rail lift cargo dimension limits $P_{kill}$ or $P_{raid annihilation}$ => $P_{Detect}$ , $P_{Decide}$ , $P_{Weapon Launch}$ , $P_{Missile Launch}$ , $P_{hit}$ $A_o$ => MTBF, MMBF
2	<b>Decomposition</b>	Operate vehicle => Move, Maneuver, Start, Initialize, Shutdown vehicle => Accelerate, move on highway, move on cross country, turn vehicle, climb obstacle, cross gap/trench Engage target => Initialize weapon, calculate ballistic solution, load weapon, point weapon, fire weapon, return to battery
3	<b>Allocation</b>	Flow down of weight, reliability, environmental conditions budgets
4	<b>No further allocation or decomposition</b>	Transportability, Personnel and training, operator manuals, facilities and facility equipment requirements

A requirement is considered to be soft when its attributes are characterized with one or more of the following:

- One or more excellent quality factors are missing
- Verb forms “should, must, may, or will” are used for the verb forms of “To be”
- Forbidden words or phrases are used in the requirement statement
- Prioritization is determined to be either desirable, optional, or tradable
- Bi-directional traceability lacks substantiated rationale or analytical foundation, or leads to an orphan requirement or childless parent requirement

#### 7.1.1.4 ARRoW Behavioral Analysis

This section describes the analysis performed to discover desired functional capabilities of the ARRoW Integrated Development Environment (IDE). An analysis of external actors was performed to understand the context in which ARRoW will operate as well as to identify automation opportunities. Textual requirements for ARRoW were written and are provided in this section. Use cases were developed to elaborate essential functionality of ARRoW and to identify an emergent logical architecture of ARRoW.

##### 7.1.1.4.1 ARRoW Actors

An analysis was performed to identify actors that historically influence the design and program management of ground combat systems. These same actors potentially might interface, either directly or indirectly, with the AIDE. This section describes the actors that were identified as a result of this analysis.

In this context, “actor” is defined to be an entity that represents the role of a human, an organization, or any external system that that participates in the use of the AIDE. Since an actor represents a ‘role’, it is possible, for example, that a particular person can assume multiple roles, and thus can be represented by multiple actors.

The rich breadth and depth of expertise that traditionally drives the design of combat systems is evidenced by the extensive, but by no means exhaustive, list of actors described herein. It should be noted that any attempt to reduce or eliminate the need for any of these actors, such as through automation techniques within the AIDE, must fill the resultant design-influence knowledge void by other means. Possibilities include:

- Development of expert systems agents as surrogates for these actors
- Development of non-traditional requirements and their attendant test cases to support automated verification that “subject matter expert” design rules are adhered to
- Reuse of configuration managed design components that are strictly accredited in terms of the specific subject matter areas represented by these actors.

Figure 7.1-1 depicts a SysML diagram of ARRoW actors. Generalization-specialization role relationships are shown in this diagram using standard SysML notation whereby arrowheads terminate on the actor with the more general role. For example, at the top of the hierarchy in this diagram is the “ARRoW User” actor – the most general actor depicted. Specializations of the “ARRoW User” include the “Acquisition Community Member” and “SoI Development SME” actors.

A non-normative convention is used in our SysML diagrams to distinguish human actors from nonhuman actors. Human actors are represented with a stick figure symbol, for example:  Nonhuman actors are represented in block form, for example: 

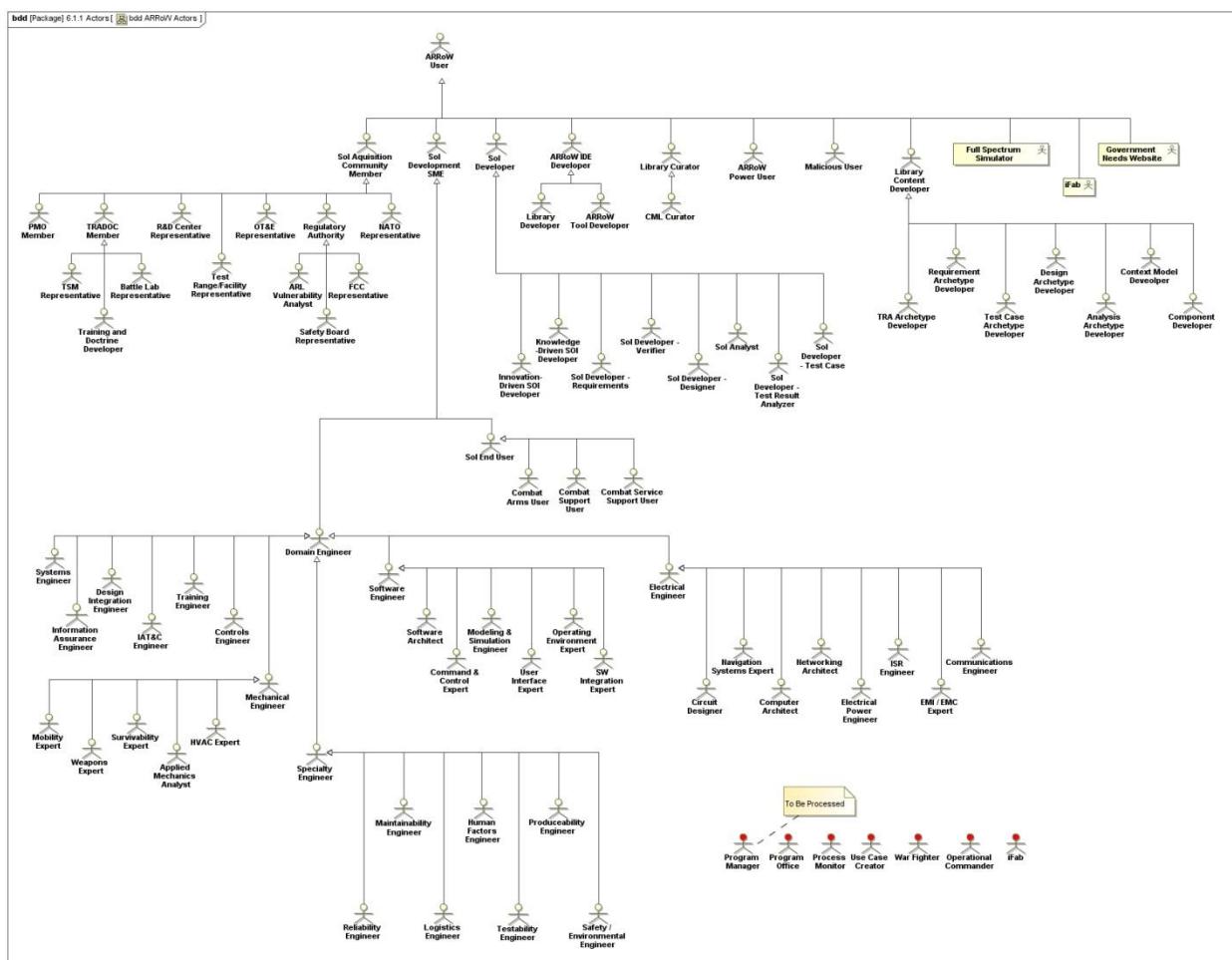


Figure 7.1-1. ARRoW Actors

Descriptions of the actors shown in Figure 7.1-1, alphabetically sorted by actor name, are provided in Table 7.1-9. These actors can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.1.1 Actors” with the description text in the documentation metadata field associated with each actor element.

**Table 7.1-9. ARRoW Actor Descriptions**

<b>Actor Name</b>	<b>Description</b>
Analysis Archetype Developer	A library content developer of any analysis archetype.
Applied Mechanics Analyst	<p>An Applied Mechanics Analyst applies advanced modeling techniques to analyze fluid dynamics, multi-body dynamics, thermal dynamics, shock/vibration analysis, etc. Generally has advanced degree.</p> <p>Typically, engineering mechanics is used to analyze and predict the acceleration and deformation (both <a href="#">elastic</a> and <a href="#">plastic</a>) of objects under known <b>forces</b> (also called <b>loads</b>) or <b>stresses</b>.</p> <p>When treated as an area of study within a larger engineering curriculum, engineering mechanics can be subdivided into:</p> <ul style="list-style-type: none"> <li>• <a href="#">Statics</a>, the study of non-moving bodies under known loads</li> <li>• <a href="#">Dynamics</a> (or <a href="#">kinetics</a>), the study of how forces affect moving bodies</li> <li>• <a href="#">Mechanics of materials</a> or <a href="#">strength of materials</a>, the study of how different <a href="#">materials</a> deform under various types of stress</li> <li>• <a href="#">Deformation mechanics</a>, the study of deformations typically in the <a href="#">elastic</a> range</li> <li>• <a href="#">Fluid mechanics</a>, the study of how fluids react to forces. Note that fluid mechanics can be further split into <a href="#">fluid statics</a> and <a href="#">fluid dynamics</a>, and is itself a subdiscipline of <a href="#">continuum mechanics</a>. The application of fluid mechanics in engineering is called <a href="#">hydraulics</a>.</li> <li>• <a href="#">Continuum mechanics</a> is a method of applying mechanics that assumes that all objects are continuous. It is contrasted by <a href="#">discrete mechanics</a>.</li> </ul> <p><a href="#">[edit]</a> Major topics of applied mechanics</p> <ul style="list-style-type: none"> <li>• <a href="#">Acoustics</a></li> <li>• <a href="#">Analytical mechanics</a></li> <li>• <a href="#">Computational mechanics</a></li> <li>• <a href="#">Contact mechanics</a></li> <li>• <a href="#">Continuum mechanics</a></li> <li>• <a href="#">Dynamics (mechanics)</a></li> <li>• <a href="#">Elasticity (physics)</a></li> <li>• <a href="#">Experimental mechanics</a></li> <li>• <a href="#">Fatigue (material)</a></li> <li>• <a href="#">Finite element method</a></li> <li>• <a href="#">Fluid mechanics</a></li> <li>• <a href="#">Fracture mechanics</a></li> <li>• <a href="#">Mechanics of materials</a></li> <li>• <a href="#">Mechanics of structures</a></li> <li>• <a href="#">Rotordynamics</a></li> <li>• <a href="#">Solid mechanics</a></li> <li>• <a href="#">Soil mechanics</a></li> <li>• <a href="#">Stress waves</a></li> <li>• <a href="#">Viscoelasticity</a></li> </ul>
ARL Vulnerability Analyst	<p>ARL = Army Research Lab</p> <p>A Vulnerability Analyst will analyze and review integrated solutions of armor, spall liners, component placement, and hull design to assess force</p>

Actor Name	Description
	<p>protection/ mission effectiveness characteristics of a system.</p> <p>Analyses could include:</p> <ul style="list-style-type: none"> <li>• Shotline Analysis</li> <li>• Ballistic Impact Analysis</li> <li>• Fragmentation Impact Analysis</li> <li>• Shape Charge Impact Analysis</li> <li>• Sympathetic Detonation Analysis</li> <li>• Mine Blast Analysis</li> </ul> <p>The ARL reviews combat systems' vulnerabilities. A system that fails such a review may not be allowed to be fielded.</p>
AIDE Developer	Developer of the AIDE product. Analyzes capabilities of, architects, designs, integrates, verifies, and deploys the AIDE.
ARRoW Power User	An ARRoW user who has advanced knowledge and skills related to use and configuration of the ARRoW toolset/environment.
ARRoW Tool Developer	An ARRoW Developer who specifically develops a tool that integrates into the AIDE.
ARRoW User	An ARRoW User is a general role for any human that uses the AIDE.
Battle Lab Representative	A member of a battle lab group who might compose a model of a real or notional system for evaluation in an operational scenario. The model might be constructive, virtual, or live.
Circuit Designer	An engineer who designs electrical circuits.
CML Curator	This role is charged with maintaining the integrity of the CML.
Combat Arms User	<p>End user that includes representatives from:</p> <ul style="list-style-type: none"> <li>• Infantry</li> <li>• Armor</li> <li>• Field Artillery</li> <li>• Air Defense Artillery</li> <li>• Aviation</li> <li>• Special Forces</li> <li>• Corps of Engineers</li> </ul>
Combat Service Support User	<p>End user that includes representatives from:</p> <ul style="list-style-type: none"> <li>• Adjutant General Corps</li> <li>• Finance Corps</li> <li>• Transportation Corps</li> <li>• Ordnance Corps</li> <li>• Quartermaster Corps</li> </ul>
Combat Support User	<p>End user that includes representatives from:</p> <ul style="list-style-type: none"> <li>• Signal Corps</li> <li>• Military Police Corps</li> <li>• Military Intelligence Corps</li> <li>• Civil Affairs</li> <li>• Chemical Corps</li> </ul>
Command & Control Expert	Expert who understands battle command software, command and control messaging, and in general any interface to external command and control

Actor Name	Description
	centers of operation.
Communications Engineer	Engineer who designs and analyzes wireless communications equipment and systems.
Component Developer	Develops components for inclusion in the CML. Adds/modifies components to Component Model Libraries pursuant to library curator policies/procedures.  Components may or may not be initially developed within the AIDE.
Computer Architect	Develops architectures for computer processing systems including networked processing systems.
Context Model Developer	Creates models of external environments and systems that might interface with the system of interest.
Controls Engineer	An engineer who develops control hardware, software and algorithms for control systems. Examples include electrical motor servo control, thermal management control, and hydraulic systems control.
Design Archetype Developer	A library content developer of any design archetype or design reference architecture.
Design Integration Engineer	A design integration engineer manages the physical integration, including assigned location, of all components within the Sol.
Domain Engineer	A general role for a classical engineering discipline such as a mechanical or electrical engineer.
Domain Tool Developer	Developer of a Domain Tool. Refer Domain Tools.
Domain Tools	Tools, such as ProE, that are used by domain specific engineers to design the system of interest or its components.
Electrical Power Engineer	An electrical engineer who architects and designs power generation, distribution, and control systems and components.
EMI / EMC Expert	EMI = Electromagnetic Interference EMC = Electromagnetic Compatibility An expert, generally with an electrical engineering background, who understands the effects of electromagnetic coupling between components and systems and who is able apply design principles to minimize adverse effects of such phenomena.
FCC Representative	FCC = Federal Communications Commission  Expert who would validate that intentional radiated emissions are permissible within the USA.
Full Spectrum Simulator	A Full Spectrum Simulator is BAE Systems' concept of an external wargaming environment that could request the AIDE to construct a virtual prototype of a specified Sol and/or serve up its simulation, for example, via an HLA federation.  Full spectrum operations range from stable peace operations to major combat operations. Full Spectrum Operations includes variant sets of tasks required to conduct offensive, defensive, stability, and civil support operations.

Actor Name	Description
Government Needs Website	<p>This is a notional website that the government acquisition community would use to publish any form of request for contributions from the crowd. Such a website might publish challenge problems, requests from the cloud for design concepts, formal Requests for Proposals, detailed System Requirements Specifications, etc. The form of any such request might be standardized such that the AIDE can link a master model to a government needs interest or request.</p> <p>Such a website might be used by unsophisticated individuals as well as major defense contractors.</p>
Human Factors Engineer	<p>An engineer who architects, designs, and analyzes the ergonomics of systems and components as well as the predicted performance of humans using the system.</p>
HVAC Expert	<p>HVAC = Heating, Ventilation, and Air Conditioning</p> <p>An HVAC expert additionally designs Chemical, Biological, and Nuclear particle filtration and overpressure systems.</p>
IAT&C Engineer	<p>IAT&amp;C = Integration, Assembly, Test, and Checkout</p>
iFab	<p>The output of the ARROW toolset is a "blueprint" which is then sent to an iFab tool for virtual manufacturing.</p>
Information Assurance Engineer	<p>An expert in the field of cyber security.</p>
Innovation-Driven SOI Developer	<p>A member who executes clockwise thinking, i.e., creates ideas without being constrained by knowledge, explores the ideas, challenges the ideas, and discovers problems that can be solved with the ideas.</p> <p>Could be a Grad Student, for example. This person may have a technical background, but might also be an artist or a science fiction writer.</p>
ISR Engineer	<p>ISR = Intelligence, Surveillance, Reconnaissance</p> <p>An ISR Expert specifies and/or designs sensors, cameras, and sensor data processing and data distribution equipment.</p>
Knowledge-Driven SOI Developer	<p>A member who executes counterclockwise thinking, i.e., uses assumptions facts, and beliefs to establish knowledge, develops solutions based on knowledge, validates solutions to the knowledge, and applies solutions to the problems at hand.</p>
Library Content Developer	<p>A library content developer works on products that are meant to be applied across a multiplicity of future potential systems of interest, whereas a Sol developer is concentrating on solutions that are targeted to a specific design solution.</p>
Library Curator	<p>A library curator administers and manages the integrity of the library that they are responsible for. A curator controls the quality of the entities within the library, the certification of the trustworthiness and integrity of the collection content, ensures library services are maintained, are functional, and perform adequately, manages configuration of the library, and controls the security of and access to the library.</p>
Library Developer	<p>A Library Developer develops the infrastructure and services that are</p>

Actor Name	Description
	associated with the library.
Logistics Engineer	A Logistics Engineer is a Sol Development Subject Matter Expert (SME) focused on the scientific organization of the purchase, transport, storage, distribution, and warehousing of materials and finished goods of the system of interest.
Maintainability Engineer	<p>A Maintainability Engineer is a Sol Development SME focused on the ease in which a product can be maintained (preventative and corrective maintenance) in order to minimize the downtime of the System of interest (Sol). Thus increasing the operational availability (Ao) of a Sol.</p> <p>Maintainability— The ease with which a Sol to be retained in, or restored to, a specified condition when maintenance is performed by personnel having specified skills using prescribed procedures and resources at each prescribed level of maintenance and repair.</p> <p>Maintainability engineering aides in maximizing Sol uptime and operational availability (Ao) by providing design influence and analysis in the following cardinal product life cycle considerations:</p> <ul style="list-style-type: none"> <li>• isolate defects or their cause</li> <li>• correct defects or their cause</li> <li>• meet new requirements</li> <li>• make future maintenance easier</li> <li>• cope with a changed environment</li> </ul> <p>In some cases, maintainability involves a system of continuous improvement - learning from the past in order to improve the ability to maintain systems, or improve reliability of systems based on maintenance experience.</p> <p>The maintainability engineering effort in the conception and design phase is critical to ensure that high system availability is obtained at optimum Life Cycle Support Cost.</p> <p>Maintainability engineering effectively influences the System of Interest's availability calculation by minimizing downtime: the time required to bring a failed system back to its operational state or capability. This down time is normally attributed to maintenance activities. This minimized downtime does not happen at random, it is made to happen by actively ensuring that full consideration is given during the conceptual and design phase. Therefore the inherent maintainability characteristics of a system must be assured. This can be achieved by the implementation of specific design practices and validated through a maintainability assessment process, utilizing both analyses and testing. The following are cardinal maintainability engineering assurance activities:</p> <ul style="list-style-type: none"> <li>• Maintainability Programs</li> <li>• Maintainability Assessment</li> <li>• Maintainability Modeling</li> <li>• Maintainability Demonstration</li> <li>• Design for Maintainability</li> <li>• Defect Reporting and Corrective Action System (DRACAS)</li> </ul>
Malicious User	This is any user of the AIDE that has malicious intent. Such a user may

Actor Name	Description
	<p>be an amateur or a member of a sophisticated organization, including foreign governments.</p> <p>A malicious user might create content that has intentional side effects that may be difficult to detect but could make its way into deployed systems.</p> <p>Additionally, a malicious user might disrupt ARRoW services such as by employing denial of service. Such an attack could be in the form of overloading web servers or by provoking non-useful simulations that consume server processing bandwidth.</p>
Mechanical Engineer	<p>Mechanical engineering is a discipline of engineering that applies the principles of physics and materials science for analysis, design, manufacturing, and maintenance of mechanical systems. It is the branch of engineering that involves the production and usage of heat and mechanical power for the design, production, and operation of machines and tools.</p>
Mobility Expert	<p>A mechanical engineer who specializes in vehicle power train, steering, and/or suspension systems.</p>
Modeling & Simulation Engineer	<p>A software engineer who specializes in the design of modeling systems used for simulating the behavior of physical or cyber designs in the context of various external environments.</p>
NATO Representative	<p>Any NATO representative that would be interested in the Sol design, capability, or its modeled behavior.</p>
Navigation Systems Expert	<p>An engineer who specializes in the capability of and integration of navigation equipment such as Global Positioning System (GPS) and inertial navigation systems.</p>
Networking Architect	<p>An electrical or software engineer who specializes in the design and analysis of local area or wide area data networks.</p>
Operating Environment Expert	<p>A software engineer who specializes in the design and integration of operating systems, hypervisors, middleware, etc.</p>
Operational Commander	<p>Actual strategic user of the system. Stakeholder based on operational capabilities and Deep Green level use cases.</p>
OT&E Representative	<p>OT&amp;E = Operational Test &amp; Evaluation</p>
PMO Member	<p>PMO = Program Management Office</p> <p>This member can be of any Program Management Office that supports the acquisition of materiel for the U.S. Government.</p>
Process Monitor	<p>Quality assurance role, making sure the vehicle is designed and built correctly with respect to DoD processes and guidelines.</p>
Producibility Engineer	<p>An engineer who can influence the design to ensure it can be manufactured most easily and cost effectively.</p>
Program Manager	<p>The lead manager of the government acquisition organization directly responsible for the proper execution of the program (cost, schedule, and performance) that develops the system of interest.</p>
Program Office	<p>The government acquisition organization that is ultimately responsible for</p>

Actor Name	Description
	the proper execution of the program that develops the system of interest. A program office might manage multiple programs, each with its own Program Manager.
R&D Center Representative	<p>R&amp;D = Research &amp; Development</p> <p>Government funded R&amp;D centers that undertake creative work on a systematic basis in order to maintain and/or increase the stock of knowledge in the interest of the United States of America, or for the betterment of mankind, culture and society.</p> <p>Research and development is often scientific and focused towards developing particular technologies, and devising new applications based on the stock of knowledge and technologies.</p>
Regulatory Authority	This includes any agency or organization representative that may need to review and/or analyze and approve the Sol design and the context in which the design may be used and is fielded.
Reliability Engineer	<p>A Reliability Engineer is a Sol Development SME focused on the study, evaluation, and life-cycle management of reliability of the System of Interest (Sol): the ability of a system or component to perform its required functions under stated conditions for a specified period of time.</p> <p>Basic Reliability - The duration or probability that a Sol will perform satisfactorily (failure-free performance) for a given time when used under specified operating conditions. As a general definition, reliability is the capacity of parts, components, equipment, products and systems to perform their required functions for desired periods of time without failure, in specified environments and with a desired confidence. There are two specialized types of reliability: Logistics Reliability and Mission Reliability.</p> <p>Logistics Reliability – The ability of a Sol to perform failure free, under specified operating conditions and time without demand on the support system, measured as a mean time between maintenance actions. Logistics reliability is a measure of a system's ability to operate without logistics support. All failures, whether the mission is or can be completed, are counted.</p> <p>Mission Reliability – The probability that the Sol is operable and capable of performing its required function for a stated mission duration or for a specified time into a mission.</p> <p>Reliability engineering and maintainability engineering are inter-dependent.</p> <p>Reliability engineering discipline concerned with predicting, monitoring, testing, and improving the reliability of a system, device, or process.</p> <p>Reliability engineering for complex systems require a different more elaborated systems approach than reliability for simple systems/parts. Reliability engineering is closely related to system Safety engineering in the sense that they both use common methods for their analysis and require input from each other. Reliability engineers should have broad skills and knowledge. Reliability engineering has important links with</p>

Actor Name	Description
	Functional design, Hardware Design, Software, Manufacturing, Transport, Handling, Storage, Spare parts, Operational issues, Human Operators and maintainers, Repair shops, Software, Manuals, Training and more.
Requirement Archetype Developer	Develops requirement archetypes for inclusion in the CML or other libraries.
Safety / Environmental Engineer	An engineer who analyzes and influences designs in terms of safety and environmental impact.
Safety Board Representative	Example: Weapon Systems Explosive Safety Review Board.
Software Architect	<p>The main responsibilities of a software architect include:</p> <ul style="list-style-type: none"> <li>• Limiting choices available during development by: <ul style="list-style-type: none"> <li>– choosing a standard way of pursuing application development</li> <li>– creating, defining, or choosing an application framework for the application</li> </ul> </li> <li>• Recognizing potential reuse in the organization or in the application by: <ul style="list-style-type: none"> <li>– Observing and understanding the broader system environment</li> <li>– Creating the component design</li> <li>– Having knowledge of other applications in the organization</li> </ul> </li> </ul> <p>Software architects can also:</p> <ul style="list-style-type: none"> <li>• Subdivide a complex application, during the design phase, into smaller, more manageable pieces</li> <li>• Grasp the functions of each component within the application</li> <li>• Understand the interactions and dependencies among components</li> <li>• Communicate these concepts to developers</li> </ul> <p>In order to perform these responsibilities effectively, software architects often use Unified Modeling Language and OOP. UML has become an important tool for software architects to use in communicating the overall system design to developers and other team members, comparable to the drawings made by building architects.</p>
Software Engineer	Software Engineering (SE) is a profession dedicated to designing, implementing, and modifying software so that it is of high quality, affordable, maintainable, and fast to build. It is a systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software.
Sol Analyst	A System of Interest (Sol) developer who uses the AIDE to analyze the design or performance of the Sol.
Sol Acquisition Community Member	General role for any member of the government acquisition community.
Sol Developer	<p>Sol = System of Interest</p> <p>A library content developer works on products that are meant to be applied across a multiplicity of future potential systems of interest, whereas a Sol developer is concentrating on solutions that are targeted to a specific design solution.</p>

Actor Name	Description
Sol Developer - Designer	A System of Interest (Sol) developer who uses the AIDE design a new Sol or any of its components.
Sol Developer - Reference Architecture	A System of Interest (Sol) developer who uses the AIDE design a new design archetype or reference architecture.
Sol Developer - Test Case	Coordinates with Requirements developer to create ARRoW-compatible test cases and test case archetypes.
Sol Developer - Test Result Analyzer	A System of Interest (Sol) developer who uses the AIDE design a new test result analyzer.
Sol Developer - Verifier	This is the person who reviews the test results reported by ARRoW for the purpose of disposition.
Sol Developer - Requirements	Performs requirements analysis, writes requirements and requirements archetypes, derives/decomposes and allocates requirements and requirements archetypes. Assigns test method to each requirement created.
Sol Development SME	<p>SME = Subject Matter Expert</p> <p>A Sol SME is someone who has some specialized expertise in a field, the application of which could potentially influence the design of the Sol.</p>
Sol End User	Actual strategic user of the fielded system of interest, including any personnel involved in its needed support services.
Specialty Engineer	Specialty Engineering is a general class of engineering disciplines that include Reliability, Maintainability, Logistics, Human Factors, Testability, Producibility, Safety, and Environmental engineering.
Survivability Expert	A Survivability Expert can analyze and design integrated solutions of armor, spall liners, component placement, and hull design to optimize force protection/mission effectiveness characteristics of a system.
SW Integration Expert	A software engineer who specializes in the integration and test of software systems.
Systems Engineer	<p>A SE is a specialized domain engineer who applies an engineering discipline to a Sol that concentrates on the design and application of the whole (system) as distinct from the parts. The SE looks at a problem in its entirety, taking into account all the facets and all the variables and relating the social to the technical aspect. A SE integrates multiple disciplines and specialty groups into a set of activities that proceed from concept to production to operation.</p> <p>A SE applies an interdisciplinary approach and means to enable the realization of a successful Sol. The SE focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding with design synthesis and system validation while considering the complete problem: operations, cost and schedule, performance, training and support, test, manufacturing, and disposal. The SE also considers both the business and the technical needs of all customers with the goal of providing a quality Sol that meets the user needs.</p>
Test Case Archetype Developer	Develops test case archetypes for inclusion in the CML or other libraries.

Actor Name	Description
Test Range/Facility Representative	<p>Test ranges and test facilities have severe constraints on Sol design and operation limits. Test articles sent to these facilities may need special configurations to conform to these constraints and may need to be independently verified that they conform to these constraints before the article can be used on the range/facility.</p> <p>Notes:</p> <ul style="list-style-type: none"> <li>- A Safety Assessment Report (SAR) is generally required to be delivered to this representative.</li> <li>- A Safety Fan Declaration is typically provided for ballistic or flyable rounds that declares the worst case maximum flight envelop through which the round will pass under any normal or failure mode condition.</li> </ul>
Testability Engineer	<p>A Testability Engineer is a Sol Development SME focused</p> <p>Testability - A design characteristic which allows the status (operable, inoperable, or degraded) of an item to be determined and the isolation of faults within the item to be performed in a timely manner (MIL-STD-2165). A design characteristic that allows its operational status to be determined and the isolation of faults to be performed efficiently (IEEE Std 1522).</p> <p>Testability Analysis – The engineering practice associated with evaluating the testability of a system, device, or process.</p>
TRA Archetype Developer	Develops test result analyzer archetypes for inclusion in the CML or other libraries.
TRADOC Member	TRADOC = Training and Doctrine Command
Training and Doctrine Developer	A Training and Doctrine Developer creates Tactics, Techniques, and Procedures (TTPs) that include use of materiel solutions, including the Sol.
Training Engineer	An engineer who specializes in the content development of embedded and non-embedded training materials.
TSM Representative (1 of 4)	<p>TSM = TRADOC System Management</p> <p>The CG, TRADOC will establish a TSM office to provide intensive management beyond the scope of normal management resources available to the proponent for:</p> <p>(1) A materiel system, a family of materiel, or a group of closely related/interdependent materiel systems that are being developed.</p> <p>(2) Non-system training devices or training systems.</p> <p>b. TRADOC System Managers will normally be considered for establishment between Milestones A and B, at the end of Materiel Solution Analysis, or when a concept is approved. Programs must meet the following criteria for establishment of a TSM:</p> <p>(1) Program must be an ACAT I, ACAT II, or other high-priority materiel system as determined by CG, TRADOC.</p>

Actor Name	Description
	<p>(2) Must be a program manager/program executive officer managed program.</p> <p>(3) Workload must be such that the program cannot be managed within the resources and structure available to the proponent.</p> <p>(4) Workload or uniqueness of the program must be such that an existing TSM cannot assume the program. Intent of this regulation is not to preclude combining of individual system responsibilities in one TSM.</p>
TSM Representative (2 of 4)	<p>(5) Program must be higher priority or have greater need for a TSM than existing TSM managed programs. Charter revisions through DCSCD whenever they perceive that a need exists. TSM duties and responsibilities. TRADOC System Managers will:</p> <ul style="list-style-type: none"> <li>a. Serve as the TRADOC user representative and single Point of Contact for systems assigned in accordance with the TSM charter.</li> <li>b. Provide intensive, centralized, total system management and integration of all DTLOMS considerations.</li> </ul> <p>(1) Doctrine. Coordinate the development of doctrine and tactics, techniques and procedures from individual to collective, tracing back to the operational and organizational concept.</p> <p>(2) Training. Coordinate development of home station and institutional training for individual, crew and unit. Coordinate development and fielding of training aids, devices (system and non-system), simulations and simulators for use in training in the institution, home station, and Combat Training Centers.</p> <p>(3) Leader Development. Coordinate development of leader (NCO and Officer) training and development.</p> <p>(4) Organization. Coordinate development of basis of issue plans for assigned systems and associated ancillary equipment, including all aspects of logistical support. Coordinate development of force design updates and Tables of Organization and Equipment (TOEs) related to assigned systems.</p> <p>(5) Materiel. Coordinate TRADOC position on system reviews, ensure requirement documents are updated as needed, ensure DTLOMS and the logistics support system are in place for system testing and first unit equipped, and plan for system product improvements and recapitalization.</p> <p>(6) Soldier. Identify and reconcile all Manpower and Personnel Integration (MANPRINT) issues, including safety. Coordinate development of new military occupational specialty (MOS) and appropriate career progression as needed.</p> <p>c. Monitor and synchronize all aspects of total system development,</p>

Actor Name	Description
	testing and evaluation, corrective actions, acquisition, materiel release, and fielding, to include direct interaction with the program/project/product managers (PMs) and materiel developers (MATDEVs) of the primary and ancillary system(s), test community, and the fielding/gaining commands.
TSM Representative (3 of 4)	<p>d. Using an Integrated Concept Team (ICT) with empowered membership from schools and MATDEVs, coordinate the development and documentation of all related materials, as needed:</p> <p>e. In coordination with the proponent Directorate of Combat Developments propose refinement of system requirements in the ORD. Justify or validate system requirements at all levels of the Army, Department of Defense (DoD), and Congress, as directed.</p> <p>f. Participate in MATDEV system concept analyses and cost performance trade-off and cost as an independent variable analyses by providing detailed warfighting capability impact of specific system characteristics. Provide TRADOC senior leadership recommendation for all design reviews.</p> <p>g. Prepare TRADOC position on, receive TRADOC leadership approval, and participate in decision reviews (In Progress Review (IPR)/Army Systems Acquisition Review Council/Army Requirements Oversight Council (AROC)/Joint Requirements Oversight Council (JROC)/Defense Acquisition Board) for assigned systems. Provide user input for documentation of these reviews, such as Acquisition Program Baseline. Act as user representative on any other acquisition reviews/boards for assigned systems.</p> <p>h. As a part of unit set fielding, support total package fielding by managing a coordinated schedule of work for TRADOC schools and activities in support of system development and initial fielding.</p> <p>i. Identify and prioritize system hardware and software deficiencies to the MATDEV for corrective action. Review and evaluate proposed actions and engineering change proposals of the project or program manager to ensure that user requirements are adequately addressed.</p> <p>j. Provide for system improvements (Preplanned Product Improvements, System Enhancement Program, Service Life Extension Program, recapitalization efforts, etc.) in coordination with the proponent. This is accomplished through the identification of Science and Technology, Science and Technology Objectives, Advanced Technology Demonstrations, Advanced Concept Technology Demonstrations, and Concept Experimentation Programs for systems assigned to the TSM.</p> <p>k. Ensure test units are trained and prepared for testing. Coordinate all user involvement in system testing (for example, scenario development, test support, unit training, and user subject matter expertise). Monitor technical and user test activities for assigned systems to keep TRADOC leadership informed of system progress and to initiate corrective action for user unit or test personnel/activities as needed.</p>
TSM Representative (4 of 4)	l. Crosswalk and reconcile O & O concept to ORD characteristics to the

Actor Name	Description
	<p>Request for Proposal (RFP) materiel specifications, ensuring the acquisition strategy meets user needs.</p> <p>m. Articulate system operational and organizational concepts associated with their system as a member of combined arms system of systems and joint environments.</p> <p>n. Provide user coordination to manpower estimates.</p> <p>o. Provide use representation in analysis of alternatives (AoAs), and other studies, evaluations, and efforts supporting the development programs.</p> <p>p. Provide TRADOC representation to allied/prospective users of the assigned systems.</p>
Use Case Creator	Creates the operational use cases and test cases that derive from them. Maintains and controls access to the tests.
User Interface Expert	An expert in the design and architecting of software and hardware interfaces with the user, including Graphical User Interfaces (GUIs).
War Fighter	Front line user of the system. Stakeholder concerns are about how it operates.
Weapons Expert	A subject matter expert in the field of combat systems armament.

#### 7.1.1.4.2 ARRoW Requirements

During the development of the ARRoW use cases and the ARRoW architecture, a few behavioral requirements emerged that were most expediently captured as text requirements. Although the list of these textual requirements is quite brief, it is provided in Table 7.1-10 for the purpose of completeness of this report. These requirements can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.1.3 Requirements”.

**Table 7.1-10. ARRoW Requirements**

ID	Name	Requirement Text
3	Requirements	{Header}
3.1	Functional Requirements	{Header}
3.1.1	Sol Test Requirements	{Header}
3.1.1.1	Estimate Execution Duration	ARRoW shall provide an estimate of Test Case execution duration prior to initiation of the test. This duration may be normalized to a benchmark standard.
3.1.1.2	Configure the Test Environment for the UUT	ARRoW shall configure the test environment for design-under-test. Rationale: The test environment includes all of the test support environment assets required to execute the test.

ID	Name	Requirement Text
3.1.1.3	Record Test Result	ARRoW shall record the test results.
3.1.1.4	Check Missing Rqmt Dependency	ARRoW shall check that each requirement dependency in a requirement archetype is mapped to an actual SoI requirement.
3.1.1.5	Record Test Failure Cause	ARRoW shall record the cause of failure when a test fails.
3.1.1.6	Test Failure Guidance	ARRoW shall compare historical test failure causes to the current design and shall notify the user if design discrepancies exist in this regard.
3.1.1.7	Recommend Order of Test Case Execution	ARRoW shall recommend the execution order when multiple test cases are run in a shared simulation.
3.1.1.8	Utility Curves	ARRoW shall support utility curve association with requirements and verification.  Rationale: as in Quality Functional Deployment, there can be degrees of requirement compliance other than just Pass/Fail. This requirement is intended to support this capability.
3.1.1.9	Report Test Result	ARRoW shall report test results such that both required results and actual results are depicted.
3.1.1.10	Configure Unit For test	ARRoW shall configure the design-under-test for testing.
3.1.2	Sol Design Requirements	{Header}
3.2	ARRoW Interface Requirements	{Header}
3.3	ARRoW Design Constraints	{Header}

#### 7.1.1.4.3 ARRoW Use Cases

The main thrust of the AIDE use case analysis for this phase of the META project was to explore how requirements imposed on a System-of-Interest (SoI) will flow to test cases that in turn can be executed to verify that the SoI satisfies those requirements. This is sometimes referred to as the “Requirements to Test Case (RTTC) flow problem”.

The SysML use case artifacts described in this section can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.1.4 Use Cases”.

Figure 7.1-2 provides a top level overview of the relationships between the generalized ARRoW actors, the top level use cases that they interact with, and the SysML packages in which these use cases are contained.

The initial set of top level use case categories include:

- Acquiring System of Interest (SOI) Designs

- Developing & Delivering SOI Designs
- Developing & Supporting the AIDE
- Protecting Against Malicious Acts

The “Acquiring SOI Designs” use case category includes lower level use cases for SOI capability analysis and capability gap analysis, and enabling interaction with government needs. The “Developing & Delivering SOI” use case category involves use cases to assist the developer of the SOI in requirements analysis, design, design analysis, verification-testing, and transition of design to production. The “Developing & Supporting the AIDE” use case category includes the generation of library content, curating the library, and developing the AIDE. The “Protect Against Malicious” use case category includes active and passive measures to protect the AIDE, and its data and users.

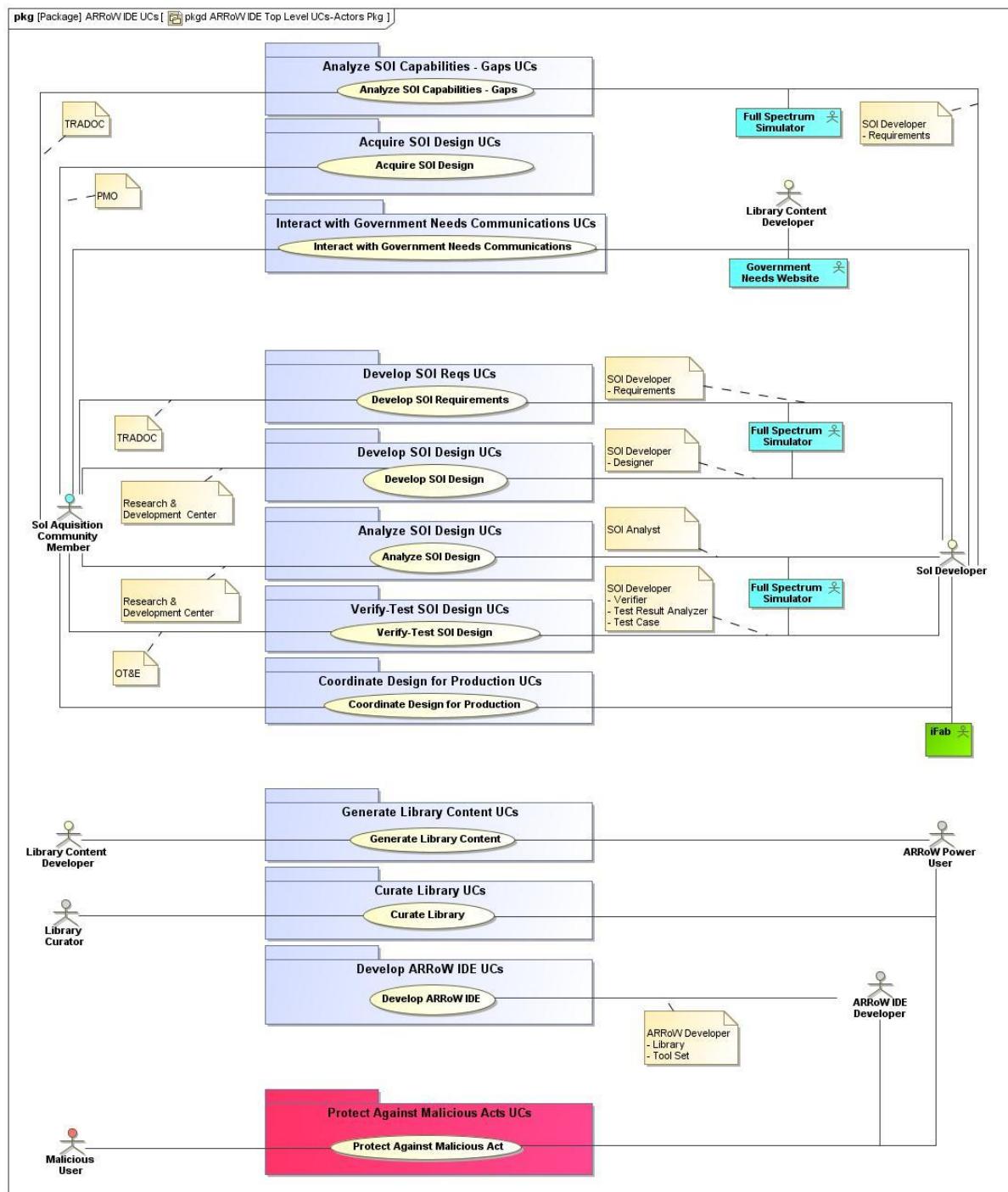
**Figure 7.1-2. AIDE Top Level UCs-Actors**

Figure 7.1-3 is a SysML diagram that shows the beginning organization of a use case analysis to explore the RTTC flow problem. The use cases associated with this AIDE functionality are contained in the package labeled “Rqmt & TC UCs”. As a ‘housekeeping’ technique, a “ToDo” package is nested within the “Rqmt & TC UCs” package, and contains those use cases that have

been identified but have not yet had internal textual content created for them. The use case labeled “Requirement to Test Case Flow” was used as the driving use case for the development of the ARRoW architecture supporting the ARRoW RTTC flow process. The text for this use case is provided in Figure 7.1-4.

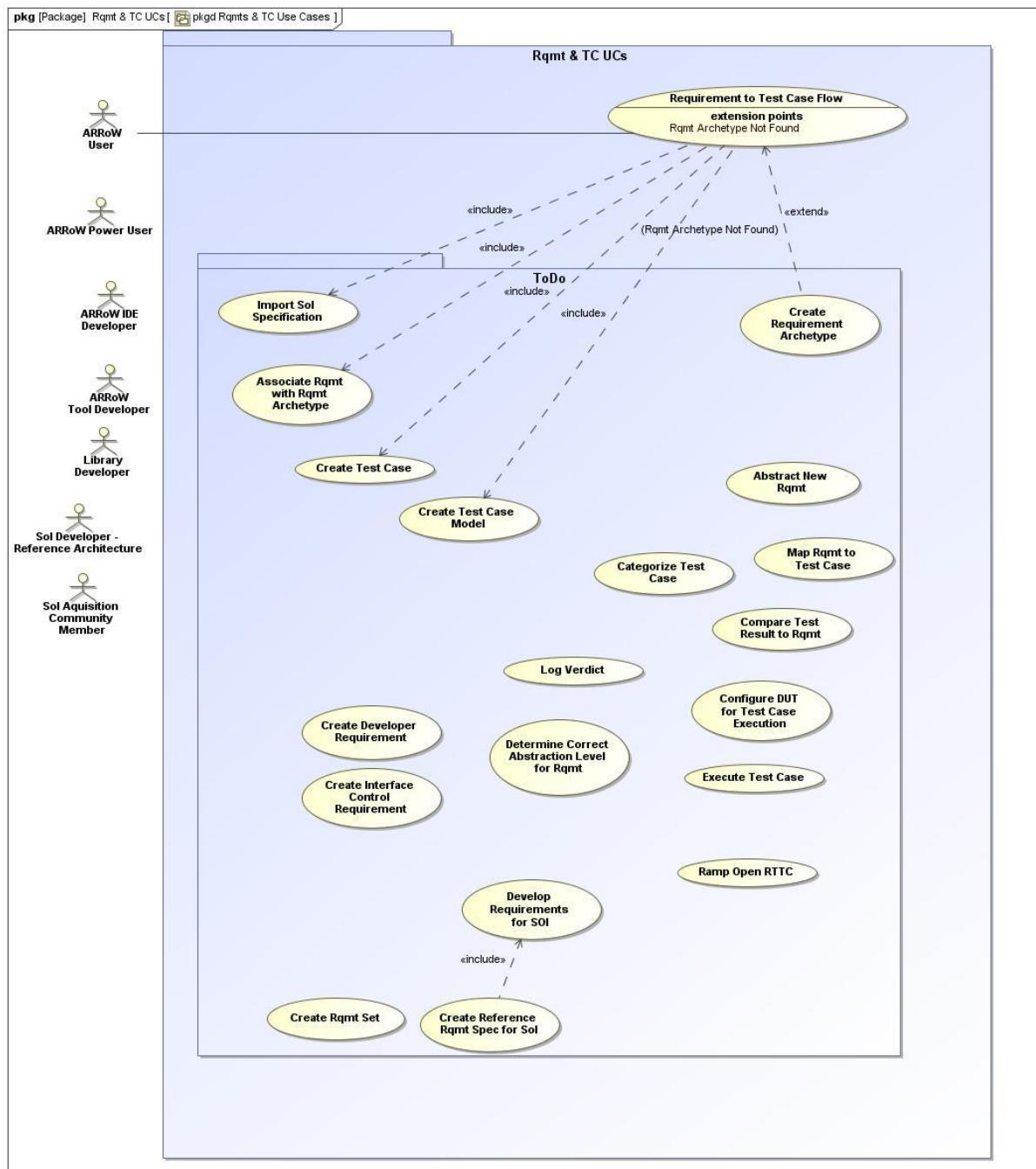


Figure 7.1-3. Rqmts & TC Use Cases

Requirements to Test Case Flow Use Case

Author  
John Bangs, Steve Schmitt

Date  
7/12/2011

Description  
This use case explores the steps involved in creating a test case that can verify a given SoI requirement.

Preconditions  
All levels of requirement specifications related to the SoI have been created.

Stimuli

Main Flow

1. Import the set of rqmts (a specification) into ARRoW. This entails recording the text of the rqmts into the ARRoW environment.
2. For each requirement:
  - 2.1. Peruse a library of existing rqmt archetypes (RAs) and find an applicable RA.
  - 2.2. Create a copy of the RA and place it in the master model.
  - 2.3. Associate the SoI rqmt with the RA (e.g., using a Wizard). TBR: this step may also allow for modification of the RA's template expression
  - 2.4. Modify and/or add to the default RA-to-design entity allocation relationships (e.g., using a Wizard).
  - 2.5. Optionally add and define a utility function.
  - 2.6. For each RA referenced in the Requirement Archetype Dependency List:
    - 2.6.1 Execute steps 2.2 through 2.6.1 using the appropriate constraint requirements in the SoI specs. TBR:  
Probably need to account for multiple grades of utility function sensitivity that flows from requirement to specific test case.
3. For each RA:
  - 3.1. Create a copy of its associated Test Case Archetype (TCA) and place it in the master model.
4. For each TCA:
  - 4.1. Modify the TCA to make it an executable Test Case (TC) (e.g., using a Wizard). Note: since step 2.4 previously allocated the RA to the design entity, and since this TC's parent TCA has a one-to-one association with that RA, then this TC is allocated to the design entity.
  - 4.2. Create a Test Result Analyzer (TRA) that compares the test results to the associated SoI requirement. Note: could be built automatically in some cases.
5. For each TC:
  - 5.1. Create a component model (CM).
  - 5.2. Associate the CM and TC into a TC-CM Pair.
  - 5.3. Associate the TC-CM Pair with the appropriate DUT.

Postconditions  
A test case and component model have been created and associated with a DUT. The TC and CM are derived from the associated SoI requirement.

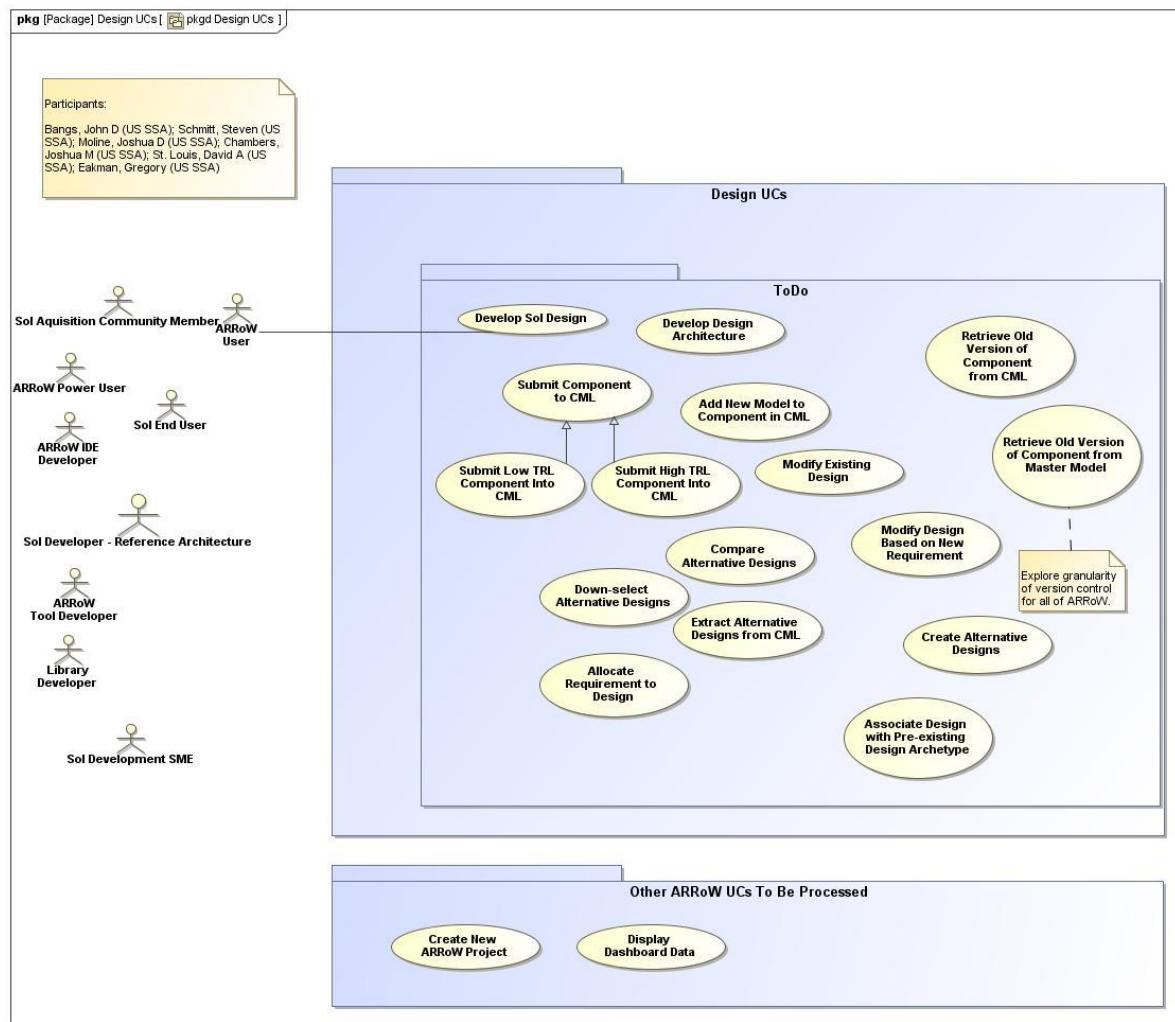
---

Alternate Flow  
<Rqmt Archetype Not Found>

- 2.1. Requirement Archetype was not found, so execute Create Requirement Archetype UC

**Figure 7.1-4. RTTC Flow Use Case**

In addition to the RTTC flow analysis, an initial analysis was performed to identify use cases that could apply to ARRoW functionality not directly related to the RTTC flow problem. Inputs were solicited from subject matter experts within the ARRoW tool design community. Figure 7.1-5 shows the use cases and actors identified in this analysis.



**Figure 7.1-5. Design UCs**

#### 7.1.1.5 Example Development Metrics

Program personnel typically use a variety of metrics charts (e.g., scorecard, line, scatter plot, control, bar, pie, histograms, area, bubble, radar, etc.) to execute a combat vehicle development project and transition into a combat vehicle production project. Figure 7.1-6 illustrates an example project metrics scorecard. Project metrics scorecards or dashboards communicate overall project status on a wide range of indicators, and aide in information briefings, and managing by exception. Figure 7.1-7 illustrates an example individual metric template chart. Individual metrics charts communicate details on plans, progress, trends, impacts, and aide in developing a thorough understanding and decision making. Development metrics can also

serve double duty or be leveraged as aids to project formal decisions, trade studies, and pivotal phase transitions such as production and deployment of combat vehicles.

### Example Project Metrics Scorecard

Project leaders configure project scorecards to provide roll ups or summaries of lower level metrics charts to meet the needs of the project forum and/or communication plan. Figure 7.1-6 illustrates an example of a project metrics scorecard for a notional IFV covering a range of typical program and project metrics for:

- Cost and system effectiveness
- Joint Capabilities Integration & Development System (JCIDS) capabilities-based assessments and gap analysis
- Design maturity and health
- Problem domain understanding
- Project health

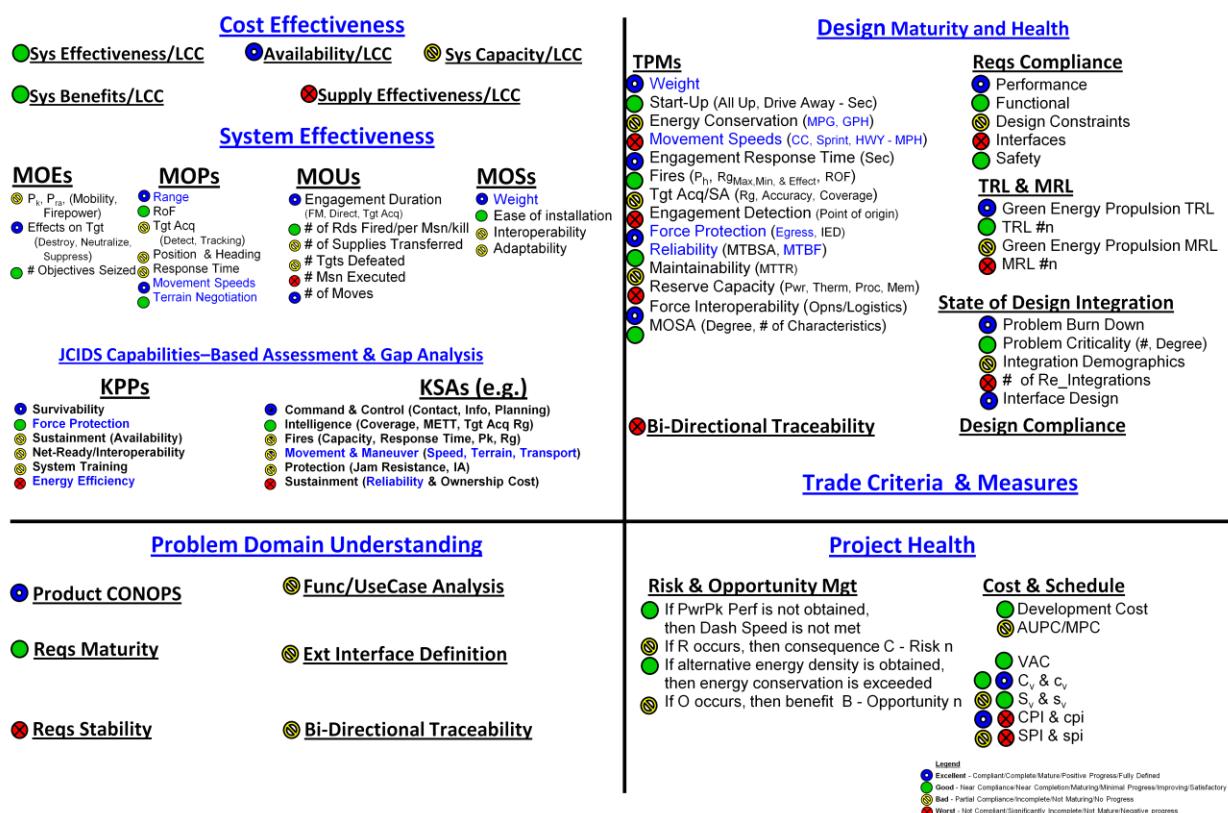


Figure 7.1-6. Example Project Scorecard for a Notional IFV

### Example Individual Metric Template Chart

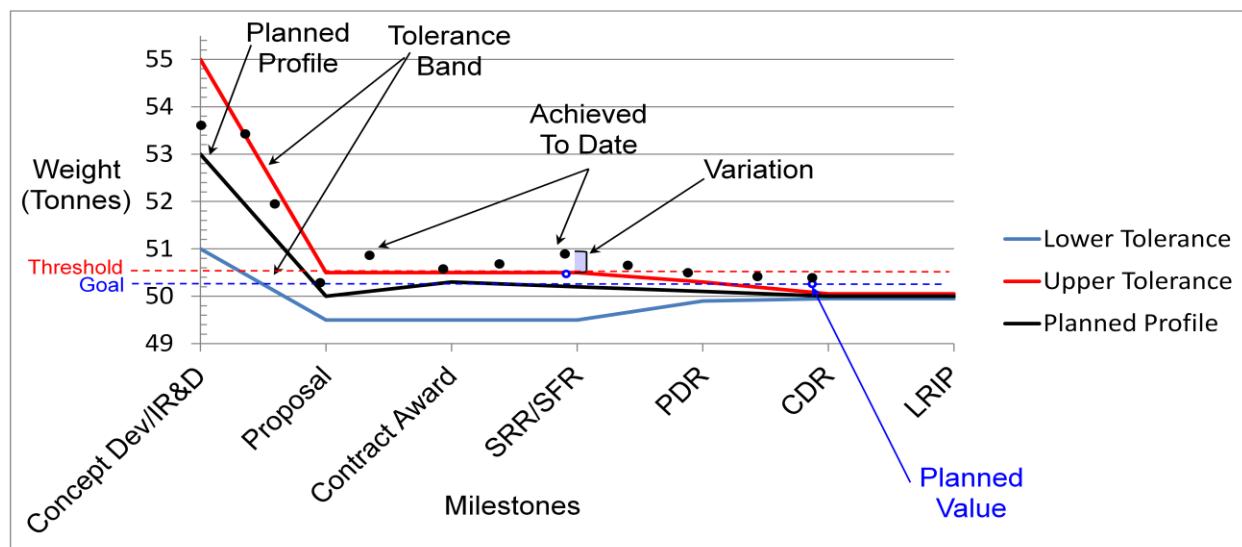
Figure 7.1-7 illustrates a typical individual metric line-control chart template used as a Systems Analysis & Control tool by program management to monitor a technical performance measurement (TPM) within project design maturity & health metrics. TPMs are created for the purpose of tracking design progress for key requirement and supporting management decision-making. A line-control metrics chart provides project leadership with the following design maturity & health knowledge enablers within the System Analysis & Control process:

- Performance tracking
- Planned performance, risk reduction, or opportunity exploitations
- Threshold triggers
- Trend indications

Individual metrics chart templates can be used to:

- Project probable performance over time
- Provide indications of design progress by recording actual performance observed
- Assist decision making by comparing actual versus projected performance
- Provide early warning of technical problems
- Support trend analysis and assessments as to whether operational requirements will be met
- Support impact analysis of proposed changes to system performance

This individual metrics chart template can be used for Design Maturity & Health Metrics (e.g., all TPMs – weight, physical dimensions, transportability, start-up times, energy conservation, movement speeds, reliability, etc.



**Figure 7.1-7. Example Individual Metric Template Chart**

## Project Formal Decision/Trade Study Criteria and Measures

Table 7.1-11 provides an example of common metrics that are used as measures for evaluation criteria in formal decisions or trade studies to:

- Balance requirements
- Assist requirements compliance path forward decisions
- Select technologies, design concepts, hardware and/or software alternatives
- Assist design development path forward decisions
- Assist integration, assembly, test, and checkout path forward decisions
- Assist verification and validation path forward decisions

**Table 7.1-11. Example Metrics for Trade Criteria and Measures**

Criterion	Description	Measure
Production Cost	Cost for recurring production of the alternative.	Average Unit Production Cost relative to Baseline Product
Development Cost	Cost to develop, design, test, implement, and certify the alternative (includes all program costs through LRIP).	Hardware Non-Recurring Engineering (HW NRE) cost
		Software Non-Recurring Engineering (SW NRE) cost
Inter-operability	The ability of the alternative to operate across new, existing, and foreign platforms.	Weapons Compatibility
		Munitions Compatibility
		Command & Control (C <sup>2</sup> ) Compatibility
		Communications Compatibility
		Logistics Compatibility—transportability, Supplies
Adaptability	The ability of the alternative to satisfy current and future operational needs. Includes commonality, scalability, modularity, and upgradeability/extensibility.	Component Commonality
		Future Capability Growth Potential
Survivability	The ability of the alternative to complete the mission under threat measures or countermeasures. Includes susceptibility, vulnerability, and recoverability.	Insensitive Munitions (IM) Characteristics
		Anti-Jam Capability (self-defense, communications)

<b>Criterion</b>	<b>Description</b>	<b>Measure</b>
Safety	The effect the alternative has on minimizing hazards or injury to operations and maintenance personnel. The criterion also includes safety to ordnance and equipment.	Basic Safety
		Equipment Safety
		Weapons & Munitions Safety
		Operational Safety
Reliability		Software Safety
	The effect the alternative has on the probability that the projectile does not fail to complete its mission under specified supply, handling, storage, and firing conditions.	Parts Count
Risk		Complexity
	The cost, schedule, and technical risk level of developing the alternative.	Technology Risk
		Schedule Risk
		Cost Risk
Capability	The capability of each alternative for providing the core functionality.	Lethality – Range, Rate of Fire (RoF) Accuracy
		Mobility – Speed, Acceleration
		C <sup>2</sup> – Planning, Execution
		Communications – Range, Throughput
		Survivability - Reaction

## Transition to Production & Deployment

The transition success from a combat vehicle development phase using the ARRoW Integrated Development Environment (IDE) into a combat vehicle production phase hinges on incorporating transition to production & deployment metrics. The AIDE would assist in conducting Manufacturing Readiness Assessments (MRAs) and use Manufacturing Readiness Levels (MRLs) to determine readiness for production.

“MRAs ensure mature manufacturing processes to meet:

- Cost commitments
- Product quality and consistency requirements
- On time delivery”<sup>[MRL10]</sup>

“MRL goals are to use:

- Mature technologies
- Stable designs
- Production processes in control”

The purpose of MRLs is to provide decision makers with a common understanding of the relative maturity and risks associated with manufacturing technologies, products, and processes being considered. The purpose and definitions of MRLs range from:

- Design readiness and producibility
- Manufacturing plans and schedules, processes, tools, training, skills, risks
- Supply chain, QA, material availability, long lead
- Demonstrated production (pilot lines, Low Rate Initial Production (LRIP), Full Rate Production [FRP])

Manufacturing readiness and producibility are as important to the successful development of a combat vehicle as those of readiness and capabilities of the technologies intended for the combat vehicle. Manufacturing risk identification and management begins at the earliest stages of technology development, and continues vigorously throughout each stage of a program’s development life-cycle. MRL levels 1 – 8 as listed in Table 7.1-12 span development phases from Materiel Solution Analysis (MSA) to Technology Development (TD) up through engineering & Manufacturing Development (EMD). <sup>[MRL10]</sup>

“MRL is a measure used to assess the maturity of manufacturing readiness serving the same purpose as Technology Readiness Levels serve for technology readiness. MRLs are designed to be measures used to assess the maturity of a given technology, component or system from a manufacturing prospective. The MRL intent was to create a measurement scale that would serve the same purpose for manufacturing readiness as technology readiness levels (TRLs) serve for technology readiness – to provide a common metric and vocabulary for assessing and discussing manufacturing maturity, risk and readiness. MRLs were designed with a numbering system to be roughly congruent with comparable levels of technology readiness levels (TRLs) for synergy and ease of understanding and use.” <sup>[MRL10]</sup>

“MRAs and MRLs answer production transition questions and reduce manufacturing risk unanswered by Technology Readiness Level (TRLs):

- Is the technology producible
- What will the design cost in production
- Can the design be made in a production environment

- Are key materials and components available” [MRL10]

**Table 7.1-12. MRL Definitions**

Manufacturing Readiness Level Definitions Error! Bookmark not defined. <sup>5</sup>			
MRL	Definition	Description	Phase
1	Basic Manufacturing Implications Identified	This is the lowest level of manufacturing readiness. Basic research expands scientific principles that may have manufacturing implications. The focus is on a high level assessment of manufacturing opportunities. The research is unfettered.	Pre Material Solution Analysis
2	Manufacturing Concepts Identified	Invention begins. Manufacturing science and/or concept described in application context. Identification of material and process approaches are limited to paper studies and analysis. Initial manufacturing feasibility and issues are emerging.	Pre Material Solution Analysis
3	Manufacturing Proof of Concept Developed	Conduct analytical or laboratory experiments to validate paper studies. Experimental hardware or processes have been created, but are not yet integrated or representative. Materials and/or processes have been characterized for manufacturability and availability but further evaluation and demonstration is required.	Pre Material Solution Analysis
4	Capability to produce the technology in a laboratory environment.	Required investments, such as manufacturing technology development identified. Processes to ensure manufacturability, producibility and quality are in place and are sufficient to produce technology demonstrators. Manufacturing risks identified for prototype build. Manufacturing cost drivers identified. Producibility assessments of design concepts have been completed. Key design performance parameters identified. Special needs identified for tooling, facilities, material handling and skills.	Material Solution Analysis (MSA) leading to a Milestone A decision.

<b>Manufacturing Readiness Level Definitions Error! Bookmark not defined.<sup>5</sup></b>			
<b>MRL</b>	<b>Definition</b>	<b>Description</b>	<b>Phase</b>
5	Capability to produce prototype components in a production relevant environment.	Mfg strategy refined and integrated with Risk Mgt Plan. Identification of enabling/critical technologies and components is complete. Prototype materials, tooling and test equipment, as well as personnel skills have been demonstrated on components in a production relevant environment, but many manufacturing processes and procedures are still in development. Manufacturing technology development efforts initiated or ongoing. Producibility assessments of key technologies and components ongoing. Cost model based upon detailed end-to-end value stream map.	Technology Development (TD) Phase.
6	Capability to produce a prototype system or subsystem in a production relevant environment.	Initial mfg approach developed. Majority of manufacturing processes have been defined and characterized, but there are still significant engineering/design changes. Preliminary design of critical components completed. Producibility assessments of key technologies complete. Prototype materials, tooling and test equipment, as well as personnel skills have been demonstrated on subsystems/ systems in a production relevant environment. Detailed cost analysis include design trades. Cost targets allocated. Producibility considerations shape system development plans. Long lead and key supply chain elements identified. Industrial Capabilities Assessment (ICA) for MS B completed.	Technology Development (TD) phase leading to a Milestone B decision.
7	Capability to produce systems, subsystems or components in a production representative environment.	Detailed design is underway. Material specifications are approved. Materials available to meet planned pilot line build schedule. Manufacturing processes and procedures demonstrated in a production representative environment. Detailed producibility trade studies and risk assessments underway. Cost models updated with detailed designs, rolled up to system level and tracked against targets. Unit cost reduction efforts underway. Supply chain and supplier QA assessed. Long lead procurement plans in place. Production tooling and test equipment design & development initiated.	Engineering & Manufacturing Development(EMD) leading to Post CDR Assessment

Manufacturing Readiness Level Definitions Error! Bookmark not defined. <sup>5</sup>			
MRL	Definition	Description	Phase
8	Pilot line capability demonstrated. Ready to begin low rate production.	Detailed system design essentially complete and sufficiently stable to enter low rate production. All materials are available to meet planned low rate production schedule. Manufacturing and quality processes and procedures proven in a pilot line environment, under control and ready for low rate production. Known producibility risks pose no significant risk for low rate production. Engineering cost model driven by detailed design and validated. Supply chain established and stable. ICA for MS C completed.	Engineering & Manufacturing Development (EMD) leading to a Milestone C decision.
9	Low Rate Production demonstrated. Capability in place to begin Full Rate Production.	Major system design features are stable and proven in test and evaluation. Materials are available to meet planned rate production schedules. Manufacturing processes and procedures are established and controlled to three-sigma or some other appropriate quality level to meet design key characteristic tolerances in a low rate production environment. Production risk monitoring ongoing. LRIP cost goals met, learning curve validated. Actual cost model developed for FRP environment, with impact of Continuous improvement.	Production & Deployment leading to a Full Rate Production (FRP) decision.
10	Full Rate Production demonstrated and lean production practices in place.	This is the highest level of production readiness. Engineering/design changes are few and generally limited to quality and cost improvements. System, components or items are in rate production and meet all engineering, performance, quality and reliability requirements. All materials, manufacturing processes and procedures, inspection and test equipment are in production and controlled to six-sigma or some other appropriate quality level. FRP unit cost meets goal, funding sufficient for production at required rates. Lean practices well established and continuous process improvements ongoing.	Full Rate Production/ Sustainment

## Acronym and Metric Definitions

**System Effectiveness** – “A probability measure that the system solution can successfully meet an overall operational demand within a given time when operated under specific conditions. System effectiveness reflects the technical characteristics of the system solution (e.g., performance, availability, supportability, dependability). System effectiveness is the ability of the system solution to do the job for which it was intended.

- Single-measures and Multiple-measures can be used to express system effectiveness.
- The objective is to reflect system design attributes and logistics support elements “[BF90]”

**Cost Effectiveness** – “A measure of a system solution in terms of mission fulfillment (system effectiveness) and total life-cycle cost (LCC). Reliability is a major factor in determining the cost effectiveness of a system solution.

- A singular cost effectiveness is hard to measure since many factors that influence the operation and support of a system solution cannot be realistically quantified e.g., interactions effects of other systems, political implications, extreme environmental factors).
- Cost effectiveness can be express in various perspectives, depending on the specific mission or system capability parameters chosen for evaluation.
- A set of cost effectiveness measures are typically used to express the cost effectiveness of a system solution.
  - System effectiveness/LCC
  - System benefits/LCC
  - Availability/LCC
  - System capacity/LCC
  - Supply effectiveness/LCC”[BF90]

**Measures of Merit (MOMs)** – A set of characteristic parameters used to define the effectiveness, performance, usage, and suitability of a system in the context of its operations and fielding. MOMs take into account mission objectives, functions, capabilities, and tactical, strategic, and political constraints.

**Measure of Effectiveness (MOEs)** – A set of operational characteristic parameters that define how well the system performs its overall and assigned missions and executes tasks in operational situations under a given sets of conditions. The MOEs are used to predict, determine, and assess force and system effectiveness. Product development use MOEs for early and continuous verification and validation:

- Design to predict that the system will perform as expected in the intended battlespace
- Development Test & Evaluation (DT&E) to determine whether the system meets its specifications
- Operational Test & Evaluation (OT&E) to determine the operational success of the system

**Measure of Performance (MOP)** – A set of capability parameters that define how well a system performs during operations and execution of assigned tasks. MOPs represent the performance abilities of the system.

**Measure of Usage (MOU)** – A set of operational and sustainment characteristic parameters that define how much the system is utilized or how many supplies are being consumed during operations and execution of assigned tasks.

**Measures of Suitability (MOS)** – A set of appropriateness characteristic parameters that define of how fitting a system is during deployment.

**Key Performance Parameters (KPPs)** – KPPs those critical system characteristics that, when achieved, allow the attainment of operational performance requirements. They are technical measures associated with Joint Capabilities Integration & Development System (JCIDS) documents: Initial Capabilities Document (ICD), Capability Development Document (CDD), and Capability Production Document (CPD).

“KPPs are those attributes of a system that are considered critical or essential to the development of an effective military capability. KPPs must be measurable and testable to enable feedback from test and evaluation efforts to the requirements process. KPPs are validated by the Joint Requirements Oversight Council (JROC) for JROC Interest documents, by the Joint Capabilities Board for JCB Interest documents, and by the DoD component for Joint Integration, Joint Information, or Independent documents. Capability development and capability production document KPPs are included verbatim in the acquisition program baseline.” [CJCS09]

**Key System Attributes (KSAs)** – “KSAs are those system attributes or characteristics considered critical or essential for an effective military capability and considered crucial to achieving a balanced solution/approach to a system, but not critical enough to be designated a KPP. KSAs provide decision makers with an additional level of capability performance characteristics below the KPP level and require a sponsor 4-star, Defense agency commander, or Principal Staff Assistant to change.” [CJCS07]

**TPMs (Technical Performance Measures)** – TPMs are those system attributes created for the purpose of tracking the health of a design, and supporting the decision-making process. They provide indications of design progress and status and/or risk mitigation progress and status. They are monitored on a frequent basis as the design is being completed. TPMs are traceable to requirements and must be measurable parameters.

### **Characteristics of a Good Measure of Merit (MOM)**

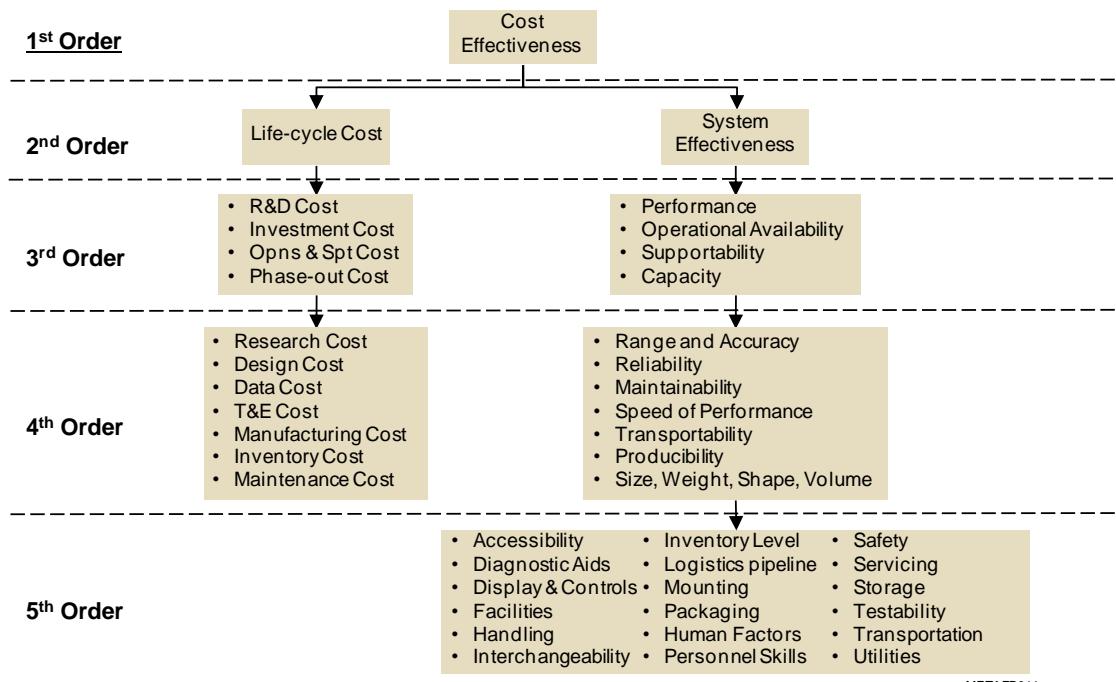
The following are characteristics of well-defined Measure of Merit (MOM)

- **Relevant** – MOMs are pertinent to the missions, functions, capabilities, critical issues, and intended uses of the product.
- **Complete** – The set of MOMs needs to be a complete set of measures to adequately represent and understand the product’s effectiveness, performance, suitability, and utility while fielded in its intended environment. A MOM needs no further amplification for understanding when separated from other MOMs.
- **Clear and Unambiguous** – MOMs are precisely stated in terms that are specific and have only one interpretation.
- **Mutually Exclusive** – Each MOM should be independent of each other to prevent dependency coupling issues.
- **Enduring Importance** – MOMs should have long-term significance for personnel starting from project beginning to closure.

- **Measurable** – MOM inputs need to be assessable using quantification methods. If MOE inputs are assessed qualitatively, then standard measurement criteria is required.

#### 7.1.1.6 Design Concept Discriminator Analysis

Infantry Fighting Vehicle (IFV) product development requires iterative decision making and progressive trade studies starting from concept exploration to the final design in a Technical Data Package (TDP) issued for production. Figure 7.1-8 illustrates examples of criteria measures in a typical hierarchical order that are commonly used in formal decisions and trade studies on DMI development programs. [BF90]



**Figure 7.1-8. Example Order of IFV Criteria Measures for Formal Decisions/Trade Studies**

#### Example 4<sup>th</sup> Order Mobility Criteria Measures for a Notional IFV

The Notional IFV Mission Statement contained in Section 7.1.1.8 was used to refine the common abstract 4<sup>th</sup> order capability criteria measures illustrated in Figure 7.1-8 to a set of typical criteria measures that could be used as design concept mobility discriminators for an IFV or all combat vehicles.

Table 7.1-13 provides example ranges of mobility performance that can be used to assist in automated IFV design concept space reduction or selection of preferred IFV design concept alternatives (Wheeled or Tracked).

**Table 7.1-13. Example Ranges for Mobility Performance**

Example Ranges for Mobility Performance			
No.	Capability	Wheeled	Tracked
		Solutions usually offer high-speed mobility with versatility in maneuver, firepower, and transportability	Solutions for Cross-Country Mobility Dominance for Maneuver or Heavy Firepower
		Major focus is on obtaining high speeds on improved surfaces (primary & secondary) with 2-4 times energy efficiency over track vehicles	Major focus is on obtaining maneuverability on un-improved surfaces (trails & cross-country) and negotiating obstacles
1	<b>Energy Efficiency</b>	<b>5.0 – 9.0 km/gal</b> (1.32 – 2.38 km/liter) @ 50 kph on primary roads	<b>1.0 – 4.0 km/gal</b> (0.26 – 1.06 km/liter) @ 50 kph on primary roads
2	<b>Dash Speed (Acceleration)</b>	<b>70 - 80 kph</b> in 15 – 20 seconds on hard level surface road	<b>40 - 50 kph</b> in 15 – 20 seconds on hard level surface road
3	<b>Highway Speed</b>	<b>80 - 105 kph</b>	<b>30 - 80 kph</b>
4	<b>Cross-country Speed</b>	<b>15 - 25 kph</b> on 1-6 inches of terrain roughness @ 20% of total distance miles	<b>30 - 45 kph</b> on 1-6 inches of terrain roughness and @ 35-50% of the total distance miles
5	<b>Turning Radius</b>	360 degree left and right turn <b>within 1.5 vehicle diagonal length</b>	360 degree left and right turn <b>within 1.0 vehicle diagonal length</b>
6	<b>Weight</b>	<b>20 - 50 tonnes</b>	<b>25 - 80 tonnes</b>
7	<b>Transportability</b>	Deploy a Brigade Combat Team (BCT) of Wheeled Combat Platforms anywhere in the world <b>within 96 hrs</b> , a Division <b>within 120 hrs</b> , and 5 Divisions in 30 days <b>using Strategic Military Lift assets</b> (Airlift - C-17/C-5) and <b>Tactical Lift assets</b> (Fix Wing (C-130) and Rotary Wing (CH-47))	Deploy a Brigade Combat Team (BCT) of Tracked Combat Platforms anywhere in the world <b>within 30 days using Strategic Military Lift assets</b> (Sealift (RORO) and Airlift (C-17/C-5))

### 7.1.1.7 META Project Product Breakdown Structure

A PBS was created to hierarchically organize the products produced by the BAE Systems Team in support of the META program. This PBS was used as a basis for organizing the package structure within the file used to capture and document SysML artifacts produced in support of the META Systems Engineering analysis effort (MagicDraw file “META\_Project.mdzip”, see Figure 7.1-9). Additionally, this PBS was used as a basis for organizing the Systems Engineering directory structure within the Subversion (SVN) version control repository (see Figure 7.1-10).

Table 7.1-14 provides a brief description for each of the PBS elements.

**Table 7.1-14. META Product Breakdown Structure Elements**

No.	Element Name	Description
1	ARRoW Toolset	All software applications and associated products that are integrated into the ARRoW Integrated Development Environment (AIDE). May be developmental or non-developmental items with respect to META project scope.
1.1	System Documentation	The User’s Manual and Version Description Document for the delivered version of the integrated toolset.
1.2	Toolset System Package	The package delivered to the DARPA customer that includes the set of software tools developed along with their supporting documentation.
1.3	Systems Engineering	Any Systems Engineering analysis artifacts specifically associated with the ARRoW Toolset development.
1.4	OTS Tools	Off-the-Shelf Tools. Non-developmental software applications integrated into the AIDE.
2	Component Model Library	Artifacts related to the infrastructure, interfaces, and content of the Component Model Library.
3	Demos	Products and documentation related to all demonstrations presented at the various PI meetings.
4	Vehicle System Model	All requirements, behavioral analysis, and design associated with a notional Combat Fighting Vehicle that is the target product of the AIDE toolset, workflow, and processes.
5	Integrated Toolset Documentation	Development effort and products associated with writing a User’s Manual and Version Description Document.
5.1	Users Manual	The User’s Manual describes the general process for using the toolset and how to use each tool within the toolset.
5.2	Version Description Document	The Version Description Document describes what is included in the release package, known issues, and system requirements and instructions for installing the toolset.
6	Systems Engineering	All Systems Engineering analysis artifacts associated with development of the AIDE toolset.
6.1	Requirements Development	All effort and artifacts associated with defining the required use of, behavior of, and capability of the AIDE.

No.	Element Name	Description
6.1.1	Actors	Description of human and non-human actors that interface with the AIDE toolset.
6.1.2	CONOPS	Concept of Operations analysis and documentation of the AIDE.
6.1.3	Requirements	Textual requirements for the AIDE.
6.1.4	Use Cases	Use case analysis and SysML documentation that drives the required behavior of the AIDE.
6.2	Architecture Development	Analysis and documentation of the AIDE architecture.
6.3	Design	Analysis and documentation of the AIDE low level design elements.
6.4	Verification & Validation	Artifacts and analysis related to the Verification & Validation of the AIDE.
6.5	IAT&C	Artifacts and analysis related to the Integration, Assembly, Test and Checkout of the AIDE.
6.6	Specialty Engineering	Any Specialty Engineering artifacts and analysis related to development the AIDE.
7	Program Management	Any analysis and artifacts associated with management of the META program.
7.1	Risk & Opportunity Management	Any artifacts associated with risk and opportunity management of the META program.
7.2	Briefings	Any artifacts associated with META program briefings to the DARPA customer.
7.3	Configuration & Data Management	Any artifacts associated with Configuration & Data Management efforts in support of the META program.
7.4	EVMS	Any artifacts associated with Earned Value Management efforts in support of the META program.

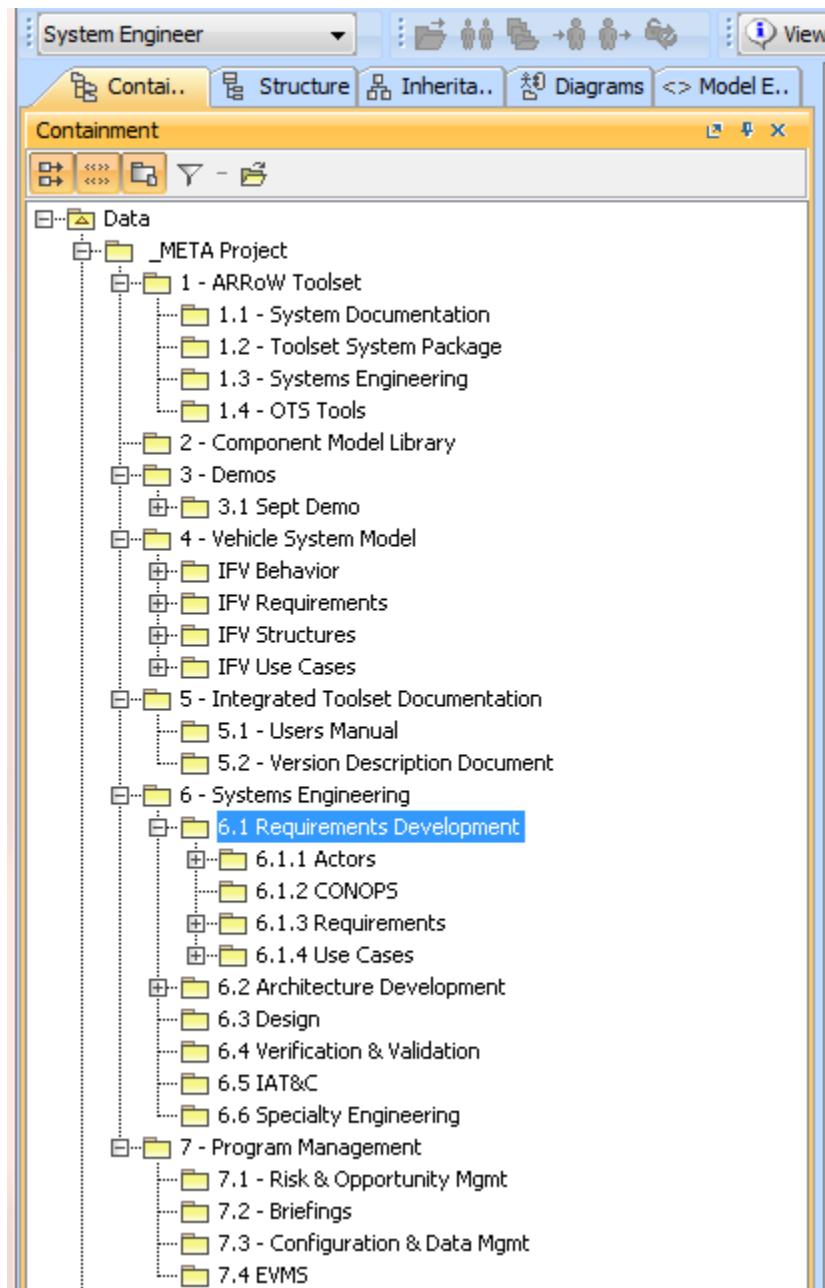
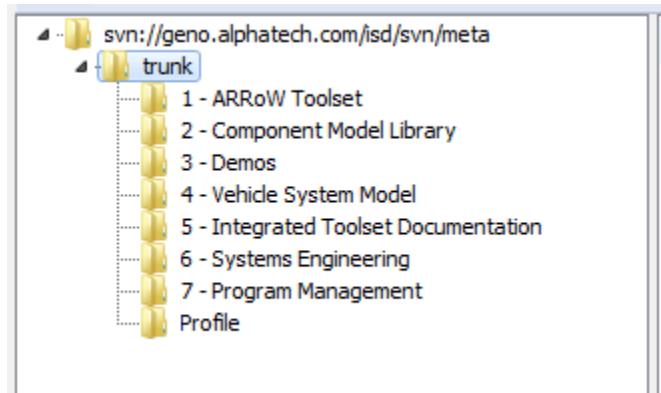


Figure 7.1-9. SysML Package Structure

**Figure 7.1-10. Subversion Repository Structure**

#### 7.1.1.8 Notional IFV System of Interest

A notional Infantry Fighting Vehicle (IFV) containing a representative set of requirements, behavior, structure, and properties was created as a System-of-Interest (SoI) for the ARRoW Integrated Development Environment (IDE).

This section contains the following:

- Notional Defense Material Item (DMI) IFV Problem Statement
- Notional Defense Material Item (DMI) IFV Mission Statement
- Notional IFV requirements analysis
- Notional IFV PBS analysis
- Notional IFV use case and behavioral analyses

**Notional DMI IFV Problem Statement:** Design a notional Infantry Fighting Vehicle (IFV) system such that the total solution is optimized to a set of system-level correctness criteria, and is conformant to the system requirements.

**Notional DMI IFV Mission Statement:** The notional IFV is a versatile medium armored vehicle which provides cross-country mobility dominance, for mounted firepower, communications, and protection to a mounted mechanized infantry squad, overwatch support for a dismounted infantry squad, and deployable anywhere in the world.

##### 7.1.1.8.1 Notional IFV Requirements Analysis

The example requirements serve as a typical set of customer requirements that could be experienced during the development of a complex cyber-physical system, such as an IFV. The Notional IFV requirements analysis identified a subset of mobility performance, environmental conditions, physical characteristics, ownership and support, design and construction requirements common across DMI Surface Vehicle Systems (SVSs) or combat vehicles (Infantry, Armor, and Artillery). Cardinal IFV capabilities include transporting a protection infantry squad and enabling egress of a squad infantry at a dismount point on the battlefield.

The example requirements are generalized to be non-program specific, amped-up or toned-down with no analytical foundation, and can be found on Web sites listed in this section or in military publications similar to [Jane's Military Vehicles](#). An independent assessment concluded that the notional IFV and Ramp Assembly requirements contained in this section and in the

IFV SysML model are not ITAR or company proprietary restricted information. The following references were used to generate the example sets of notional IFV and ramp assembly requirements that are non-ITAR and/or company proprietary restricted:

- Material Need (MN) For An Infantry Fighting Vehicle/Cavalry Fighting Vehicle (IFV/CFV), 2 March 1978
- Operational Requirements Document for Bradley Modernization Program (M2/M3A3), 12 March 2001
- IFV (M2A3) AND CFV (M3A3) Performance Specification Rev J (19207-12465518), 17 December 2009
- Bradley System Specification Rev C (19207-12386023), 1 June 1999
- Crusader System Specification, 14 November 1997
- <http://www.army-guide.com/eng/product364.html>
- [http://www.youtube.com/watch?v=6\\_RMDiLCRRM](http://www.youtube.com/watch?v=6_RMDiLCRRM)
- [http://military.wikia.com/wiki/M2\\_Bradley](http://military.wikia.com/wiki/M2_Bradley)
- [http://en.wikipedia.org/wiki/M2/M3\\_Bradley\\_Fighting\\_Vehicle](http://en.wikipedia.org/wiki/M2/M3_Bradley_Fighting_Vehicle)
- <http://www.fas.org/man/dod-101/sys/land/m2.htm>
- <http://www.army.mil/factfiles/equipment/tracked/bradley.html>
- <http://www.history.army.mil/books/www/256.htm>
- <http://www.history.army.mil/books/www/256.htm>
- [http://images.search.yahoo.com/search/images?\\_adv\\_prop=image&fr=yfp-t-894-s&va=m2+m3+bradley+fighting+vehicle](http://images.search.yahoo.com/search/images?_adv_prop=image&fr=yfp-t-894-s&va=m2+m3+bradley+fighting+vehicle)
- <http://www.army-technology.com/projects/bradley/>
- [http://www.historyofwar.org/articles/weapons\\_bradley.html](http://www.historyofwar.org/articles/weapons_bradley.html)
- <http://militarytechyard.blogspot.com/2009/04/m2m3-bradley-fighting-vehicle.html>
- [http://www.armedforces-int.com/projects/m2\\_m3\\_bradley\\_fighting\\_vehicles.html](http://www.armedforces-int.com/projects/m2_m3_bradley_fighting_vehicles.html)
- [http://pediaview.com/openpedia/M2/M3\\_Bradley\\_Fighting\\_Vehicle#Armament](http://pediaview.com/openpedia/M2/M3_Bradley_Fighting_Vehicle#Armament)
- [http://www.wikinfo.org/index.php/M2\\_Bradley](http://www.wikinfo.org/index.php/M2_Bradley)
- [http://en.citizendium.org/wiki/M2\\_Bradley\\_%28armored\\_fighting\\_vehicle%29](http://en.citizendium.org/wiki/M2_Bradley_%28armored_fighting_vehicle%29)
- [http://www.military-today.com/apc/m2\\_bradley.htm](http://www.military-today.com/apc/m2_bradley.htm)
- [http://www.military-today.com/apc/m3\\_bradley.htm](http://www.military-today.com/apc/m3_bradley.htm)
- <http://www.3ad.org/18inf/documents.htm>

**Example of Notional IFV Requirements in a SysML Model**

Table 7.1-15 provides examples of notional IFV requirements captured in a SysML model.

**Table 7.1-15. Example of Notional IFV Requirements in a SysML Model**

Example of Notional IFV Requirements in a SysML Model			
#	ID	Name	Text
1	IFV1	1.0 Scope	Header
2	IFV2	2.0 Applicable Documents	Header
3	IFV3	3.0 Requirements	Header
4	IFV3-1	3.1 Ramp Assembly Description	Header
5	IFV3-2	3.2 Performance Requirements	Header
6	IFV3-2-3-1-1	3.2.3.1.1 Protect Against Ballistic Threats	The IFV shall provide protection against 14.5 mm machine gun and RPG-7 threats.
7	IFV3-2-5-2-1	3.2.5.2.1 Lower Ramp	The IFV shall achieve an opening ramp duration of not greater than 10 seconds.
8	IFV3-3	3.3 Interface Requirements	Header
10	IFV3-4	3.4 Physical Requirements	Header
11	IFV3-4-1	3.4.1 Weight	The IFV maximum combat weight shall be not greater than 45359.24 kg (TBR 100000 lbs).
12	IFV3-5	3.5 Ownership and Support Requirements	Header
13	IFV3-5-1	3.5.1 Reliability	Header
14	IFV3-5-1-1	3.5.1.1 MTBF	The IFV predicted Mean Time Between Failures (MTBF) shall be greater than 120 hours (Threshold) and 168 hours (Objective) (TBR).
15	IFV3-6	3.6 Environmental Requirements	Header
16	IFV3-7	3.7 Design and Construction Requirements	Header
17	IFV3-7-1	3.7.1 Materials, Processes, and Parts	Header
18	IFV3-7-1-1	3.7.1.1 Watertightness	The IFV shall restrict the entrance of water into the vehicle during fording operations at 48 inches deep.
19	IFV3-7-2	3.7.7 Human Systems Integration	Header
19	IFV3-7-2-1	3.7.7.1 Ingress and Egress	The IFV shall permit ingress and egress of a 95th percentile (in size) male wearing Arctic gear.

## Additional Examples of Notional IFV Requirements

Table 7.1-16 provides additional examples of notional IFV requirements.

**Table 7.1-16. Additional Examples of Notional IFV Requirements**

Additional Examples of Notional IFV Requirements		
No.	Name	Text
1	1.0 Scope	Header
2	1.3 System Overview	The IFV is a tracked, medium armored vehicle which provides cross-country mobility, for mounted firepower, communications, and protection to a mounted mechanized infantry squad, and overwatch support for a dismounted infantry squad.
3	1.4 Document Overview	This document is a “representative set” of performance, functional, interface, and design constraint requirements for an Infantry Fighting Vehicle (IFV). Both mechanized infantry problem and solution domains in breadth and depth are stated as requirements. The requirement statements vary in maturation and quality due to issues such as: necessity, conciseness, measurability, clarity, implementation/design freedom, attainability/feasibility, completeness & stand-alone, consistency, verifiability, singularity, uniqueness, proper level, and positivity.
4	2.0 Applicable Documents	Header
5	3.0 Requirements	Header
6	3.1 Performance Requirements	Unless otherwise specified, performance requirements in the following paragraphs shall be met with the Infantry Fighting Vehicle (IFV) at maximum combat weight, resting on a flat, hard, level surface, and over the range of environmental conditions specified herein. Requirements relating to personnel shall apply to males in the 5th through 95th percentile in stature wearing Mission Oriented Protective Posture (MOPP-IV) gear and Arctic gear.
7	3.1.1 Mobility	Except where otherwise specified, the automotive performance shall be on dry, level, hard-surfaced roads and the IFV shall perform as specified herein without irregular operation, damage to any component, or danger to any crew or squad member.

Additional Examples of Notional IFV Requirements		
No.	Name	Text
8	3.1.1.1 Operational Profile	<p>The IFV shall be capable of 24 continuous hours of combat as follows:</p> <p>a. Sixteen hours shall consist of:</p> <ul style="list-style-type: none"> <li>- 35% (5.6 hr) at rated engine idle speed.</li> <li>- 35% (5.6hr) over cross-country terrain from 2.0 miles per hour (mph) to maximum safe speed.</li> <li>- 20% (3.2 hr) over dirt and gravel roads from 10 mph to maximum safe speed.</li> <li>- 10% (1.6 hr) on hard-surfaced roads at 10 mph to maximum operating speed.</li> </ul> <p>b. Eight hours shall be at silent watch with electrical equipment operated as needed for no more than three continuous hours, depending on ambient temperature, without recharging batteries</p>
9	3.1.1.2 Cruising Range	The IFV shall operate on internally carried fuel for at least 300 miles at an average sustained speed of 30 miles per hour.
10	3.1.1.3 Dash Speed (Acceleration)	The IFV at combat weight shall accelerate from a standing start with the engine idling to 50 mph in not more than 25 sec under nominal conditions. The IFV, at curb weight, shall accelerate from 0 to 50 mph in not more than 20 sec.
11	3.1.1.4 Highway Speed	The IFV shall attain a highway speed of not less than 50 mph.
12	3.1.1.5 Cross-Country Speed	The IFV shall attain a cross-country speed of not less than 28 mph.
13	3.1.1.6 Slope Operation	The IFV shall ascend or descend dry slopes up to 60% either forward or backward, and shall maintain at least 15 mph in the forward direction while climbing hard-surfaced slopes up to 15%. The IFV shall maneuver on dry side slopes up to 45% either forward or backward direction.
14	3.1.1.7 Turning Radius	The IFV shall pivot 360 deg right or left within a 35-ft diameter circle.
15	3.1.1.8 Fording	Under its own power, the IFV without special preparation, shall ford water up to 50 in deep with up to 35% embankment slopes, while retaining full functionality.
16	3.1.1.9 Climb Obstacle	The IFV shall climb obstacles at a height not less than 1.5 m.
17	3.1.1.10 Cross Gap	The IFV shall cross trenches at a width not less than 1.5 m.
18	3.1.1.11 Towing	The IFV, operating either forward or in reverse, shall tow comparable IFVs over cross-country terrain. In the forward direction, the IFV shall be capable of towing such an IFV cross-country at up to 5 mph for 10 miles.
19	3.1.2 Survivability	Header
20	3.1.2.1 Protect Against Ballistic Threats	The IFV shall provide protection against 14.5 mm machine gun and RPG-7 threats.

Additional Examples of Notional IFV Requirements		
No.	Name	Text
21	3.1.3 Auxiliary Systems	Header
22	3.1.3.1 Intercom	The IFV shall accommodate a vehicular intercommunication system with controls at the commander's station and communications ports at each vehicle member station.
23	3.1.3.2 Rear Ramp	The time required for the rear ramp to fully open or close with the engine running shall not exceed 10 sec. The ramp lock mechanism shall permit single hand locking and unlocking.
24	3.1.3.3 Seals	Static seals shall prevent Class II and Class III leaks. Dynamic seals shall prevent Class III leaks.
25	3.1.3.4 Blackout Lighting	Header
26	3.1.3.4.1 Interior Lighting	All interior lights, except lights for turret control and turret drive power indication, shall extinguish automatically when either the rear ramp or the rear door is opened.
27	3.1.3.5 Driver's Switches and Indicators	The IFV shall provide the following functions and indicators: a. Ramp Up/Down switch and unlocked indicator
28	3.1.4 Emergency Operations	The IFV shall provide an emergency operation capability in the case of electronics failures. Vehicle operations requiring backup include: a. Fuel Pump operation b. Steering operation c. Transmission operation d. Ramp up/down e. Ramp Lock/Unlock
29	3.2 Physical Characteristics	Header
30	3.2.1 Weight	The air shipping weight of the IFV shall not exceed 60,000 lb. The curb weight shall not exceed 100,000 lb. The maximum combat weight shall not exceed 120,000 lb.
31	3.2.2 Dimensions	The dimensions when configured for shipping, height shall not exceed 120 in, width 110 in, and length 250 in.
32	3.2.3 Angle of Approach/Angle of Departure	The angle of approach for the IFV, defined as the angle between the ground and a line through the forward most part of the hull and track, shall be a minimum of 75 deg. The angle of departure, defined as the angle between the ground and the rear-most part of the hull and track (excluding the pintle) up to at least 40 in, shall be a minimum of 50 deg.
33	3.2.4 Ground Clearance	The IFV shall have a minimum ground clearance to the bottom of the hull of 18 in at the front and 16 in at the rear.
34	3.2.5 Interior Arrangement	Header

Additional Examples of Notional IFV Requirements								
No.	Name	Text						
35	3.2.5.1 Personnel Seating Capacity	<p>The IFV shall provide seats for personnel as shown in table XV.</p> <p style="text-align: center;">Table XV. Personnel Seating Capacity.</p> <table border="1"> <thead> <tr> <th>Configuration</th><th>Number of Crew Members</th><th>Number of Squad Members</th></tr> </thead> <tbody> <tr> <td>IFV Personnel</td><td>3</td><td>7</td></tr> </tbody> </table>	Configuration	Number of Crew Members	Number of Squad Members	IFV Personnel	3	7
Configuration	Number of Crew Members	Number of Squad Members						
IFV Personnel	3	7						
36	3.2.5.2 Space Allowance	Space calculations shall use a 95th percentile (in stature) male wearing Arctic clothing and MOPP-IV gear. Space allocation for the squad members, driver, gunner, and commander shall be in accordance with MIL-HDBK-759B. Interior stowage space shall be provided for the fighting equipment of the squad.						
37	3.2.6 Ramp	<p>The IFV shall include a ramp at its rear that permits rapid entry and exit of personnel and supplies. The ramp shall include a door. The ramp shall satisfy the following requirements:</p> <ul style="list-style-type: none"> <li>a. Incorporates a quick-opening/closing device, an internal hold-closed locking device, and a hold-open device.</li> <li>b. Incorporates an automatic blackout switch.</li> <li>c. Restricts the entrance of water into the IFV during fording.</li> <li>d. Has a means of being padlocked from the outside.</li> <li>e. Permits side-by-side mount/dismount of two 95th percentile (in stature) males wearing Arctic clothing and MOPP-IV gear.</li> </ul>						
38	3.3 Environmental Conditions	Header						
39	3.3.1 Storage and Transport	The IFV shall be capable of being stored and in transit without sustaining damage under the climate design types hot, basic, cold, and severe cold, including all daily cycle categories as defined in AR 70-38 table 2-1; i.e., -60 °F to +160 °F induced air temperature.						
40	3.3.1.1 Storage and Transit Humidity	The IFV shall be capable of being stored and in transit without sustaining damage under the climatic design types hot, basic, cold, and severe cold, including all daily cycle categories as defined in AR 70-38 table 2-1; i.e., nil to 100% induced relative humidity.						
41	3.3.1.2 Storage	The IFV shall not require preservation for storage less than 120 days. The IFV shall require preservation prior to storage exceeding 120 days.						
42	3.3.1.3 Altitude	The IFV shall be capable of being stored and in transit up to 40,000 ft above sea level.						
43	3.3.2 Operating Conditions	Header						
44	3.3.2.1 Climate	The IFV shall be capable of operating under the conditions specified in AR 70-38, for the climatic categories hot and basic without a cold start aid, and categories cold and severe cold with an aid, with the exceptions in paragraph 3.3.2.2.						

Additional Examples of Notional IFV Requirements		
No.	Name	Text
45	3.3.3 Steam and Waterjet Cleaning	The IFV shall demonstrate no performance degradation and show no evidence of damage or deformation following a steam and waterjet cleaning process which uses a cleaner conforming to P-C-437 Type II, P-D220D, or commercial equivalent. Jet pressure shall be 100 +/-10 pounds per square inch gage (psig) for steam and 40 +/-10 psig for water. The jet shall be applied perpendicular to the assembly from a distance of not more than 1 ft for steam and not more than 3 ft for water. The assembly shall be subjected to the jet at the rate of not less than 1 ft <sup>2</sup> /min.
46	3.5 Reliability	The IFV including Government furnished equipment shall maintain at least 500 Mean Miles Between Failures (MMBF) when operated as described in 3.1.1.1. The IFV Mean Time Between Failures (MTBF) shall be greater than 120 hours (Threshold) and 168 hours (Objective).
47	3.6 Availability	The IFV including government furnished equipment shall maintain achieved availability of at least 0.80 when operated as described in 3.1.1.1. Achieved availability is defined as the ratio of operating time to the total of operating and maintenance time.
48	3.7 Safety	The IFV shall ensure the highest degree of safety and health consistent with mission requirements throughout its life cycle.
49	3.8 Logistics/Diagnostics	Header
50	3.8.1 Built-In Test (BIT)	Header
51	3.8.1.1 Self-Test BIT (SBIT)	SBITs, internal to each subsystem, shall execute automatically upon power up and results shall be displayed within 20 sec of the application of power to the turret electronics.
52	3.9 Transportability	The IFV shall have exterior lifting and tiedown provisions for the modes of transport listed below:
53	3.9.1 Road	The IFV shall be capable of being transported on a Heavy Equipment Transporter.
54	3.9.2 Rail	The IFV shall be capable of being transported over U.S. railways without disassembly. For foreign transport, IFV width requirements may be met by removing side armor and the closure kit, if installed. IFV width requirements described by TM 55-2350-252-14 shall be met.
55	3.9.3 Water	The IFV shall be capable of being transported by break-bulk cargo ships, barge-carriers, roll-on/roll-off ships, and military landing craft.

Additional Examples of Notional IFV Requirements		
No.	Name	Text
56	3.9.4 Air	<p>The IFV shall be capable of being transported in C5 and C17 aircraft in conformance with MIL-STD-1791, “Designing for Internal Aerial Delivery in Fixed Wing Aircraft,” and as described in Air Force Systems Command (AFSC) Design Handbook DH 1-11.</p> <ul style="list-style-type: none"> <li>a. The width, height, and weight shall be reducible for air transport by the IFV crew within one hour, assisted by unit tools and personnel, using organic assets (fork lift, M88, or M578.).</li> <li>b. The IFV will arrive at the loading pad minus its ammunition, Basic Issue Items (BII), weapons, fuel, personal gear, and supplemental armor.</li> </ul>
57	3.10 Design and Construction	Header
58	3.10.1 Materials	All materials, parts, and processes selected for use in the IFV construction shall be compatible with the safety, performance, and environmental requirements as specified herein.
59	3.10.1.1 Fungal growth	Materials used in the IFV shall not support fungal growth.
60	3.10.1.2 Corrosion Resistance	Metals and alloys used in the construction of the IFV that are exposed to corrosive environmental conditions shall be corrosion resistant or shall be coated or metallurgically processed to resist corrosion. Except where impractical, dissimilar metal combinations that promote corrosion through galvanic action shall be insulated to prevent corrosion.
61	6.0 Note	Header
62	6.1 Definitions	Header
63	6.1.1 Curb Weight	The IFV is complete with all components and systems, fully serviced with liquids and one-fourth full fuel tank, with track pads, driver, no OVE, no weapons installed, no other crew or squad aboard, no BII, AAL or ICOEI, no ammunition or water, and no supplemental armor tiles. Items may be simulated by ballast weights located at the appropriate center of gravity.
64	6.1.2 Combat Weight	The IFV is complete with all components and systems, fully serviced with liquids and a full fuel tank, with track pads, all OVE BII, AAL, ICOEI, 25 mm and 7.62 mm weapons installed, all ammunition and water, crew and squad, and supplemental armor tiles installed. Items, such as crew, ammunition, supplemental armor tiles, etc., may be simulated by ballast weights located at the appropriate center of gravity.
65	6.1.3 Approximately	As close as reasonable for the intended purpose. In the opinion of the operator the item being tested will not cause failure or malfunction of the system, or cause the system to not function.
66	6.1.4 Smooth	In the opinion of the operator, the item being tested does not exhibit discernable erratic operation, chatter, jump, bind, skip, or does not prevent the operator from properly functioning the item being tested.

Additional Examples of Notional IFV Requirements		
No.	Name	Text
67	6.1.5 Subjectively	An intuitive and conscious consideration by the operator, that the item being tested, observed, or checked meets or exceeds the intended function.
68	6.1.6 Focus	Clear, without blurriness, objects at a distance of more than 200 m are sharp and clear.
69	6.1.7 Subjective Evaluation	This verification is a subjective evaluation of the operation or response of the system or component in question. Conclusions of success depend on the interpretations of an experienced operator/tester, rather than on numbers derived from instrumentation, bus data, or other quantitative results.
70	6.1.8 Hardware/Software Test	Specific functions, responses, and other parameters of the system or component in question have been measured or determined during Software/Hardware Final Qualification Tests, component tests, or subsystem tests. Therefore, quantitative or instrumented measurements at the system/IFV level may not be required.
71	6.1.9 Previous Tests	Where appropriate, use the procedures and results of tests of other functions as evidence that the requirements of this paragraph are met.
72	6.1.10 Classification of Leaks	Class I: Fluid seepage is not great enough to form drops, but is shown by wetness or color changes. Class II: Fluid leakage is great enough to form drops. Drops do not drip from the item being checked or inspected. Class III: Fluid leakage is great enough to form drops that fall from the item being checked or inspected.

### Notional Ramp Assembly Requirements in a SysML model

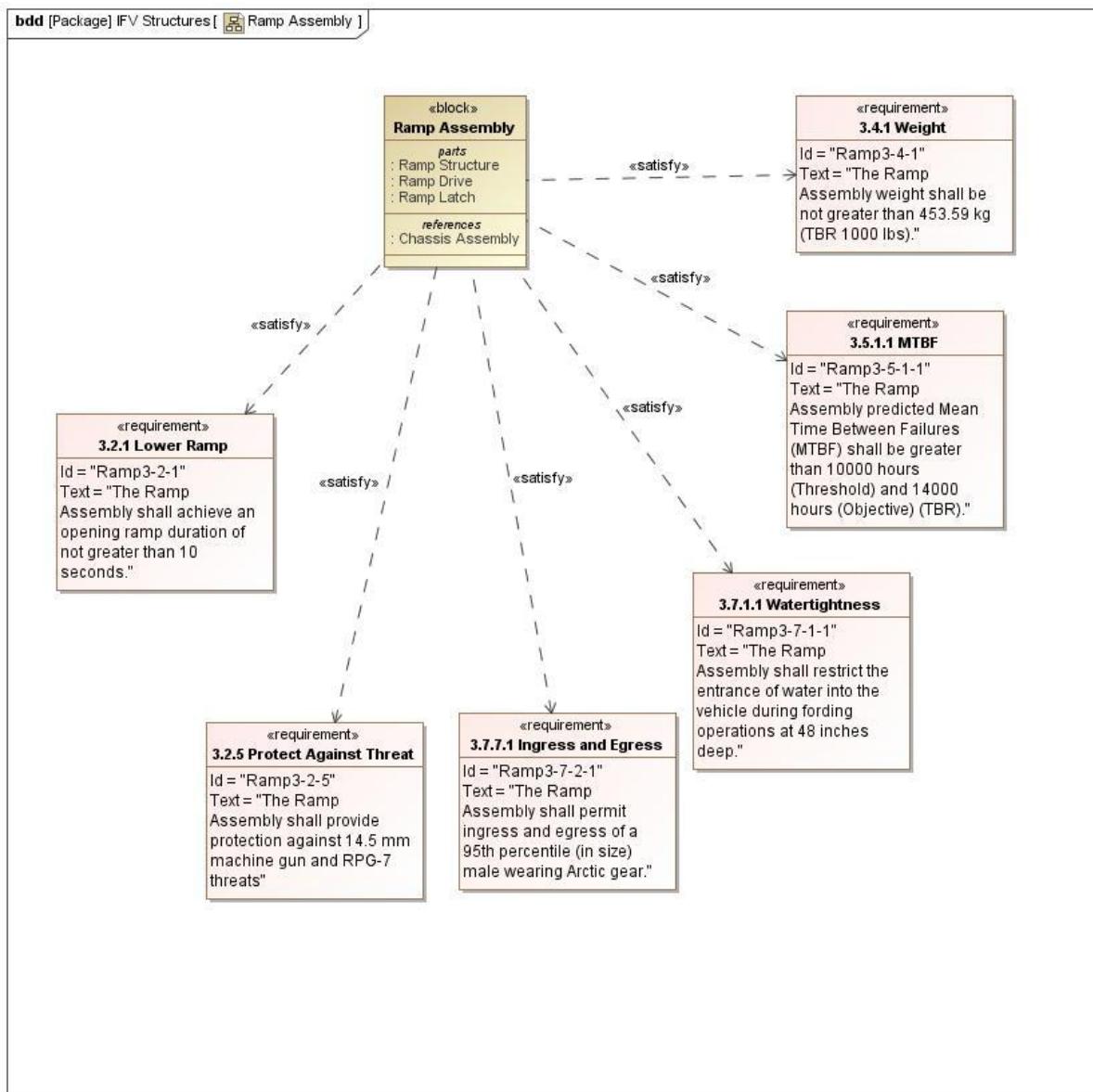
Table 7.1-17 provides notional IFV Ramp Assembly requirements captured in a SysML model.

**Table 7.1-17. Notional Ramp Assembly Requirements in a SysML Model**

Notional Ramp Assembly Requirements in a SysML Model			
No.	ID	Name	Text
1	Ramp1	1.0 Scope	Header
2	Ramp2	2.0 Applicable Documents	Header
3	Ramp3	3.0 Requirements	Header
4	Ramp3-1	3.1 Ramp Assembly Description	The Rear Egress/Ingress assembly is an automated inclined vehicle pathway that connects the vehicle personnel payload compartment with the ground surface.  The Rear Egress/Ingress assembly enables onloading and offloading of personnel for a vehicle.

Notional Ramp Assembly Requirements in a SysML Model			
No.	ID	Name	Text
5	Ramp3-2	3.2 Performance Requirements	Header
6	Ramp3-2-1	3.2.5.2.1 Lower Ramp	The Ramp Assembly shall achieve an opening ramp duration of not greater than 10 seconds.
7	Ramp3-2-2	3.2.5.2.2 Raise Ramp	TBP
8	Ramp3-2-3	3.2.3 Initialize/Prep Ramp	TBP
9	Ramp3-2-4	3.2.4 Shutdown Ramp	TBP
10	Ramp3-2-5	3.2.3.1.1 Protect Against Ballistic Threats	The Ramp Assembly shall provide protection against 14.5 mm machine gun and RPG-7 threats.
11	Ramp3-3	3.3 Interface Requirements	Header
12	Ramp3-4	3.4 Physical Requirements	Header
13	Ramp3-4-1	3.4.1 Weight	The Ramp Assembly weight shall be not greater than 453.59 kg (TBR 1000 lbs).
14	Ramp3-5	3.5 Ownership and Support Requirements	Header
15	Ramp3-5-1	3.5.1 Reliability	Header
16	Ramp3-5-1-1	3.5.1.1 MTBF	The Ramp Assembly predicted Mean Time Between Failures (MTBF) shall be greater than 10000 hours (Threshold) and 14000 hours (Objective) (TBR).
17	Ramp3-6	3.6 Environmental Requirements	Header
18	Ramp3-7	3.7 Design and Construction Requirements	Header
19	Ramp3-7-1	3.7.1 Materials, Processes, and Parts	Header
20	Ramp3-7-1-1	3.7.1.1 Watertightness	The Ramp Assembly shall restrict the entrance of water into the vehicle during fording operations at 48 inches deep.
21	Ramp3-7-2	3.7.7 Human Systems Integration	Header
22	Ramp3-7-2-1	3.7.7.1 Ingress and Egress	The Ramp Assembly shall permit ingress and egress of a 95th percentile (in size) male wearing Arctic gear.

Figure 7.1-11 illustrates an example allocation of requirements to the IFV Ramp assembly.



**Figure 7.1-11. Example Allocation of Requirements to Ramp Assembly**

### 7.1.1.8.2 IFV Product Breakdown Structure (PBS) Analysis

#### PBS-based Reference Model

Department of Defense (DoD) Systems Engineering defines a “Reference Model” as a common conceptual framework for a System-of-Interest (SoI). DoD Handbook (MIL-HDBK-881A) identifies (Product Breakdown Structure) “PBS-based” reference models for Defense Materiel Items (DMIs).<sup>[SEF01]</sup>

A PBS-based Reference model is a direct output of the Architecture Development process and is also a Systems Analysis & Control tool because of its multi-faceted utility on product development, and engineering and project management. It aids the development of a SoI by providing a PBS-based framework for engineering work products in the following process areas:

- Requirements Development (RD) – Product concept of operations & use cases, functional/logical architecture, decomposed & derived requirements, model-based specifications, requirement maturation metrics, requirement repository & management environment
- System Design (SD) - Requirements allocation & flow down, architecture views/view points, architecture/concept/design alternatives, specialty engineering design influence, integrated design & domain engineering, computer aided design & model-based designs and analysis, interface design definition & budgets, technology maturation & growth, SD maturity & health assessment metrics (technical performance measurements (TPMs), requirements compliance, technology readiness assessments (TRA)s & manufacturing readiness assessments (MRAs), state-of-integration readiness)
- System Analysis & Control (SAC)- Effectiveness analysis (cost & system), capability & gap assessments, formal decisions/trades, Risk & opportunity management, configuration & data management, interface (I/F) management
- Verification & Validation (V&V) – Strategy & plans, test cases & procedures results, V&V metrics

A PBS-based reference model also provides structure for:

- Identifying products, processes, data, documents, and models,
- Organizing risk and opportunity management,
- Enabling configuration and data management,
- Organizing integrated product development teams,
- Developing work packages for work orders and material/parts ordering, and
- Organizing technical reviews and audits

## Defense Material Items (DMIs)

MIL-HDBK-881A identifies End/Mission PBS information to aid in the creation of a PBS-based reference model for each of the following DMI types:

- “Surface Vehicle Systems (SVSs)
- Ordnance Systems
- Missile Systems
- Sea Systems
- Aircraft Systems
- Space Systems
- Electronic/Automated SW Systems
- Unmanned Air Vehicle Systems” [AMSC05]

“MIL-HDBK-881A also identifies common products and services to develop, produce, and support the end/mission product. The common products and services are categorized as the following enabling products:

- “Systems Engineering/Program Management (SE/PM)
- System Test and Evaluation (T&E)
- Development Test Evaluation/Operational Test Evaluation (DTE/OTE)
- Training (Equipment, Services, Facilities),
- Data (Technical Publications, Engineering Data, Support Data, Management Data, Data Repository)
- Peculiar Support Equip (PSE) and Common Support Equip (CSE)
- Operational/Site Activation, Industrial Facilities, Initial Spares & Repair Parts” [AMSC05]

## Surface Vehicle System (SVS) PBS-Based Reference Model

The SVS is an abstract and generic structure for all DoD primary and secondary vehicles that navigate over the earth’s surface including manned and unmanned surface systems and amphibious vehicles. The SVS PBS-based reference model provides the framework for the IFV PBS-based sub-reference model and other like sub-reference models depending on vehicle role, mission, or deployment:

- Combat Vehicles (CVs), Combat Support Vehicles (CSVs), and Combat Service Support Vehicles (CSSVs)
- Vehicle roles and/or missions Fires/Effects, Maneuver (Infantry & Armor), Reconnaissance, Engineer, Ordnance, Amphibians, Cargo and Logistics, Transportation, Medical, Food Service, Class III (POL), Mobile Work Units.
- Unmanned Ground Vehicles (UGSs) and Manned Ground Vehicles (MGVs)

## Example IFV PBS-based Reference Model Development

Figure 7.1-12 illustrates MIL-HDBK-811A PBS-based elements of a notional IFV that served as a basis for the creation of an example notional IFV Reference Architecture and PBS-based Design Archetypes for the AIDE.

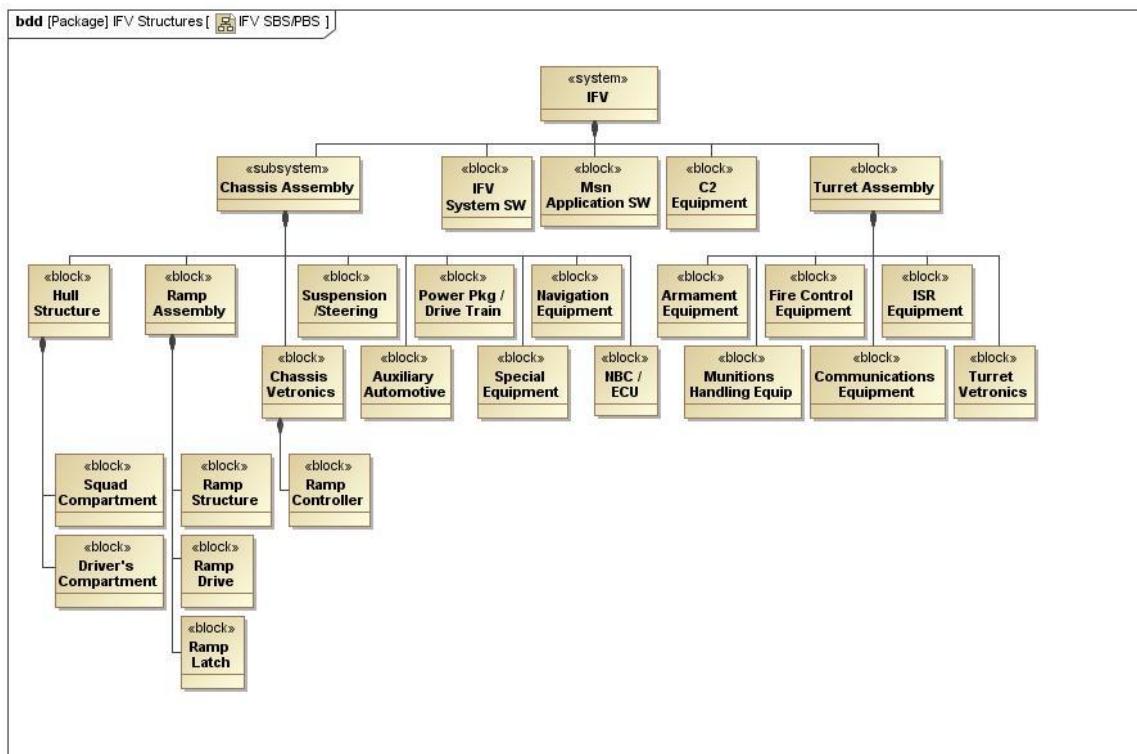


Figure 7.1-12. Notional IFV PBS

Table 7.1-18 provides a brief description for each of the Notional IFV PBS elements.

**Table 7.1-18. Notional IFV PBS Elements**

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1	IFV	<p>A vehicle system with the capability to navigate over the surface. Surface vehicle categories include vehicles primarily intended for general purpose applications and those intended for mating with specialized payloads. Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Cargo and logistics vehicles, mobile work units and combat vehicles</li> <li>b. Combat vehicles serving as armored weapons platforms, reconnaissance vehicles, and amphibians</li> </ul> <p>The mobile element of the system embodying means for performing operational missions. Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Means of propulsion and structure for adaptation of mission equipment or accommodations for disposable loads</li> </ul>
1.1	Chassis Assembly	<p>The vehicle's assembly of structure, compartments and equipment installations required to provide the mobility element of combatant vehicles. Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Hull Structure</li> <li>b. Personnel and weapons compartments</li> <li>c. Chassis Electronics</li> <li>d. Suspension &amp; Steering</li> <li>e. Auxiliary Equipment</li> <li>f. Power Package/Power Train</li> <li>g. Special equipment</li> <li>h. NBC/ECU</li> <li>i. Software</li> </ul>
1.1.1	Hull Structure	<p>The vehicle's primary load bearing component which provides the structural integrity to withstand the operational loading stresses generated while traversing various terrain profiles.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Simple wheeled vehicle frame or combat vehicle hull which satisfies the structural requirements and also provides armor protection</li> <li>b. Structural subassemblies and appendages which attach directly to the primary structure</li> <li>c. Towing and lifting fittings, bumpers, hatches, and grilles</li> <li>d. Provision to accommodate other subsystems such as mountings for suspension, weapons, turret, truck body, cab, special equipment loads</li> </ul>
1.1.1.1	Squad Compartment	<p>The major component to be mated to a chassis to provide a complete vehicle having a defined mission capability.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Accommodations for personnel, cargo, and such subsystems as need to be placed in proximity to operators</li> </ul>

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1.1.1.2	Driver's Compartment	<p>The major component to be mated to a chassis to provide a complete vehicle having a defined mission capability.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Accommodations for personnel, cargo, and such subsystems as need to be placed in proximity to operators</li> </ul>
1.1.2	Ramp Assembly	<p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Ramp structure which satisfies the structural requirements and also provides armor protection</li> <li>b. Structural subassemblies and appendages which attach directly to the ramp structure</li> <li>c. Provision to lower/raise and secure the ramp structure</li> <li>d. Provision to accommodate other subsystems such as mountings for survivability equipment loads</li> </ul>
1.1.2.1	Ramp Structure	<p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Ramp structure which satisfies the structural requirements and also provides armor protection</li> </ul>
1.1.2.2	Ramp Drive	Includes the means to lower and raise the ramp structure
1.1.2.3	Ramp Latch	Includes the means secure the ramp structure when in the closed position
1.1.3	Chassis Vetronics	<p>All hardware/software used to integrate the electronic subsystems and components of the vehicle, such as computer resources, data control and distribution, controls and displays, and power generation and management.</p> <p>Electronic Subsystems and components to be integrated include, for example:</p> <ul style="list-style-type: none"> <li>a. Information systems such as command and control (C2), mission planning and logistics functions, C4ISR</li> <li>b. High end real-time systems such as sensors, robotics, active protection, mission critical applications</li> <li>c. High power load management systems such as the electronic turret, electric drive, autoloader</li> <li>d. Automotive/utility systems such as steering, brake and throttle by wire and the auxiliary load management</li> </ul>
1.1.3.1	Ramp Controller	<p>All hardware/software used to integrate the ramp electronic subsystems and components of the ramp controller, such as computer resources, data control and distribution, controls and displays, and power generation and management.</p> <p>Electronic Subsystems and components to be integrated include, for example:</p> <ul style="list-style-type: none"> <li>a. High end real-time systems such as sensors, mission critical applications</li> <li>b. High power load management systems such as the electronic electric drive</li> </ul>

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1.1.4	Suspension/ Steering	<p>The means for generating tractive efforts, thrust, lift, and steering forces generally at or near the earth's surface and adapting the vehicle to the irregularities of the surface.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Wheels, tracks, brakes, and steering gears for traction and control functions</li> <li>b. Rudder thrust devices and trim vanes for amphibians</li> <li>c. Springs, shock absorbers, skirts, and other suspension members</li> </ul>
1.1.5	Auxiliary Automotive	<p>The group of hardware and software subsystems which provide services to all of the primary vehicle subsystems (as distinguished from the special equipment subsystems) and which outfit the chassis.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. The on-board diagnostics/prognostics system, fire extinguisher system and controls, chassis mounted accessories</li> <li>b. The winch and power take-off, tools and on-vehicle equipment</li> <li>c. Crew accommodations (when otherwise not provided for)</li> </ul> <p>Excludes, for example:</p> <ul style="list-style-type: none"> <li>a. Electrical subsystems and components which are now included in the vetronics WBS element.</li> </ul>
1.1.6	Power Package/ Drive Train	<p>The means for generating and delivering power in the required quantities and driving rates to the driving member.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Engine-mounted auxiliaries such as air ducting and manifolds, controls and instrumentation, exhaust systems, and cooling means</li> <li>b. Power transport components as clutches, transmission, shafting assemblies, torque converters, differentials, final drivers, and power takeoffs</li> <li>c. Brakes and steering when integral to power transmission rather than in the suspension/steering element</li> </ul>
1.1.7	Special Equipment	<p>The special equipment (hardware and software) to be mated to a chassis or a chassis/body/cab assembly to achieve a special mission capability.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. All items required to convert basic vehicle configurations to special-purpose configurations</li> <li>b. Blades, booms, winches, robotic arms or manipulators, etc., to equip wreckers, recovery vehicles, supply vehicles and other field work units</li> <li>c. Furnishings and equipment for command, shop, medical and other special-purpose vehicles</li> </ul>

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1.1.8	Navigation Equipment	<p>The equipment (hardware and software) installed in the vehicle which permits the crew to determine vehicle location and to plot the course of the vehicle.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Navigation systems such as dead reckoning, inertial, and global positioning systems</li> <li>b. Landmark recognition algorithms and processors</li> </ul>
1.1.9	NBC/ECU	<p>The subassemblies or components which provide nuclear, biological, chemical protection and survivability to the vehicle crew, either individually or collectively, during a nuclear, biological, chemical attack.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. A positive pressure system; micro-climate cooling; air conditioning and purification system; ventilated face piece (mask); nuclear, biological, chemical detection and warning devices; decontamination kits; and chemical resistant coatings</li> </ul>
1.2	IFV System SW	<p>That software designed for a specific computer system or family of computer systems to facilitate the operation and maintenance of the computer system and associated programs for the primary vehicle. (ref. ANSI/IEEE Std 610.12)</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Operating systems—software that controls the execution of programs</li> <li>b. Compilers—computer programs used to translate higher order language programs into relocatable or absolute machine code equivalents</li> <li>c. Utilities—computer programs or routines designed to perform the general support function required by other application software, by the operating system, or by system users</li> <li>d. All effort required to design, develop, integrate, and checkout the air vehicle system software including all software developed to support any primary vehicle applications software development</li> <li>e. Primary vehicle system software required to facilitate development, integration, and maintenance of any primary vehicle software build and CSCI</li> </ul> <p>Excludes, for example:</p> <ul style="list-style-type: none"> <li>a. All software that is an integral part of any specific subsystem specification or specifically designed and developed for system test and evaluation</li> <li>b. Software that is an integral part of any specific subsystem, and software that is related to other WBS Level 2 elements</li> </ul>

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1.3	Msn Application SW	<p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. All the software that is specifically produced for the functional use of a computer system or multiplex data base in the primary vehicle (ref. ANSI/IEEE Std 610.12)</li> <li>b. All effort required to design, develop, integrate, and checkout the primary vehicle applications Computer Software Configuration Items (CSCIs)</li> </ul> <p>Excludes, for example:</p> <ul style="list-style-type: none"> <li>a. The non-software portion of air vehicle firmware development and production</li> <li>b. Software that is an integral part of any specific subsystem and software that is related to other WBS Level 2 elements</li> </ul>
1.4	C2 Equipment	<p>All hardware/software used to integrate the Command &amp; Control (C2) subsystems and components of the vehicle, such as computer resources, data control and distribution, controls and displays.</p> <p>C2 Subsystems and components to be integrated include, for example:</p> <ul style="list-style-type: none"> <li>a. Information systems such as mission planning and logistics functions</li> </ul>
1.5	Turret Assembly	<p>The structure and equipment installations required to provide the fighting compartment element of combatant vehicles.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. Turret armor and radiological shielding, turret rings, slip rings</li> <li>b. Attachments and appendages such as hatches and cupolas</li> <li>c. Accommodations for personnel, weapons, and command and control</li> </ul> <p>Excludes, for example:</p> <ul style="list-style-type: none"> <li>a. Fire control and stabilization system</li> </ul>
1.5.1	Armament Equipment	<p>The means for combatant vehicles to deliver fire on hostile targets and for logistics and other vehicles to exercise self-defense.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. main gun, launchers, and secondary armament</li> </ul> <p>Excludes, for example:</p> <ul style="list-style-type: none"> <li>a. Fire control systems</li> </ul>
1.5.2	Munitions Handling Equipment	<p>Automatic Loading. The equipment (hardware and software) for selecting ammunition from a stored position in the vehicle, transferring it, and loading the armament system.</p> <p>Includes, for example:</p> <ul style="list-style-type: none"> <li>a. The means to eject spent cases and misfired rounds</li> <li>b. Ammunition storage racks, transfer/lift mechanisms, ramming and ejecting mechanisms, as well as specialized hydraulic and electrical controls</li> </ul>

Notional IFV PBS Element Definitions		
Element Level #	Element Name	Description
1.5.3	Fire Control Equipment	The equipment (hardware and software) installed in the vehicle which provides intelligence necessary for weapons delivery such as launching and firing. Includes, for example: a. Radars and other sensors necessary for search, recognition and/or tracking b. Controls and displays c. Sights or scopes d. Range finders, computers, computer programs, turret and gun drives, and stabilization systems
1.5.4	Communications Equipment	The equipment (hardware and software) within the system for commanding, controlling, and transmitting information to vehicle crews and other personnel exterior to operating vehicles. Includes, for example: a. Radio frequency equipment, microwave and fiber optic communication links, networking equipment for multiple vehicle control, and intercom and external phone systems b. Means for supplementary communication like visual signaling devices c. Navigation system and data displays not integral to crew stations in the turret assembly or the driver's automotive display in the cab
1.5.5	Intelligence, Reconnaissance, Surveillance (ISR) Equipment	All hardware/software used to integrate the intelligence, surveillance, reconnaissance (ISR) subsystems and components of the vehicle, such as computer resources, data control and distribution, controls and displays.
1.5.6	Turret Vetronics	All hardware/software used to integrate the electronic subsystems and components of the turret, such as computer resources, data control and distribution, controls and displays, and power generation and management. Electronic Subsystems and components to be integrated include, for example: a. Information systems such as command and control (C2), mission planning and logistics functions, C4ISR b. High end real-time systems such as sensors, robotics, active protection, mission critical applications c. High power load management systems such as the electronic turret, electric drive, autoloader

### Example Mobility Components and Properties

The following organization of mobility subsystems for a notional IFV, a specialized combat vehicle, into three subsystems provides design alternative flexibility for both wheeled and tracked combat vehicle concepts:

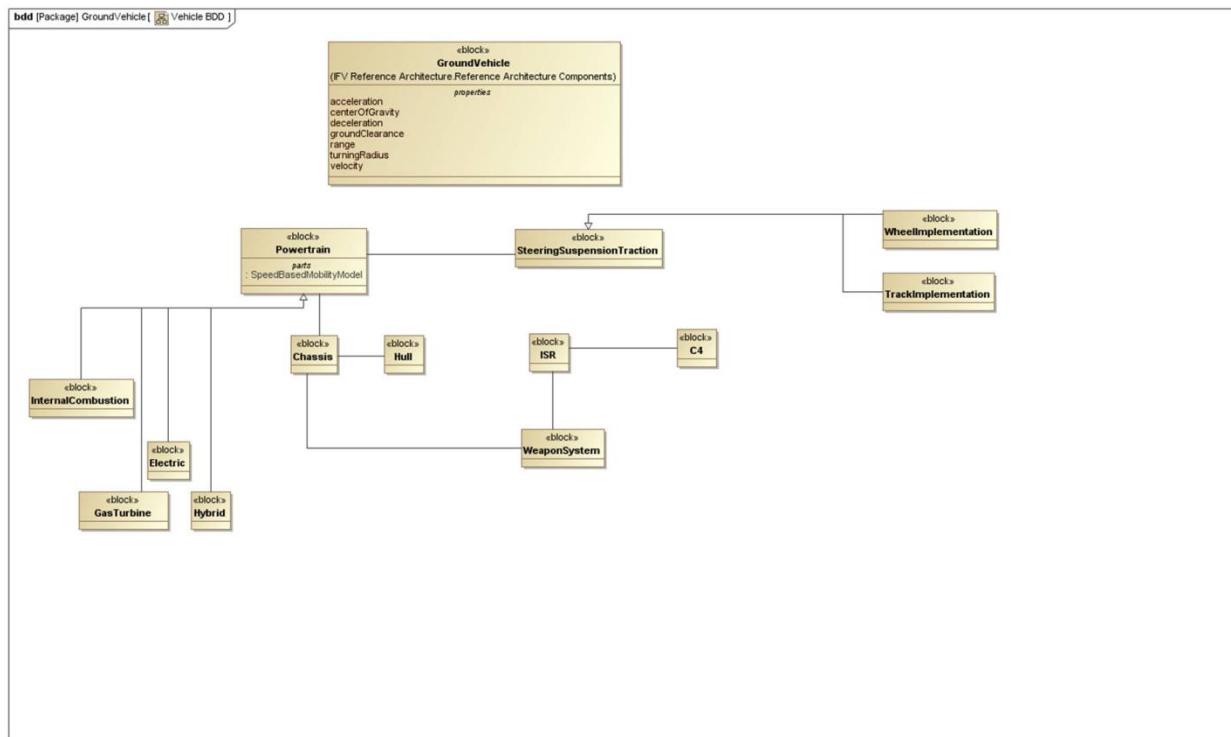
- Power Package/Power Train (Engine & Transmission)
- Steering & Braking
- Suspension

The steering and braking subsystem is integral to the power package/train subsystem for tracked combat vehicle concepts and integral to the suspension subsystem for wheeled combat vehicle concepts. MIL-HDBK-881A served as a guide to organize the mobility subsystems and lower level components.

Table 7.1-19 identifies an initial example of notional IFV mobility subsystem components and properties. Figure 7.1-13 illustrates an example of a notional IFV PBS with affiliated product design properties using a reference architecture design archetype. For example, initial properties are identified for acceleration (dash speed), center of gravity, deceleration (retardation), ground clearance, cruising range, turning radius, and velocity (speeds for cross country and highway travel) at the notional IFV level.

**Table 7.1-19. Initial Example of Mobility Components and Properties**

		Subsystem			
Property	Unit of Measure	Power Package/Power Train	Steering & Braking	Suspension	
		Component			
		Engine	Power Transport (Transmission)		
Power	hp (kW)	x	x		
Torque	lb-ft (N-m)	x	x		
Dimensions/Space Claim (h x w x l)	in (mm) x in (mm) x in (mm)	x	x		
Weight	lbs (kg)	x	x		



**Figure 7.1-13. Notional IFV Reference Architecture Components and Properties**

Table 7.1-20, Table 7.1-21, Table 7.1-22, and Table 7.1-23 provide further details on mobility subsystem components and properties to be taken into consideration when further developing the ARRoW design engineering and analysis tools for a notional IFV.

**Table 7.1-20. Power Package/Power Train Subsystem Components and Properties**

		Subsystem	1 <sup>st</sup> Tier Components	
Property	Unit of Measure	Power Package/ Power Train	Engine	Power Transport
Lower Level Components				Transmission, Torque Converter, Clutch, Shaft Assembly, Differential, Final Drive, Power Take-Off
Type	Various	x	Size X.X (Inline/V/Rotary) # cylinders	x
Power	hp (kW)	x	x	x

		Subsystem	1 <sup>st</sup> Tier Components	
Property	Unit of Measure	Power Package/ Power Train	Engine	Power Transport
Gross Input Power (Max)	hp (kW)			x
Gross Net Power (Max)	hp (kW)			x
Torque	lb-ft (N-m)	x	x	x
Gross Input Torque (Max)	lb-ft (N-m)			x
Gross Net Torque (Max)	lb-ft (N-m)			x
Dimensions (h x w x l)	in (mm) x in (mm) x in (mm)	x	x	x
Weight	lbs (kg)	x	x	x
Rate Speed	min-max rpm	x	x	x
Bore x Stroke (dia x dist)	in (mm) x in (mm)	x	x	
Displacement	cid (cc)	x	x	
Materials	Various	x	x	x
Cost	\$	x	x	x
Mounting	list of connections	x	x	x
Orientation		x	x	x
Gearing	# of speeds	x		x
Fwd Speeds	# of speeds	x		x
Rwd Speeds	# of speeds	x		x
Drive Selection	All Wheel, Selective, Full time	x		x
Power Takeoff (PTO)		x		x

Table 7.1-21. Steering &amp; Braking Subsystem Components and Properties

		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure	Steering & Braking	Steering Gears	Brakes	Rudder Thrust Devices	Trim Vanes
Type		x				
Brake Power	hp (kW)					

		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure	Steering & Braking	Steering Gears	Brakes	Rudder Thrust Devices	Trim Vanes
Dimensions (h x w x l)	in (mm) x in (mm) x in (mm)	x				
Weight	lbs (kg)	x				
Materials	Various	x				
Cost	\$	x				
Mounting	list of connections	x				
Orientation		x				

**Table 7.1-22. Suspension Subsystem Components & Common Properties**

Suspension Subsystem Components & Common Properties						
		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure or Description	Suspension	Springs & Dampers	Wheels	Tracks	Skirts
Lower Level Components			Shock Absorbers, Torsion Bars, Struts			
Type	Various	x	x	x	x	x
Dimensions – Space Occupied (h x w x l)	in (mm) x in (mm) x in (mm)	x	x	x	x	x
Weight	lbs (kg)	x	x	x	x	x
Materials	Various	x	x	x	x	x
Cost	\$	x	x	x	x	x
Mounting	list of connections	x	x	x	x	x
Orientation	Forward, Rear, Center, Left Right, Top, Bottom	x	x	x	x	x

**Table 7.1-23. Suspension Subsystem Component Unique Properties**

Suspension Subsystem Component Unique Properties [WS11]						
		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure or Description	Suspension	Springs & Dampers	Wheels	Tracks	Skirts
Spring Rate		x	x			
1) Used to isolate vehicle from terrain 2) A ratio used to measure how resistant a spring is to being compressed or expanded during the spring's deflection						
Wheel Rate		x				
The effective spring rate when measured at the wheel						
Roll Couple %		x				
1) The effective wheel rate, in roll, of each axle of the vehicle as a ratio of the vehicle's total roll rate. 2) Critical in accurately balancing the handling of a vehicle. 3) Commonly adjusted through the use of anti-roll bars, but can also be changed through the use of different springs.						
Weight Transfer		x				
1) The total amount of weight transfer is affected by four factors: the distance between wheel centers (wheelbase in the case of braking, or track width in the case of cornering) the height of the center of gravity, the mass of the vehicle, and the amount of acceleration experienced. 2) The speed at which weight transfer occurs as well as through which components it transfers is complex and is determined by many factors including but not limited to roll center height, spring and damper rates, anti-roll bar stiffness and the kinematic design of the suspension links. 3) Weight transfer during cornering, acceleration or braking is usually calculated per individual wheel and compared with the static weights for the same wheels.						
Unsprung Wgt Xfer		x				
Unsprung weight transfer is calculated based on the weight of the vehicle's components that are not supported by the springs. This includes tires, wheels, brakes, spindles, half the control arm's weight and other components. These components are then (for calculation purposes) assumed to be connected to a vehicle with zero sprung weight. They are then put through the same dynamic loads. The weight transfer for cornering in the front would be equal to the total unsprung front weight times the G-Force times the front unsprung center of gravity height divided by the front track width. The same is true for the rear.						
Sprung Wgt Xfer		x				

Suspension Subsystem Component Unique Properties [WS11]						
		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure or Description	Suspension	Springs & Dampers	Wheels	Tracks	Skirts
<p>Sprung weight transfer is the weight transferred by only the weight of the vehicle resting on the springs, not the total vehicle weight. Calculating this requires knowing the vehicle's sprung weight (total weight less the unsprung weight), the front and rear roll center heights and the sprung center of gravity height (used to calculate the roll moment arm length). Calculating the front and rear sprung weight transfer will also require knowing the roll couple percentage.</p> <p>The roll axis is the line through the front and rear roll centers that the vehicle rolls around during cornering. The distance from this axis to the sprung center of gravity height is the roll moment arm length. The total sprung weight transfer is equal to the G-force times the sprung weight times the roll moment arm length divided by the effective track width. The front sprung weight transfer is calculated by multiplying the roll couple percentage times the total sprung weight transfer. The rear is the total minus the front transfer.</p>						
Jacking Forces		x				
<p>Jacking forces are the sum of the vertical force components experienced by the suspension links. The resultant force acts to lift the sprung mass if the roll center is above ground, or compress it if underground. Generally, the higher the roll center, the more jacking force is experienced.</p>						
Travel		x				
<p>Travel is the measure of distance from the bottom of the suspension stroke (such as when the vehicle is on a jack and the wheel hangs freely) to the top of the suspension stroke (such as when the vehicle's wheel can no longer travel in an upward direction toward the vehicle).</p>						
Damping		x	x			
<p>Damping is the control of motion or oscillation, as seen with the use of hydraulic gates and valves in a vehicle's shock absorber. This may also vary, intentionally or unintentionally. Like spring rate, the optimal damping for comfort may be less than for control.</p>						
<p>Damping controls the travel speed and resistance of the vehicle's suspension. An undamped car will oscillate up and down. With proper damping levels, the car will settle back to a normal state in a minimal amount of time. Most damping in modern vehicles can be controlled by increasing or decreasing the resistance to fluid flow in the shock absorber.</p>						
Camber Control		x				
<p>Camber changes due to wheel travel, body roll and suspension system deflection or compliance. In general, a tire wears and brakes best at -1 to -2° of camber from vertical. Depending on the tire and the road surface, it may hold the road best at a slightly different angle. Small changes in camber, front and rear, can be used to tune handling. Some race cars are tuned with -2~7° camber depending on the type of handling desired and the tire construction. Oftentimes, too much camber will result in the decrease of braking performance due to a reduced contact patch size through excessive camber variation in the suspension geometry. The amount of camber change in bump is determined by the instantaneous front view swing arm (FVSA) length of the suspension geometry, or in other words, the tendency of the tire to camber inward when compressed in bump.</p>						
Roll Center Height		x				

Suspension Subsystem Component Unique Properties [WS11]						
		Subsystem	1 <sup>st</sup> Tier Components			
Property	Unit of Measure or Description	Suspension	Springs & Dampers	Wheels	Tracks	Skirts
This is important to body roll and to front to rear roll stiffness distribution. However, the roll stiffness distribution in most cars is set more by the antiroll bars than the RCH. The height of the roll center is related to the amount of jacking forces experienced.						
Instant Center		x				
Due to the fact that the wheel and tire's motion is constrained by the suspension links on the vehicle, the motion of the wheel package in the front view will scribe an imaginary arc in space with an "instantaneous center" of rotation at any given point along its path. The instant center for any wheel package can be found by following imaginary lines drawn through the suspension links to their intersection point.						
Anti-Dive & Anti Squat		x				
Anti-dive and anti-squat are percentages and refer to the front diving under braking and the rear squatting under acceleration. They can be thought of as the counterparts for braking and acceleration as jacking forces are to cornering. The main reason for the difference is due to the different design goals between front and rear suspension, whereas suspension is usually symmetrical between the left and right of the vehicle.						
Flexibility & Vibration Modes of Suspension Element		x				
In modern cars, the flexibility is mainly in the rubber bushings. For high-stress suspensions, such as off-road vehicles, polyurethane bushings are available, which offer far more longevity under greater stresses.						
Isolation from High Frequency Shock		x				
For most purposes, the weight of the suspension components is unimportant, but at high frequencies, caused by road surface roughness, the parts isolated by rubber bushings act as a multistage filter to suppress noise and vibration better than can be done with only the tires and springs. (The springs work mainly in the vertical direction.)						
Contribution to Unsprung Wgt and Total Wgt		x				
These are usually small, except that the suspension is related to whether the brakes and differential(s) are sprung.						
Force distribution		x				
The suspension attachment must match the frame design in geometry, strength and rigidity.						
Air Resistance (drag)		x	x	x	x	x
Certain modern vehicles have height adjustable suspension in order to improve aerodynamics and fuel efficiency. And modern formula cars, that have exposed wheels and suspension, typically use streamlined tubing rather than simple round tubing for their suspension arms to reduce drag. Also typical is the use of rocker arm, push rod, or pull rod type suspensions, that among other things, places the spring/damper unit inboard and out of the air stream to further reduce air resistance.						

### 7.1.1.8.3 Notional IFV Use Case and Behavioral Analyses

The section contains notional IFV use case and behavioral views used to develop ARRoW design and analysis tools. This section also identifies and describes the actors associated with IFV use cases.

#### Example Breadth of Notional IFV Mounted Operations Use Cases

Figure 7.1-14 illustrates a breadth of use case examples and affiliated actors for IFV Mounted Operations. The example set of mounted operations use cases includes:

- Startup and shutdown of an IFV by an IFV crew member
- Manage IFV power by an IFV crew member
- Control vehicle movement and maneuver a mounted infantry squad by the driver and/or squad member
- Command & control the IFV operations by the vehicle commander, squad leader, or squad member
- Deliver IFV effects by the vehicle commander or gunner
- Exploit IFV intelligence, surveillance, and reconnaissance ISR/situation awareness (SA) by the vehicle commander or gunner
- Communicate with operational nodes by the vehicle commander
- Sustain the IFV by a mechanic

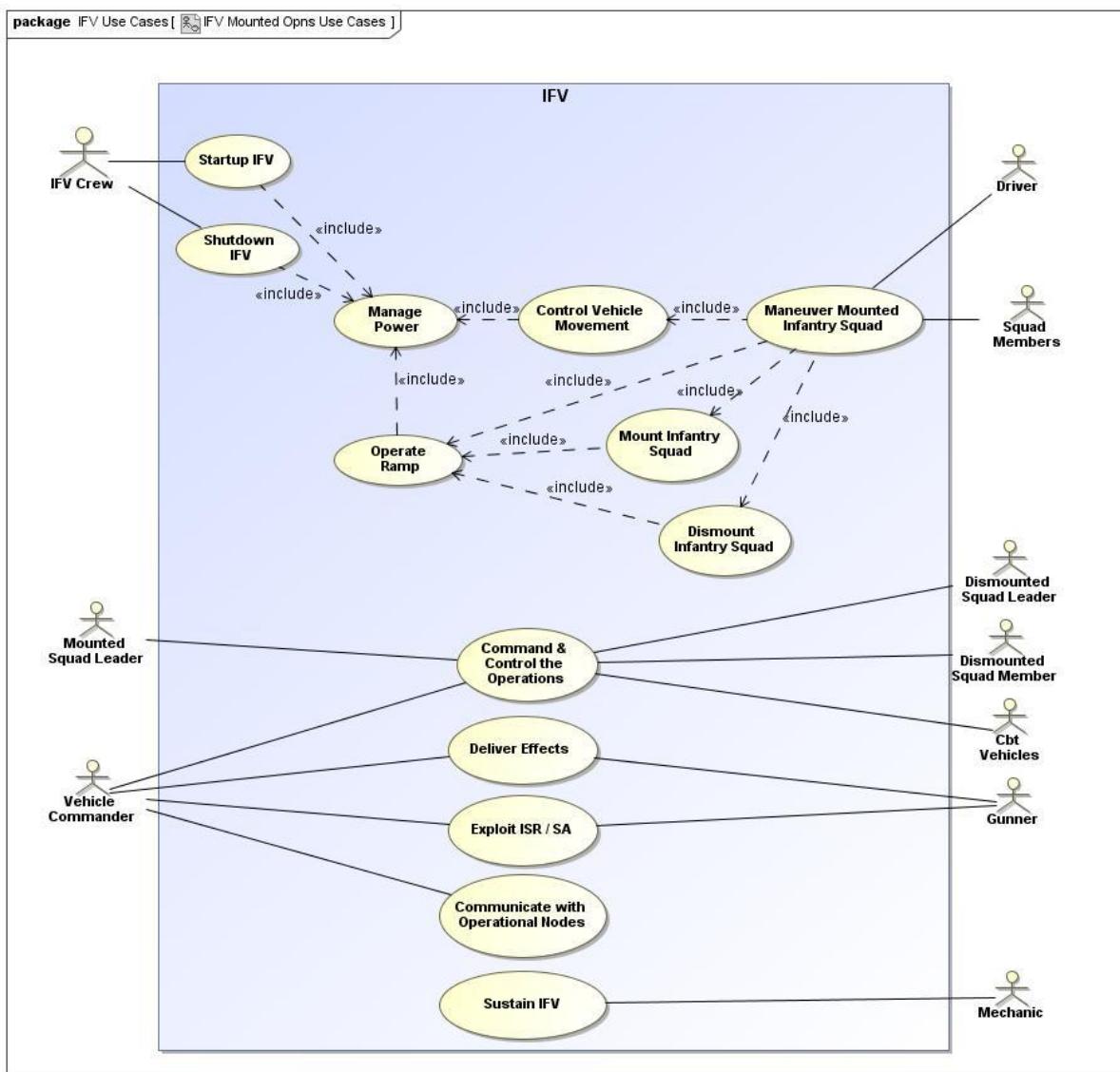
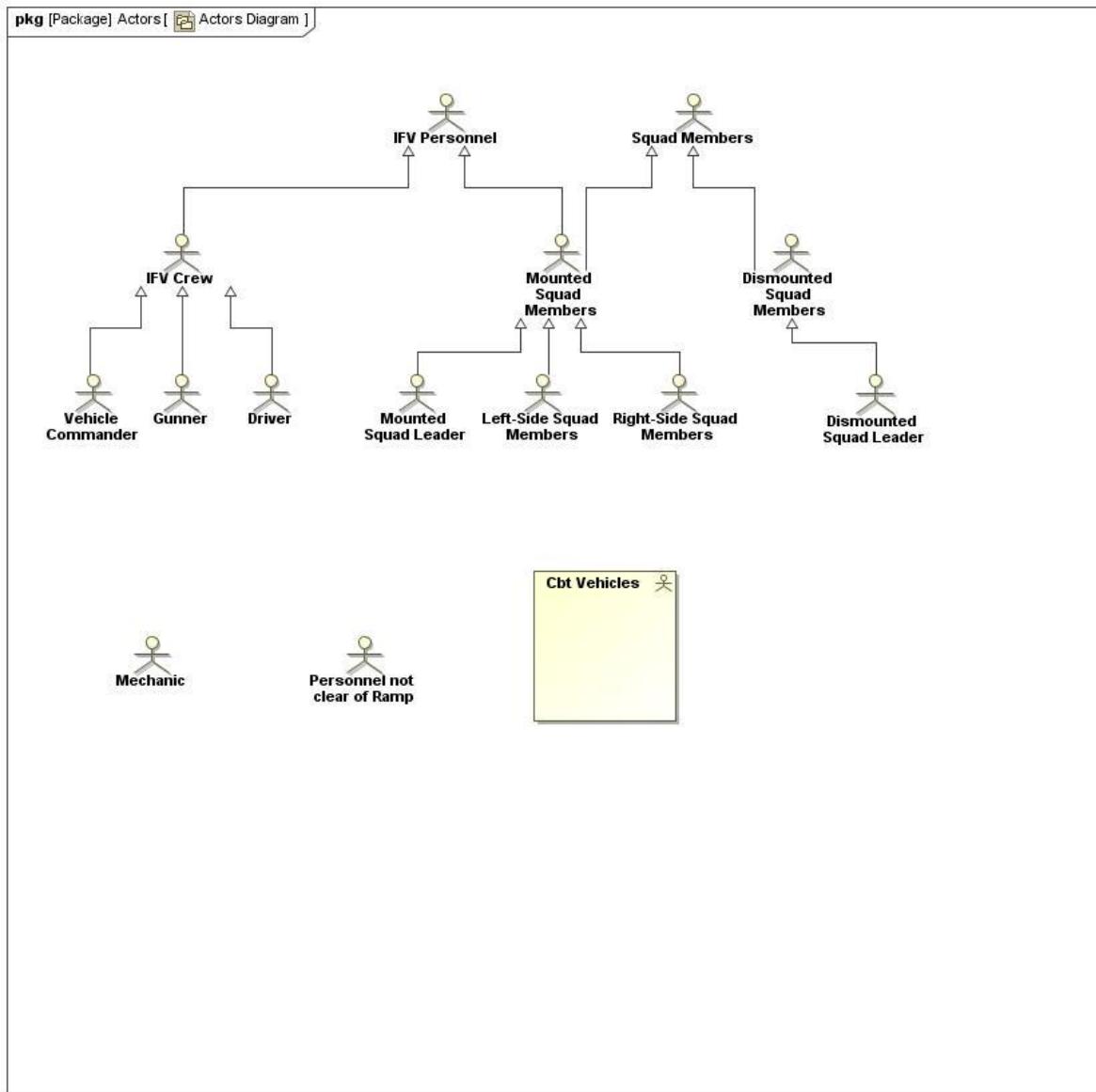


Figure 7.1-14. Notional IFV Mounted Operations Use Cases Diagram

## Actors

Figure 7.1-15 illustrates an example set of actors affiliated with the happy path, design limit and safety testing, and bad day testing for IFV operations. The set of actors that interact with the IFV range from mounted squad members, dismounted squad members, vehicle crew members to other vehicles and personnel that could be in proximity to the IFV operations.



**Figure 7.1-15. Notional IFV Actors Diagram**

Table 7.1-24 provides example descriptions for each of the actors that could be affiliated with the happy path, performance envelope, and bad day testing for IFV operations.

**Table 7.1-24. Example Actor Descriptions**

Actor Name	Description
IFV Personnel	Represents all IFV crew members and mounted squad members.
IFV Crew	Represents all IFV crew member roles of vehicle commander, gunner, or driver.
Vehicle Commander	Represents the IFV crew member commanding the overall operations and use of the IFV. The Vehicle Commander issues commands to the driver for IFV movement and maneuver and ramp operations, and the gunner for weapon operations and target engagement and intelligence, surveillance, and reconnaissance operations.
Gunner	Represents the IFV crew member operating an infantry support weapon such as a main gun/cannon, secondary gun/machine gun, anti-tank guided missile launcher, or automatic grenade launcher,
Driver	Represents the IFV crew member operating the IFV for movement and maneuver, and ramp operations.
Squad Members	Represents all squad positions/roles of squad leader, fire team leader, rifleman, automatic rifleman, grenadier, or designated marksman.
Mounted Squad Members	Represents all squad positions/roles of squad leader, fire team leader, rifleman, automatic rifleman, grenadier, or designated marksman while being transported in a personnel carrying vehicle.
Mounted Squad Leader	Represents the first-line supervisor in an enlisted individual's chain of command while being transported in a personnel carrying vehicle. The Squad Leader is in command of the squad and issues orders to Fire Team Leaders. This individual is responsible for the daily activities that an individual performs, to include any career-mandated training. Ultimately, this individual becomes a mentor, trainer, role model, and supervisor all in one.
Left Side Squad Members	Represents all squad positions/roles of squad leader, fire team leader, rifleman, automatic rifleman, grenadier, or designated marksman while being transported on the left side of a personnel carrying vehicle.
Right Side Squad Member	Represents all squad positions/roles of squad leader, fire team leader, rifleman, automatic rifleman, grenadier, or designated marksman while being transported on the right side of a personnel carrying vehicle.
Dismounted Squad Members	Represents all squad positions/roles of squad leader, fire team leader, rifleman, automatic rifleman, grenadier, or designated marksman while dismounted from a personnel carrying vehicle.
Dismounted Squad Leader	Represents the first-line supervisor in an enlisted individual's chain of command when dismounted from a personnel carrying vehicle. Same responsibilities as the mounted squad leader

Actor Name	Description
Mechanic	Represents all maintenance personnel tasked to perform organizational or higher level organizational maintenance on the IFV and that could be in close proximity to IFV ramp assembly operations.
Personnel Not Clear of the Vehicle	Represents all personnel types that could be in close proximity to IFV ramp assembly operations. Includes: crew members, mounted squad members, dismounted squad members, maintainers, etc.
Cbt Vehicles	Represents all vehicle types that could be in close proximity to IFV ramp assembly operations. Includes: combat vehicles (e.g., IFV), combat support vehicles (e.g., engineer), and combat service support vehicles (e.g., medical, ammunition, maintenance).

### Definitions for Ramps, Doors, Hatches

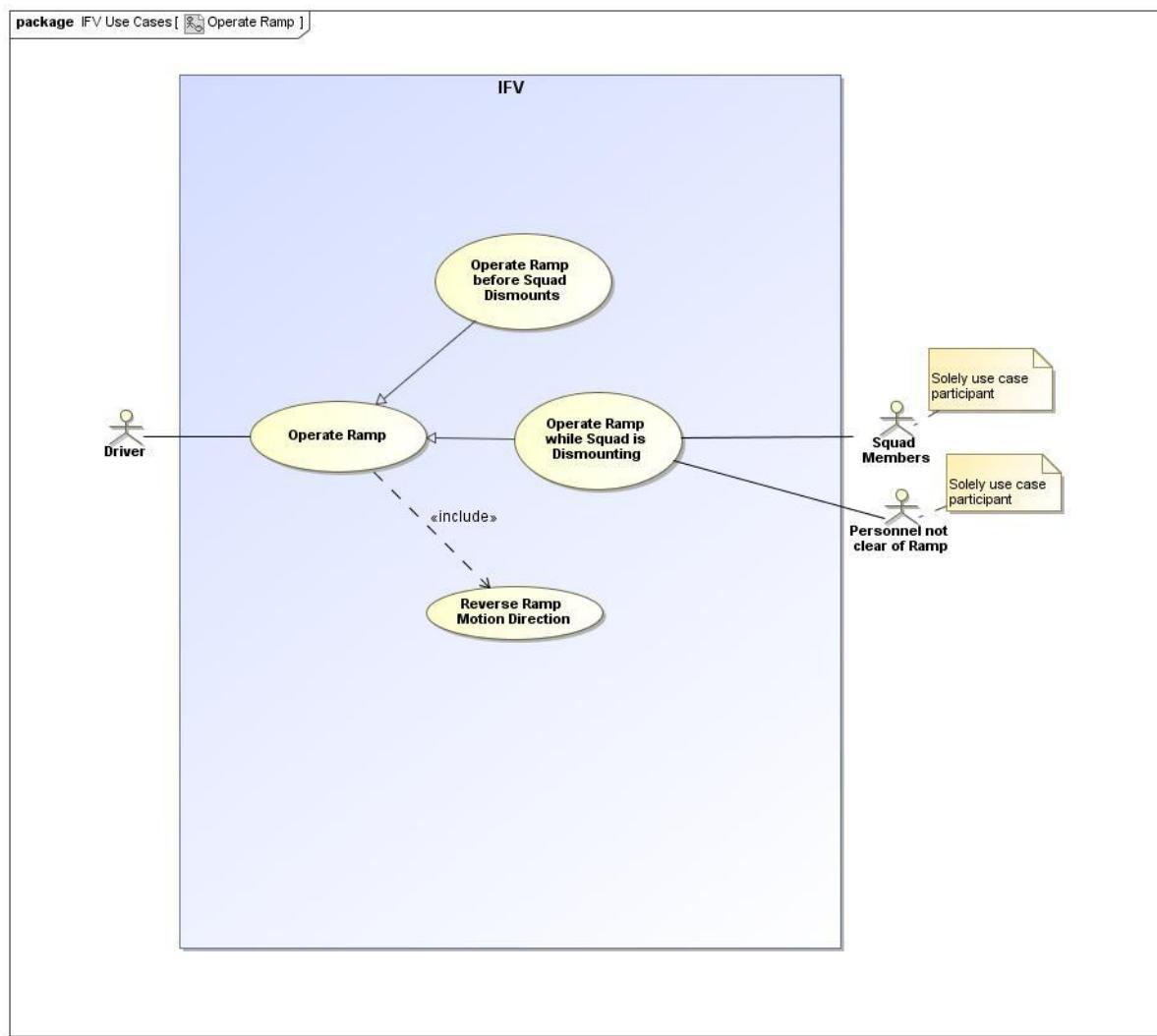
A ramp is an inclined vehicle egress/ingress pathway that connects the infantry fighting vehicle squad/payload compartment with the ground surface. Ramps and hatches pivot on a horizontal or near horizontal axis while doors pivot on a vertical or near vertical axis. Hatches are incorporated on the top and/or bottom of a frame/hull primarily to provide egress/ingress to a crew/operator position and for emergency exits. Doors are incorporated on the sides of vehicle hull/frame to provide access to vehicle position and/or equipment, egress/ingress, and/or upload and offload of cargo and supplies. A Ramp is incorporated into the vehicle hull/frame to provide a continuous path way from the vehicle to the ground for upload and offload of personnel, cargo, and/or supplies.

### Operate Ramp Use Cases Diagram

Initial IFV ramp assembly use case and behavioral analyses provided doctrinal operational context (e.g., dismount/mount warriors side-by-side, after a fully opened ramp). Subsequent use case and behavioral analyses (e.g., dismount/mount warriors side-by-side while the ramp is in motion) was provided to determine the operational impacts on ramp assembly design limits. Figure 7.1-16 illustrates Operate Ramp use cases that were used to develop ARRoW design and analysis tools and assist in developing an optimal fault tolerant cost effective IFV Ramp assembly design:

- A happy path/doctrinal ramp operations use case where the ramp completes lowering before the squad dismounts
- A use case where the infantry squad dismounts while the ramp is moving to determine design limits, safety qualify, or exercise bad day operations

These views also served to develop and analyze IFV ramp design alternatives for quarterly demonstrations.



**Figure 7.1-16. Example Operate IFV Ramp Use Cases Diagram**

### Example Details on Operate Ramp Use Case

Figure 7.1-17 illustrates example details on the Operate Ramp use case for consideration in detailed design activities for the Ramp assembly. Example details on the Operate Ramp use case includes:

- Stopping and parking the vehicle before ramp operations
- Observing clearance to the rear of the vehicle
- Sounding audio alarm before operating the ramp
- Turning on master power
- Removing the loads on/tension off the moving parts of the ramp,
- Releasing the ramp lock and unlocking the ramp
- Lowering and reversing the direction of the ramp motion

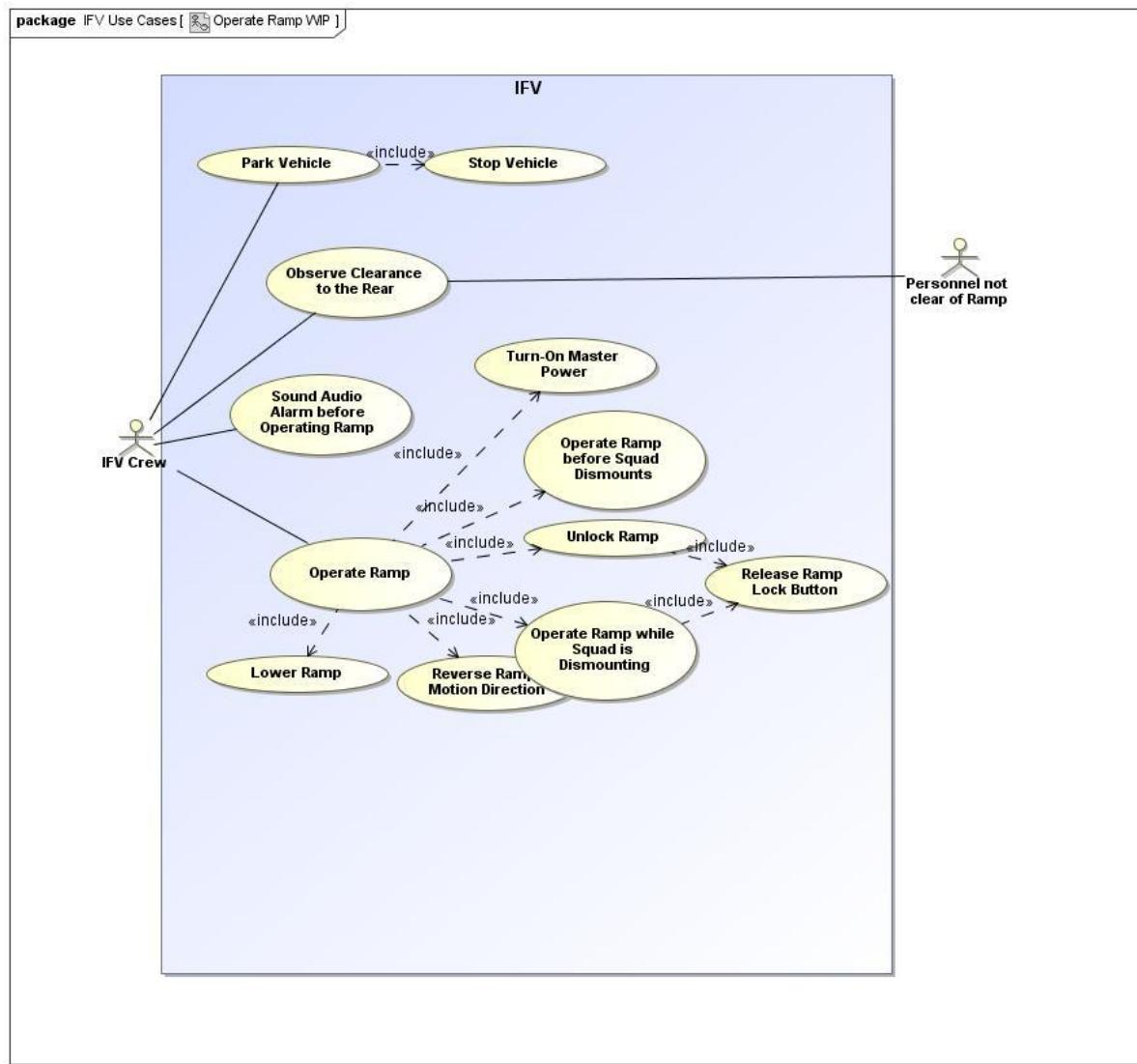
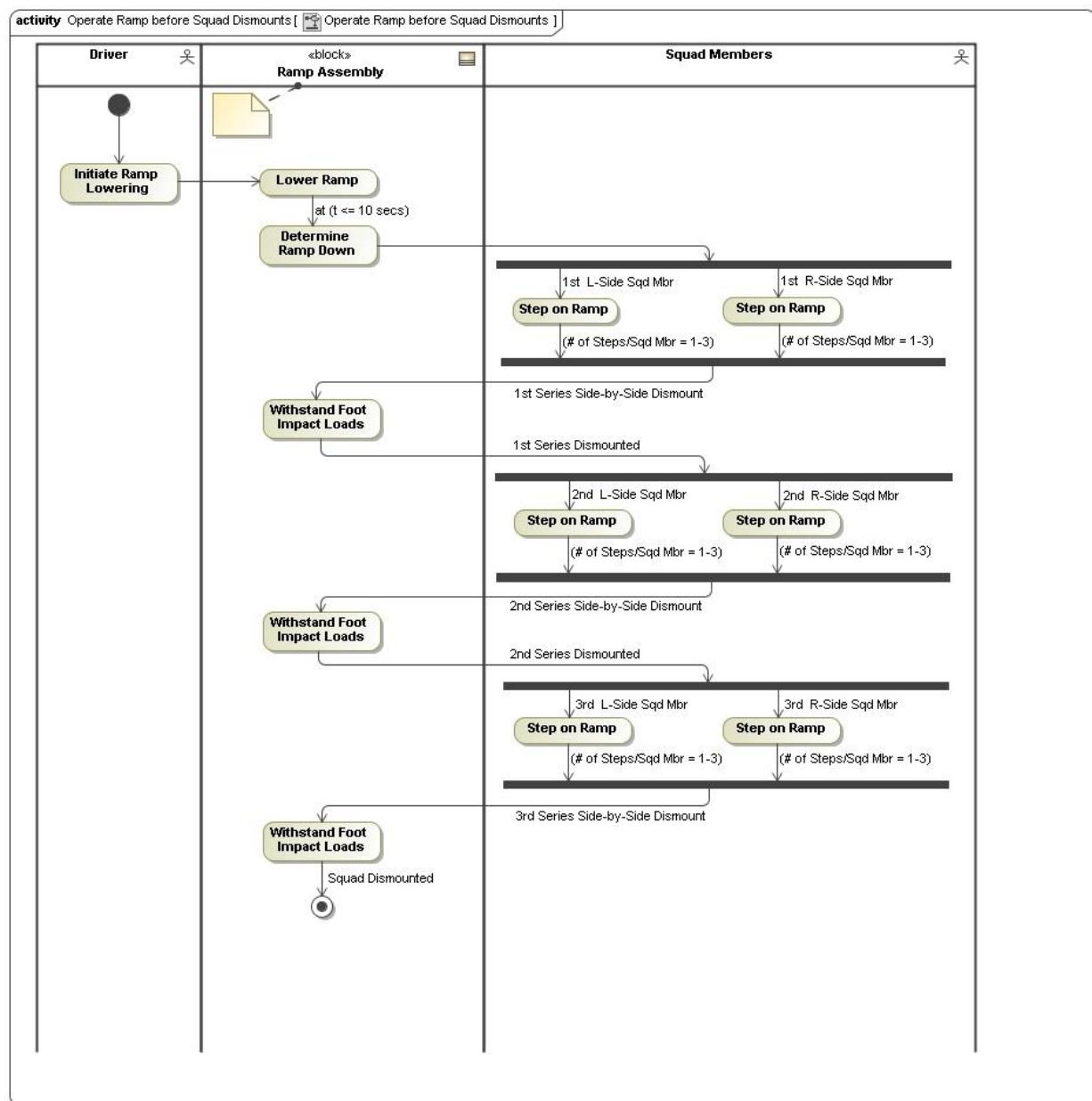
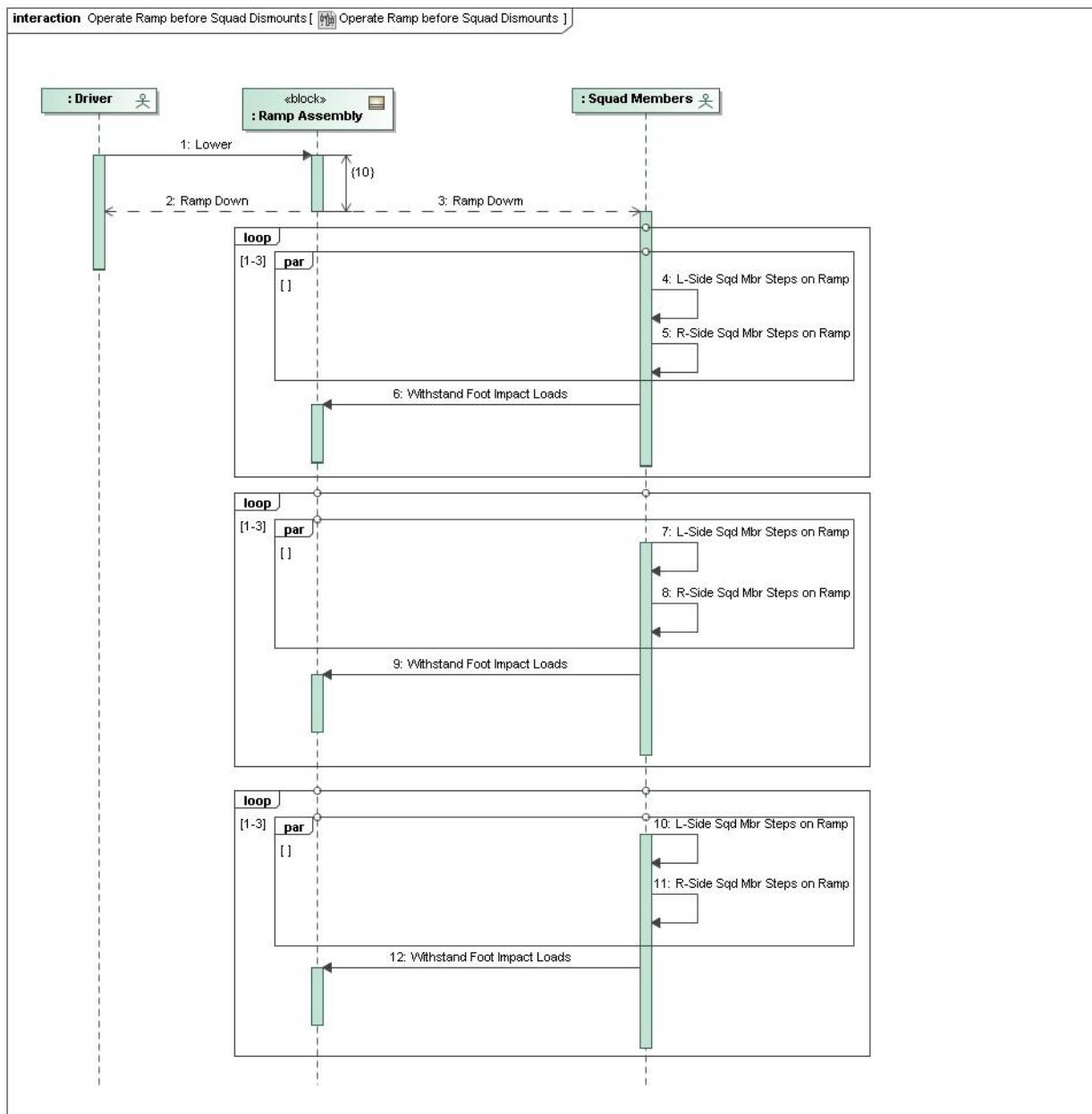


Figure 7.1-17. Example Details On Operate Ramp Use Case

### Operate Ramp before Squad Dismounts Activity and Interaction Diagrams

Figure 7.1-18 illustrates the activities and Figure 7.1-19 illustrates the interactions for the Operate Ramp before Squad Dismounts use case.

**Figure 7.1-18. Operate Ramp before Squad Dismounts Activity Diagram**



**Figure 7.1-19. Operate Ramp before Squad Dismounts Interaction Diagram**

## Operate Ramp while Squad Dismounts Activity and Interaction Diagrams

Figure 7.1-20 illustrates the activities and Figure 7.1-21 illustrates the interactions for the Operate Ramp while Squad Dismounts use case.

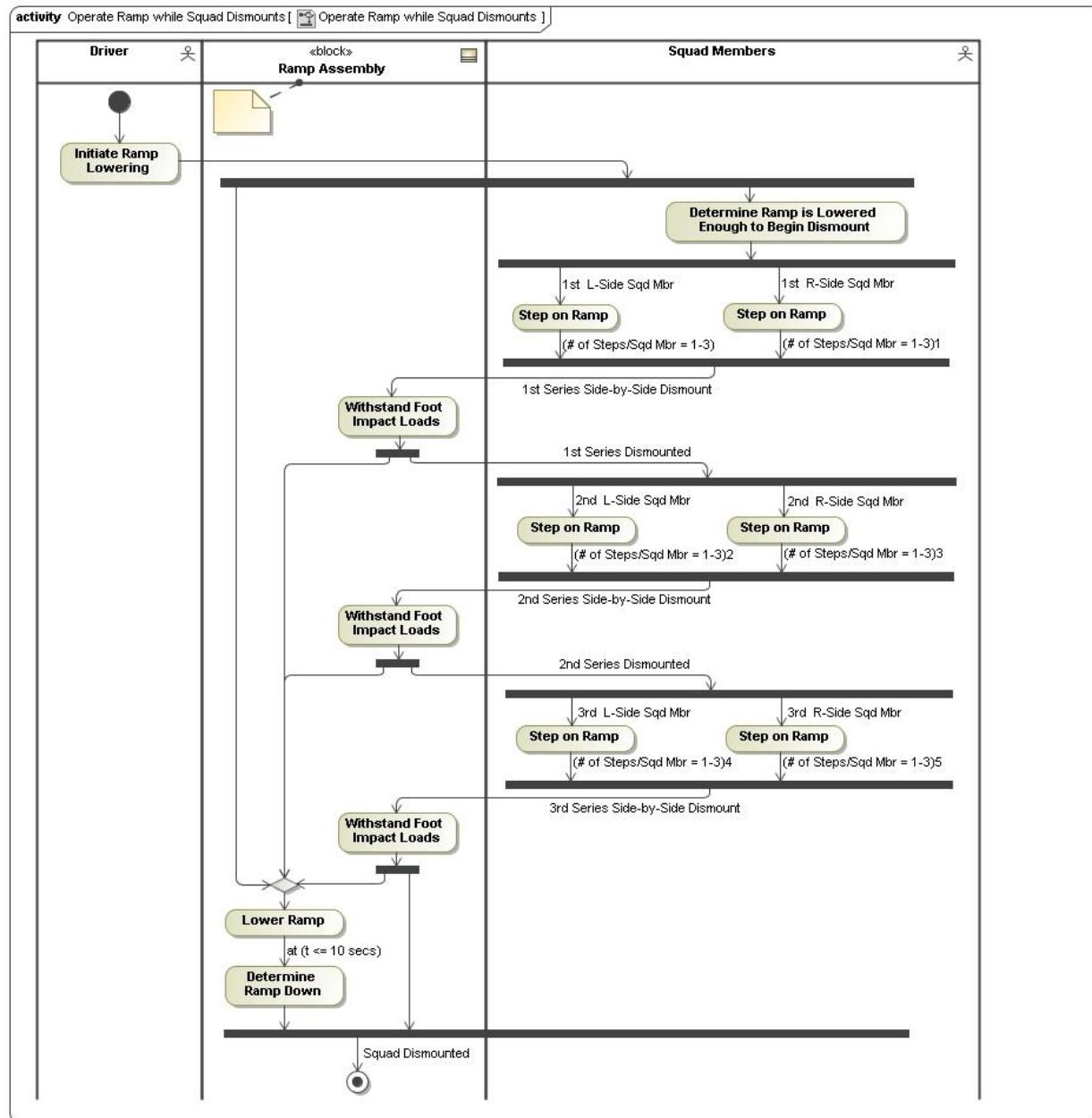
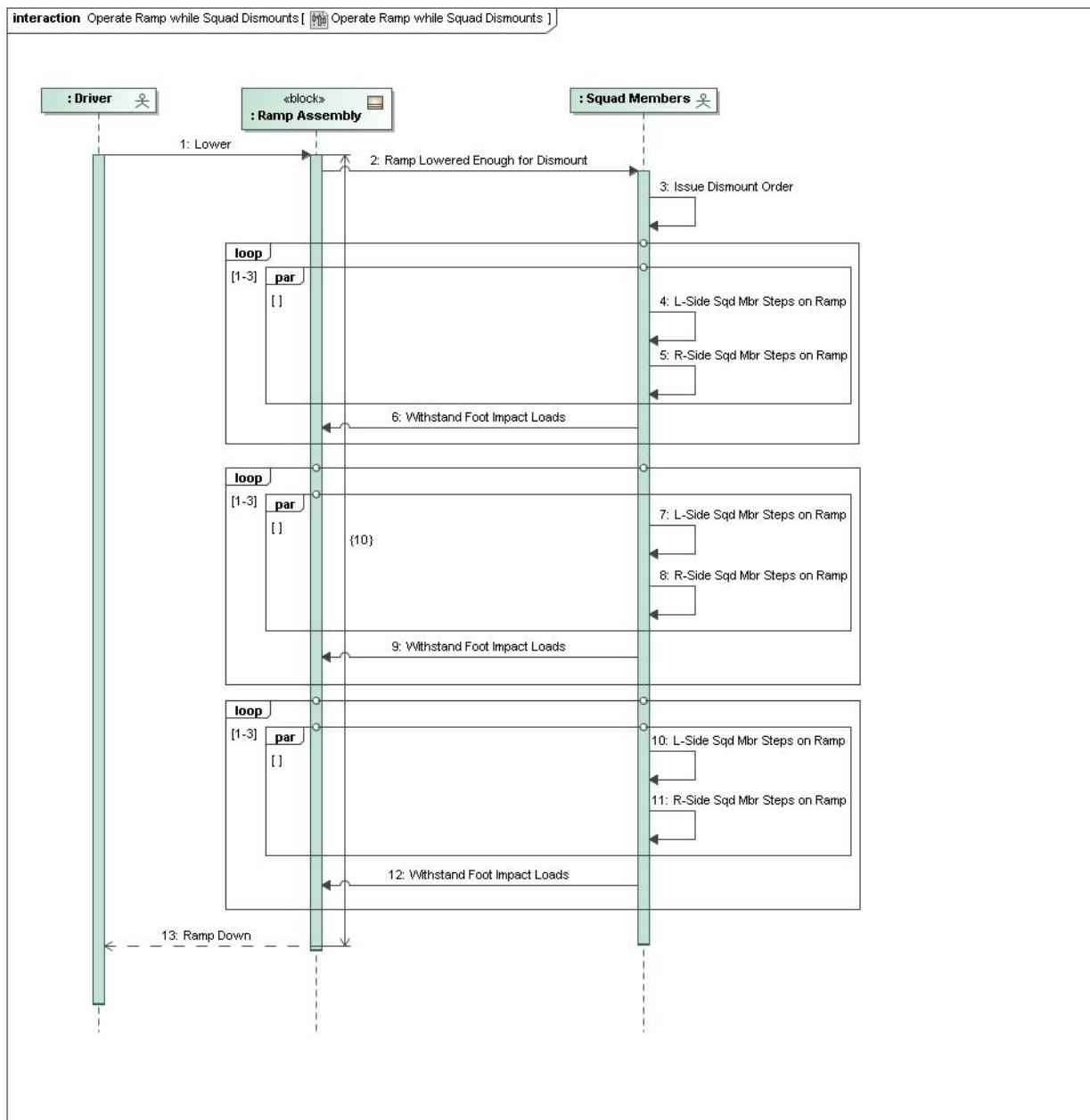


Figure 7.1-20. Operate Ramp while Squad Dismounts Activity Diagram



**Figure 7.1-21. Operate Ramp while Squad Dismounts Interaction Diagram**

### 7.1.1.9 IFV Reference Architecture SysML Model

A SysML model of an Infantry Fighting Vehicle (IFV) reference architecture was created (a reference architecture is a kind of design archetype). An automated report generated from the MagicDraw® SysML model is provided in an accompanying reference document. Its filename is “IFVRefArchStructuralReport.rtf”.

## 7.1.2 ARRoW Architecture

This section describes the ARRoW architecture developed for META phase 1b. This architecture was influenced by, and developed to be consistent with, the analytical foundation described in section 7.1.1 above.

### 7.1.2.1 Archetypes

The ARRoW architecture fundamentally depends on the pervasive use of a variety of archetypes. This section introduces the subject of archetypes and describes in detail some of the specific archetypes used in the ARRoW architecture.

An archetype is “the original pattern or model from which all things of the same kind are copied or on which they are based; a model or first form; prototype.” [DC11]

Archetypes are *foremost created for the purpose of reuse*. Because they are intended to be reused, archetypes are generally stored in the CML so that they can be easily discoverable and available for application on multiple projects. Archetypes are generally copied from the CML into the Master Model, and are then refined into a form, either manually or through automation techniques, specific to the system-of-interest being developed.

### 7.1.2.1.1 Benefits of Archetypes

A number of benefits can be realized through the use of archetypes:

1. Once an archetype is developed, significant development time can be saved through its reuse. If the archetype is properly constructed, the time and effort required to refine the archetype into an instance specific to the new application will be much less than creating it from scratch.
2. Proven concepts, best practices, and accredited approaches can be designed into the archetypes so that designs conforming to institutional quality standards can confidently be applied to new projects.
3. Using principles of model based engineering, relationships between archetype elements can be pre-allocated in the CML and these relationships can then be reused when imported into the master model. Because the analysis required to derive and define these relationships can frequently be significant, the reuse of these relationships can be a significant system analysis time-saver. Examples include:
  - a. Hierarchical “product structure” containment relationships, supporting reference architecture reuse of entire systems and subsystems.
  - b. “White-box” pre-allocation of test cases and requirements to design components, supporting automated verification of designs.
  - c. Groupings of requirements (requirement sets) related to MIL-STDs, System Performance Specifications, system capabilities, common design constraints, best practice design standards, etc.
  - d. Programmatic performance metrics linked to both producers and consumers of the data.

4. Archetypes can provide a base architecture and associated templates that are conducive to automation techniques. Examples include:
  - a. Assumption-Guarantee Contracts (Requirement Archetypes)
  - b. Requirement Specification Languages (RSLs) (Requirement Archetypes)
  - c. Natural Language Interpreters (Requirement Archetypes)
  - d. Various Reasoners (Test Case, Analysis Archetypes)

### **7.1.2.1.2 Requirements and Requirement Archetypes**

Requirements have been traditionally expressed in the form of natural language text, and have historically been delivered to developers of systems in the form of specification documents that are comprised of multiple requirements.

In recent years, advances in requirement management tools have supported the treatment of requirements as records in a central relational database. This approach has facilitated improved methods of managing relationships between requirements and other requirements, relationships between requirements and specific elements of the design, and relationships between requirements and test procedures/test result reports. Additionally, customizable metadata can be attached to requirements using these tools. These tools frequently provide the capability to import/export to electronic document formats such as MS Word® for compatibility with legacy methods of managing requirements. Examples of such tools include IBM Rational® RequisitePro® [IBM11b], IBM® Rational® DOORS® [IBM11a], and Cameo™ Requirements+ [MD11].

SysML provides support for documenting requirements, their associated relationships, and customizable metadata as well. Although today's SysML authoring tools tend to be somewhat primitive in terms of requirements authoring and management capability, the SysML language itself provides the underlying essential semantics needed to describe requirements and their relationships with the system of interest. We have chosen to use SysML as the common language for documenting requirements within the ARRoW environment because of its standardization and ubiquity. Many of today's SysML authoring tools provide plug-ins and/or data exchange capability with multiple high-end requirements management tools, so the best of both worlds (standard underlying model and powerful requirements management) can be achieved using SysML.

When we refer to the “ingestion” of requirements into ARRoW, we mean the process of

- Importing the source requirements into the ARRoW master model
- If necessary, translating the requirements from their native format into the SysML language.

After requirements are ingested, they must be processed (“digested”) to be usable in the ARRoW environment. This involves

- Binding the requirements to the appropriate master model elements
- Associating and updating any desired metadata with the requirements
- If appropriate, translating the text of the requirements into a form more suitable to being processed within the ARRoW environment

The source requirements that are ingested and processed are generally associated with the system level of the system of interest rather than lower order subsystems and components of the system. The ultimate manifestation of the ARRoW concept would ideally eliminate the

need for all but system level requirements, instead supporting only a hierarchy of test cases against the design that culminate in verification of the system requirements. In fact, there is nothing in the ARRoW concept that depends on lower level requirements existing or being verified, *per se*. However, there will likely be needs outside of the ARRoW environment that will impose a required capability of ARRoW to support a hierachal structure of requirements and automated verification techniques below the system level. Some of these needs include:

- Not all elements of the system, at least initially, will lend their complete design to ARRoW and/or their implementation by iFAB. Examples include the development of needed software components (SW development is not yet a part of ARRoW), new technology requiring hands-on experimentation with hardware, safety critical systems where bodies within the government may insist on development and test procedures outside the scope of ARRoW, or simply dependency on suppliers not yet integrated into the ARRoW/iFAB environment. In each of these cases, it may be necessary to develop requirements (natural language or otherwise) that are formally imposed on an organization for development of a portion of the system of interest. These requirements will need to be derived from the system requirements.
- For the purposes of project and design management, it may be desirable to have “watch points” interspersed at lower levels within the system design that can notify the developer if certain limits are exceeded. These watch points need not be formal requirements, but can be of identical form to requirements, including the manner in which they interact with the verification mechanics of ARRoW. Examples of useful watch points might be weight budgets, cost budgets, message latencies, etc.
- Trade studies can be automated within ARRoW to compare the relative utility of alternative designs residing at arbitrary levels within the system product structure. Requirements can be used to support the analysis of multiple criteria used to assess these alternatives. Requirements in the ARRoW master model are generally verified by comparing test results to an expected value or characteristic described in the requirement. The verdict resulting from this comparison can be binary in nature (pass/fail), or it can be the output of a utility function whose input domain is a test result value and whose output is a score that grades (continuously or discretely) across a range from “no effective utility” to “maximum possible utility”. Thus, the standard mechanism for verifying requirements can be used to analyze trade study criteria as well.
- Subsystems and components will eventually reside in the CML, and should have performance specifications associated with them in some form to aid in the proper selection of elements from the CML. These specifications should be in the form of requirements to best interface with the ARRoW tool chain. These requirements may or may not be in the form of assumption-guarantee contracts, but as identified in the contracts approach, simulation processing time can be saved if the system can be analyzed based on the guaranteed performance of a system’s components instead needing to simulate the behavior of all of its components.

Since the processing of requirements (system-level or derived lower-level) can involve a significant amount of analysis and documentation effort, they are prime candidates for improvement within the development paradigm of ARRoW. To that end, we propose the concept of “Requirement Archetypes” (RAs).

RAs are patterns for requirements of like types. Requirements are refined from RAs within ARRoW. The process of refinement generally involves copying the RA from the CML, modifying the archetype pattern to make it a requirement specific to the system of interest being developed, and associating the refined requirement (either directly or indirectly) with the source requirement. RAs exist in the CML in two major forms: 1) they are pre-allocated to design components and to their corresponding test case archetypes such that the act of importing designs or design archetypes from the CML into the master model will also import the associated RAs, and 2) they are grouped into Requirement Archetype Sets (RASs) that are logical groupings of RAs that are associated with typical functions, capabilities, or constraints of systems that might be developed. Examples of grouping criteria for RASs might be sample performance specifications (composed of RAs, not requirements), RAs grouped by associated MIL-STDs, functional groupings such as heavy combat vehicle mobility RAs, etc.

The process of copying RAs into the master model involves the use of both of these forms. Candidate RASs can be imported by doing keyword searches based on source requirements' subject matter. Likewise, candidate design archetypes and their associated RAs can also be imported based on similar keyword searches. Once both of these RA forms are in the master model, they are compared as an aid to determine if there are missing source requirements or missing requirements allocated to the design.

Note that to the extent that source requirements provided by the customer conform to the existing scope and form of RASs already in the CML, the requirements analysis effort needed for the new system of interest can be greatly simplified.

Requirements and RAs are of identical form and can have similar associations with other master model elements, but generally differ in the values assigned to their metadata (although in some cases, RAs may exist in the CML that are fully refined to requirements). An example of a requirement archetype is shown in Figure 7.1-22.

«metaExtendedRequirement» SizeRqmtArchetype	
Id = "1"	
IsMissionCritical = true	
IsSafetyCritical = false	
IsTbdTbr = false	
IsUserTaskDependent = false	
IsValidated = false	
IsVerified = false	
Keywords = ""	
Priority/Utility = ""	
ProgrammaticPurpose = CustomerReviewable	
risk = Medium	
SecurityClassification = Unclassified	
State = WIP	
Text = "The <subject> <Dimension type> shall be not greater than height <X.X> m, width <Y.Y> m, and length <Z.Z> m."	
Type = Performance	
verifyMethod = Analysis	

**Figure 7.1-22. Sample Requirement Archetype with Metadata**

An RA is a SysML «requirement» model element from which new stereotypes can be derived to provide additional metadata that may be of use in managing the requirement. Each SysML «requirement» minimally has an “Id” and a “Text” field. The Id is a project unique identifier for the requirement. The Text is the actual body of the requirement.

RAs are patterns for requirements. They can include additional metadata for suggested use (such as that shown in Figure 7.1-22), they can be pre-allocated to elements in the CML (including other RAs, design components, and test case archetypes), and the value of their Text field can also be of a form that is a pattern for general use.

The nature of the RA or requirement text can take on many forms based on the type of requirement involved, including references to non-textual model elements. Just a few examples include:

- References to Use Case or Sequence Diagram based required behavior
- Assumption-Guarantee Contracts boilerplate
- Formal Requirements Specification Language (RSL) boilerplate
- Languages facilitating Qualitative/Quantitative Reasoners
- Three-dimensional Physical Envelope Constraints
- Operational requirements stated in a way that invites proper construction of appropriate dynamic, multi-physics based Operational Virtual Prototype simulations
- Performance Requirements that lend themselves to parametric descriptions

Examples of how this last bullet can be provided in archetype form and then refined to specific infantry fighting vehicle requirements is shown in Table 7.1-25.

**Table 7.1-25. Sample Requirement Archetype Text**

Topic	Archetype	IFV Refinement Examples
<b>Mobility Requirement Archetype Set</b>		
Dash Speed	The <subject> shall accelerate from <X1> to <X2> kph in not more than <Y> sec on <Terrain type> at a <Z> degree slope	The <IFV> shall accelerate from <0> to <80> kph in not more than <25> secs on <primary roads> at a <0> degree slope.
		The <IFV> shall accelerate from <0> to <45> kph in not more than <25> secs on <cross-country terrain> at a <0> degree slope.
Speeds	The <subject> shall attain <Direction> speed of not less than <X> kph on <Terrain> at a <Z> degree slope	The <IFV> shall attain <forward> speed of not less than <80> kph on <primary roads> at a <0> degree slope.
		The <IFV> shall attain <rearward> speed of not less than <16> kph on <primary roads> at a <0> degree slope.
		The <IFV> shall attain <forward> speed of not less than <45> kph on <cross-country terrain> at a <0> degree slope.
		The <IFV> shall attain <rearward> speed of not less than <5> kph on <cross-country terrain> at a <0> degree slope.

Topic	Archetype	IFV Refinement Examples
		The <IFV> shall <b>attain</b> <objective forward> speed of not less than <80> kph on <primary roads> at a <0> degree slope.
		The <IFV> shall <b>attain</b> <threshold forward> speed of not less than <30> kph on <primary roads> at a <0> degree slope.
	The <subject> shall <b>attain</b> <Direction> speed within and including the endpoints of the range from <X1> to <X2> kph on <Terrain> at a <Z> degree slope	The <IFV> shall <b>attain</b> <forward> speed within and including the endpoints of the range from <30> to <80> kph on <primary roads> at a <0> degree slope.
		The <IFV> shall <b>attain</b> <rearward> speed within and including the endpoints of the range from <8> to <16> kph on <primary roads> at a <0> degree slope.
		The <IFV> shall <b>attain</b> <forward> speed within and including the endpoints of the range from <30> to <45> kph on <cross-country terrain> at a <0> degree slope.
		The <IFV> shall <b>attain</b> <rearward> speed within and including the endpoints of the range from <5> to <8> kph on <cross-country terrain> at a <0> degree slope.
Turning Radius	The <subject> shall <b>turn</b> <Bearing1> or <Bearing2> <X> degrees within <Y.Y> times the vehicle diagonal length <Z.Z> m	The <IFV> shall <b>turn</b> <left> or <right> <360> degrees within <1.0> times the vehicle diagonal length <7.5> m.
Energy Efficiency	The <subject> shall <b>consume</b> not greater than <X.XX> KPL at <Y> kph on <Terrain Type> at a <Z> degree slope	The <IFV> shall <b>consume</b> not greater than <0.32-0.95> KPL at <48> kph on <primary roads> at a <0> degree slope.
Climb Obstacle	The <subject> shall <b>climb</b> obstacles at a height not less than <X.X> m	The <IFV> shall <b>climb</b> obstacles at a height not less than <1.5> m.
Cross Gap	The <subject> <b>cross</b> trenches at a width not less than <X.X> m	The <IFV> shall <b>cross</b> trenches at a width not less than <1.5> m.
Fording	The <subject> shall <b>ford</b> water at a depth not less than <X.X> m.	The <IFV> shall <b>ford</b> water at a depth not less than <1.5> m.
Cruising Range	The <subject> shall <b>travel</b> on internally carried fuel for no less than <X> km at <Y> kph average sustained speed on <Terrain type>.	The <IFV> shall <b>travel</b> on internally carried fuel for no less than <480> km at <45> kph average sustained speed on <primary roads>.
<b>Personnel Capacity Requirement Archetype Set</b>		

Topic	Archetype	IFV Refinement Examples
<b>Personnel Capacity</b>	The <subject> shall <b>be sized</b> for a <Personnel Organization type> of <X> soldiers.	The <IFV> shall <b>be sized</b> for a <crew> of <2> soldiers.
		The <IFV> shall <b>be sized</b> for a <crew> of <3> soldiers.
		The <IFV> shall <b>be sized</b> for a <squad> of <6> soldiers.
		The <IFV> shall <b>be sized</b> for a <squad> of <9> soldiers.
<b>Physical Characteristics Requirement Archetype Set</b>		
<b>Weight</b>	The <subject> <Weight type> shall <b>be</b> not greater than <X.X> tonnes	The <IFV> <air shipping weight> shall <b>be</b> not greater than <45.0> tonnes.
		The <IFV> <curb weight> shall <b>be</b> not greater than <50.0> tonnes.
		The <IFV> <maximum combat weight> shall <b>be</b> not greater than <55.5> tonnes.
		The <IFV> <transport dimensions> shall <b>be</b> not greater than height <3.0> m, width <2.7> m, and length <6.3> m.
<b>Dimensions</b>	The <subject> <Dimension type> shall be not greater than height <X.X> m, width <Y.Y> m, and length <Z.Z> m	The <IFV> <operational dimensions> shall <b>be</b> not greater than height <4.0> m, width <3.3> m, and length <6.8> m.
<b>Ground Clearance</b>	The <subject> <Orientation> ground clearance type shall be not less than <X.X> m	The <IFV> <front> ground clearance shall <b>be</b> not less than <0.45> m.
		The <IFV> <rear> ground clearance shall <b>be</b> no less than <0.4> m.
<b>Ramp Angle Clearance</b>	The <subject> <Ramp Angle Clearance type> shall be not less than <X.X> degrees	The <IFV> <Angle of Approach> shall <b>be</b> not less than <75> degrees.
		The <IFV> <Angle of Departure> shall <b>be</b> not less than <50> degrees.
<b>Transportability Requirement Archetype Set</b>		
<b>Airlift Cargo Dimension Limits</b>	The <subject> shall <b>comply with</b> <Airlift asset type> cargo transportability dimension limits of height <X.X> m at width <Y.Y> m, width <Y.Y> m, and length <Z.Z> m.	The <IFV> shall <b>comply with</b> <C-17> cargo transportability dimension limits of height <3.6> m at width <5.2> m, width <5.2> m, and length <19.9> m.

Topic	Archetype	IFV Refinement Examples
		The <IFV> shall <b>comply with</b> <C-17> cargo transportability dimension limits of height <3.9> m at width <4.3> m, width <4.3> m, and length <19.9> m.
		The <IFV> shall <b>comply with</b> <C-5> cargo transportability dimension limits of height <2.7> m at <5.4> m width, width <5.4> m, and length <36.9> m.
		The <IFV> shall <b>comply with</b> <C-5> cargo transportability dimension limits of height <3.9> m at <3.6> m width, width <3.6> m, and length <36.9> m.
		The <IFV> shall <b>comply with</b> <C-130> cargo transportability dimension limits of height <2.6> m at width <2.7> m, width <2.7> m, and length <16.8> m.
<b>Raillift Cargo Dimension Limits</b>	The <subject> shall comply with <Raillift asset type> cargo transportability dimension limits of height <X.X> m at width <Y.Y> m, width <Y.Y> m, and length <Z.Z> m.	The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <4.37> m at width <1.26> m, width <1.26> m, and length <15.30> m.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <4.25> m at width <1.85> m, width <1.85> m, and length <15.30> m.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <4.15> m at width <2.17> m, width <2.17> m, and length <15.30> m.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <3.91> m at width <2.60> m, width <2.60> m, and length <15.30> m.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <3.49> m at width <3.04> m, width <3.04> m, and length <15.30> m.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability dimension limits of height <3.25> m at width <3.15> m, width <3.15> m, and length <15.30> m.

Topic	Archetype	IFV Refinement Examples
<b>Militarylift Cargo Weight Limit</b>	The <subject> shall <b>comply with</b> <Militarylift> cargo transportability weight limit of <X.X> tones.	The <IFV> shall <b>comply with</b> <C-17> cargo transportability weight limit of <58.9> tonnes.
		The <IFV> shall <b>comply with</b> <C-5> cargo transportability weight limit of <80.7> tonnes.
		The <IFV> shall <b>comply with</b> <C-130> cargo transportability weight limit of <19.5> tonnes.
		The <IFV> shall <b>comply with</b> <NATO Rail M Envelope> cargo transportability weight limit of <70> tonnes.
<b>Survivability Requirement Archetype Set</b>		
<b>Threat Munition Protection</b>	The <subject> shall <b>protect against</b> <Threat type> munitions.	The <IFV> shall <b>protect against</b> <14.5 mm machine gun> munitions.
		The <IFV> shall <b>protect against</b> <RPG-7> munitions.
<b>Vehicle Threat Protection</b>	The <subject> shall <b>provide</b> <Vehicle Amount type> protection against <Threat type> munitions	The <IFV> shall <b>provide</b> <360 degree> protection against <Threat type> munitions.
		The <IFV> shall <b>provide</b> <crew compartment> protection against <Threat type> munitions.
		The <IFV> shall <b>provide</b> <engine compartment> protection against <Threat type> munitions.
		The <IFV> shall <b>provide</b> <weapon compartment> protection against <Threat type> munitions.
		The <IFV> shall <b>provide</b> <squad compartment> protection against <Threat type> munitions.

Figure 7.1-23 and Figure 7.1-24 provide examples of how two additional classes of requirements might be categorized and how they might share a common verification approach within each category. Although we have not yet constructed requirement archetypes for these categories, it can be seen that the dramatically different forms of respective verification might invite RA patterns that can be optimally expressed to relate to the corresponding most appropriate test cases.

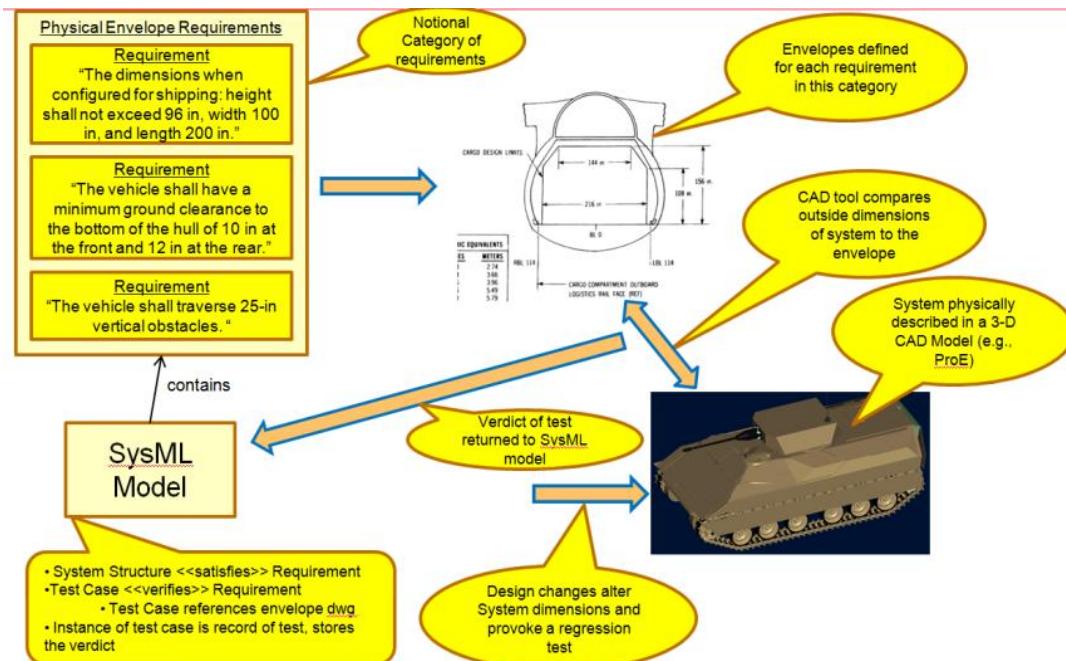


Figure 7.1-23. Physical Envelope Verification

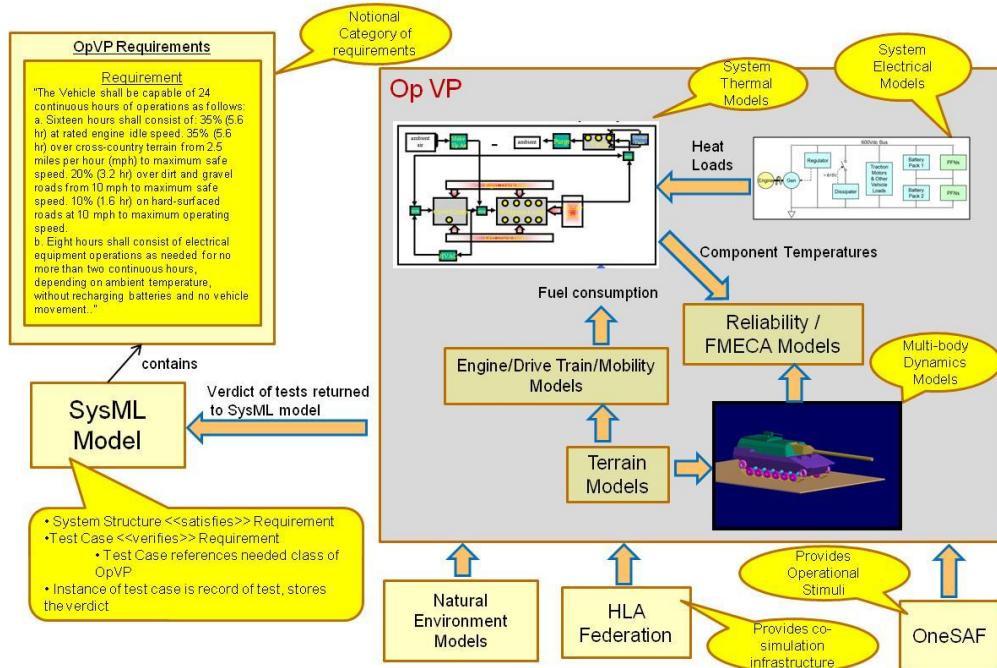


Figure 7.1-24. Operational Virtual Prototype Verification

### 7.1.2.1.3 Test Cases and Test Case Archetypes

Refer to section 7.1.2.6 for additional information relating to the ARRoW entities that interact with Test Cases and Test Case Archetypes, including diagrams related to this topic.

Within the scope of ARRoW, a Test Case (TC) is defined to be an executable that configures and orchestrates the test of, and stimulates the inputs of a Design Component (DC) for the purpose of verifying one or more requirements levied against that DC or a product structure parent of that DC.

The DC has a corresponding Component Model (CM) that models the behavior or physical properties of its detailed design. This CM reacts to the stimuli generated by the TC, and produces output that is compared to a required output and/or routed to some other element within the Master Model (MM) for further processing.

A TC does not have visibility into the internal structure or behavior of the DC, including its associated CMs. In other words, a TC has a “black box” interface to the DC. However, a TC does impose a required interface standard on the DC for all inputs and outputs related to the test. This interface standard might include both run time data exchange interfaces as well as configuration/setup and status reporting interfaces.

TCs are frequently refined from Test Case Archetypes (TCAs). Within the CML, TCAs are pre-allocated to design archetypes. TCAs might be composed of pseudo-code, parameterized functions/services expressed in a general purpose language, Simulink® blocks, SysML parametric diagrams, or any form of expression that can provide a template for the logic of a TC. A TC may be a modification of a TCA, highly refined within the master model to make it an executable for supporting test of a specific DC, or it may be the unmodified TCA itself if that TCA can appropriately be applied to the run time test of the DC. For example, the CML will contain, in addition to design archetypes, design components associated with specific model numbers of equipment. The TCAs for such components may be indistinguishable from TCs.

In a top-down design process, it frequently is necessary to specify requirements for a design component before that component is actually designed. In fact, new components are frequently designed pursuant to the requirements imposed on them. In general, a black box test of a component for the purpose of verifying a requirement can be constructed if all relevant external interfaces of that component are known and well defined. This standardized test mechanism can thus be reused against many alternative design solutions and revisions without needing to change the structure/logic of the TC. Additionally, proprietary information related to the component design need not be exposed as long as the external interfaces are openly documented.

Thus, TCs and TCAs are generally expected to be open-source, whereas CMs may be proprietary and might be published as an executable without exposing the source code. Both TCAs and CMs are expected to be published in the CML.

Many requirements will have a recurrent set of environmental constraint (context) requirements that apply during their verification test as *test conditions*. For example, ambient temperature, altitude, and road surface properties will normally influence mobility performance, so the verification test of, say an acceleration requirement of a vehicle, would be performed under prescribed test conditions of the required ambient operational temperature range, maximum required altitude, and specified road condition model.

When a TC is constructed, it must account for all the relevant test conditions imposed on the test based on the relevant context requirements that will typically apply to the associated design component. TCAs provide a mechanism by which this analysis can be reused. TCAs can have references to multiple Requirement Archetypes (RAs) that act in the role of test conditions for the test of the primary requirement(s) being verified.

There is a logical two-dimensional matrix that relates the set of all context requirements imposed on a system and the set of all requirements whose verification is dependent on test conditions specified by context requirements. Any particular context requirement is verified only after every requirement dependent on that context is verified. The AIDE will support verification of context requirements by keeping track of which context-dependent requirements have been verified against the current system of interest design..

#### 7.1.2.2 ARRoW Context Diagram

"ARRoW" is the BAE Systems META team's solution to the META problem. ARRoW includes tools (both developmental as well as non-developmental), but also includes all architectural templates, interface standards, libraries, etc. necessary to support development of new systems. These new systems are herein referred to as "System(s) of Interest" (SoI).

The "ARRoW Integrated Development Environment" ("ARRoW IDE" or "AIDE") represents the entire deployed runtime environment that supports development of a System of Interest, as well as any documentation supporting the AIDE user for the use and maintenance of the AIDE. For the purposes of this analysis, "ARRoW" and "ARRoW IDE" can be used interchangeably,

The AIDE is an "integrated" environment, meaning that the developer of ARRoW has architected and verified that the ARRoW tool chain and data will function together as a system. Additionally, subsequent to deployment, either the initial developer or a subsequent authority assures the AIDE continues as an integrated solution, including the integrated functionality of non-developmental, open-source, and freeware tools.

The AIDE users will include SoI end users, the SoI acquisition community, SoI developers, regulatory authorities, and component vendors.

A context diagram depicting major actors that interface with the AIDE is shown in Figure 7.1-25. Refer to section 7.1.1.4.1 above for descriptions of these actors.

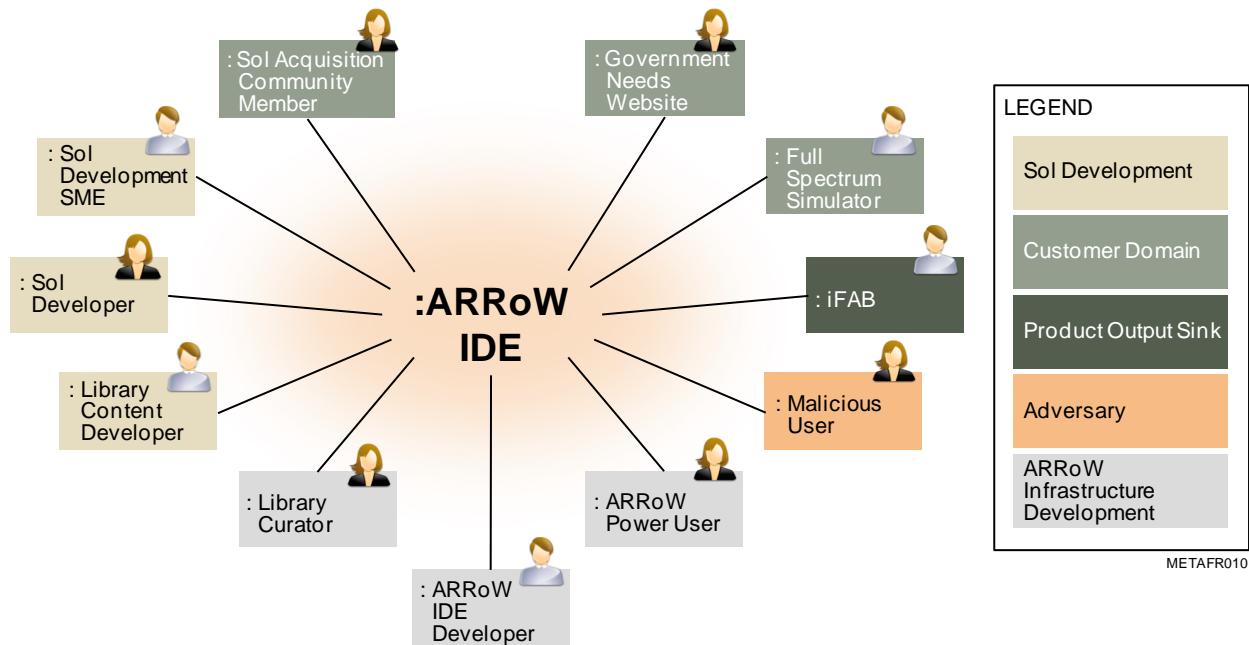
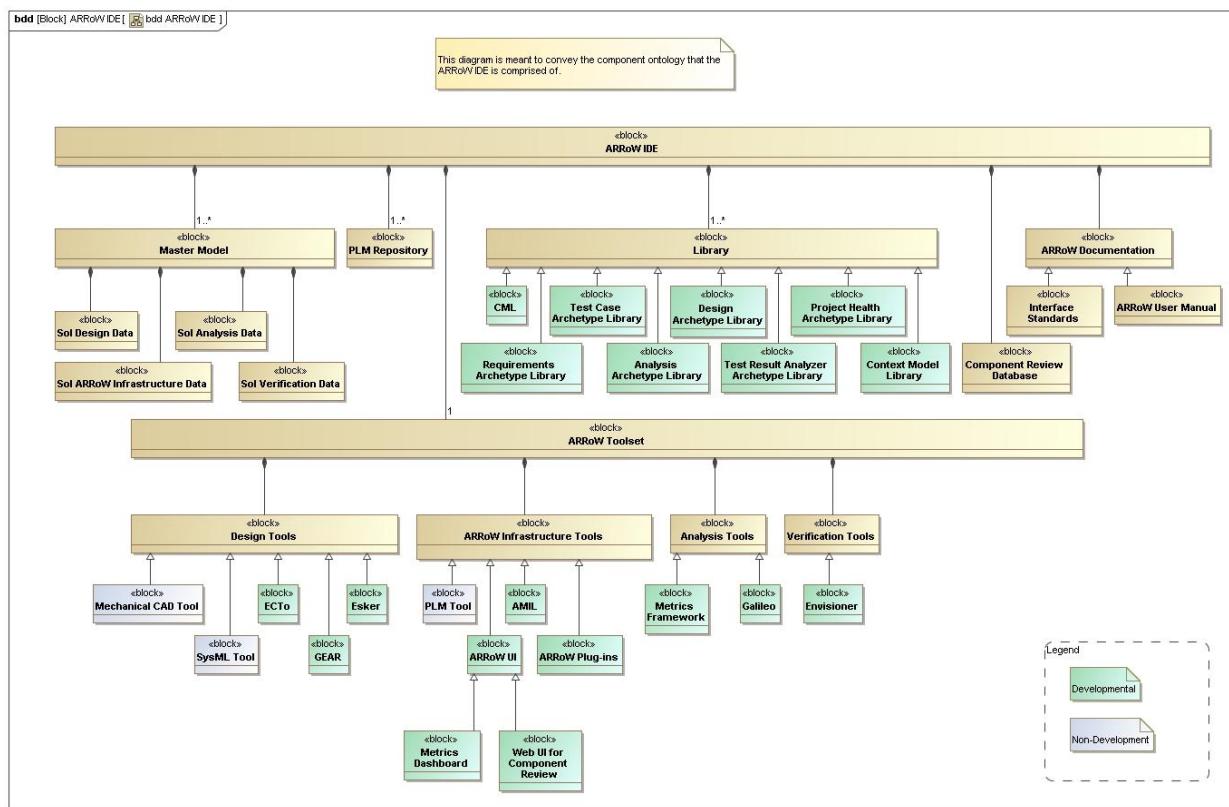


Figure 7.1-25. AIDE Context Diagram

### 7.1.2.3 AIDE Entities

Figure 7.1-26 is a SysML block definition diagram that provides an overview of the major components and tools that comprise the AIDE. Connectors with diamonds indicate that the block on the diamond side contains the other block. Connectors with arrow heads indicate that the block on the arrow head side is a generalization of the other block.

**Figure 7.1-26. AIDE**

A high level overview of Figure 7.1-26 follows.

### Master Model

The ARRoW development paradigm embraces principles of Model Based Engineering (MBE) and Model Based Systems Engineering (MBSE) whereby, to the maximum extent practical, all aspects of the SoI design and its development process are contained in an underlying unified model that can be accessed and manipulated by a diverse set of development tools. In ARRoW, this is called the Master Model (MM).

The Master Model (MM) contains the system being developed beginning with requirements and culminating in the handoff to iFab. This includes all data associated with the specific design, including requirements, metrics, geometry, software, controls, links to specific entities within the component model library, and all data normally contained in a Technical Data Package for a military system. All components of the toolchain use and store data within the MM via AMIL.

### PLM Repository

Because development of a SoI will normally involve multiple developers working concurrently, the AIDE will need to support this. The evolution and configuration of the MM will need to be managed to the same rigor that data is managed on complex projects today. Thus, we anticipate that the MM will reside in a repository managed by a sophisticated Product Lifecycle Management (PLM) toolset.

### Library

The ARRoW development process generally entails the import of model elements from various libraries into the MM, and then a new SoI is developed within the MM by modifying some of the imported elements, creating new elements, and creating new relationships between elements.

In the context of the AIDE, we have identified a number of archetype categories that can potentially contribute to more efficient development of new systems. We have logically organized these archetype categories as if they would each have their own library structure and custodian. Whether each of these notional libraries is actually implemented as part of the ARRoW solution is left as a design decision for future consideration.

### ARRoW Documentation

Critical to the success of any IDE is a set of quality documentation that describes its use and capabilities. A user manual is an example of such documentation. Additionally, we anticipate that certain data exchange standards will be required, and will pertain to both tool interfaces and library interfaces.

### ARRoW Toolset

The ARRoW Toolset is a set of software tools to:

- Aid in SoI design, analysis, and verification
- Provide an infrastructure for the AIDE

Descriptions of the blocks shown in Figure 7.1-26, alphabetically sorted by block name, are provided in Table 7.1-26. The above diagram, and its constituent blocks can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.2 Architecture Development”, with the description text in the documentation metadata field associated with each block.

**Table 7.1-26. AIDE Block Descriptions**

Block Name	Description
AMIL	ARRoW Model Interconnection Language (AMIL) is a tool used to represent the links or the network of those models of the System of Interest (SoI) that ARRoW works with either directly or as a proxy.
Analysis Tools	Analysis Tools is a set of application programs that aid in the development of project and product metrics information that includes: <ul style="list-style-type: none"> <li>• Project health</li> <li>• System and cost effectiveness</li> <li>• Problem domain understanding</li> <li>• Design maturity and health</li> </ul>

Block Name	Description
Analysis Archetype Library	<p>The Analysis Archetype Library is a repository for templates of analysis types (categories). It is expected that the analysis archetype templates will include:</p> <ul style="list-style-type: none"> <li>• System Analysis</li> <li>• Operational Analysis</li> <li>• Requirement Analysis</li> <li>• Design Analysis</li> <li>• Design maturity and health</li> <li>• Support Analysis</li> </ul>
ARRoW Documentation	<p>AIDE documentation set is a collection of documents that:</p> <ul style="list-style-type: none"> <li>• Aid users in the operations and maintenance of the AIDE</li> <li>• Identify and define internal and external interfaces of the AIDE</li> </ul>
AIDE	<p>The Adaptive, Reflective, Robust Workflow (ARRoW) Integrated Development Environment (IDE) includes the following elements:</p> <ul style="list-style-type: none"> <li>• Set of master models to represent the data for the pertinent System of Interest (SoI) being developed</li> <li>• A Product Lifecycle Management (PLM) repository for all product information relating from concept through design and release for production, to operations &amp; support and disposal</li> <li>• An ARRoW toolset to aid in SoI design, analysis, and verification, and provide an infrastructure for the integrated development environment</li> <li>• A library set consisting of product component models, project &amp; product development archetypes, and product context models</li> <li>• A database of relevant stakeholder review comments on components in development or deployed</li> <li>• -AIDE documentation set for users, maintainers, and interface standards</li> </ul>
ARRoW Infrastructure Tools	<p>Adaptive, Reflective, Robust Workflow (ARRoW) Infrastructure Tools are application programs that provide for and support the foundation of the ARRoW integrated development environment (IDE).</p>
ARRoW Plug-ins	<p>Adaptive Reflective Robust Workflow (ARRoW) Plug-ins are application programs that add capabilities to an existing application programs or to enable customization of application functionality so that existing application programs can be utilized in the AIDE.</p>
ARRoW Toolset	<p>The Adaptive, Reflective, Robust Workflow (ARRoW) Toolset is a set of tools to:</p> <ul style="list-style-type: none"> <li>• Aid in SoI design, analysis, and verification</li> <li>• Provide an infrastructure for ARRoW integrated development environment (IDE)</li> </ul>
ARRoW UI	<p>The ARRoW User Interface (UI) includes all needed UIs to support the set of ARRoW users.</p>
ARRoW User Manual	<p>ARRoW User Manual is a collection of documents that aid users in the operations and maintenance of the AIDE.</p>

Block Name	Description
CML	<p>A Component Model Library (CML) is a collection of component models that represent products at various lifecycle stages:</p> <ul style="list-style-type: none"> <li>• Technology Development</li> <li>• Engineering &amp; Manufacturing Development</li> <li>• Production &amp; Deployment</li> <li>• Operations &amp; Support</li> </ul>
Component Review Database	<p>Component Review Database contains review comments provided by end users, the acquisition community, developers, etc. against any entity that is either published in a library or physically in use in the real world.</p>
Context Model Library	<p>The Context Model Library includes Aberdeen Proving Grounds (APG) Course models, Weather Models, satellite constellation models, threat models, lethal effects models, OneSAF, etc.</p>
Design Archetype Library	<p>The Design Archetype Library is a repository for templates of design types (categories). It is expected that the design archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Product Breakdown Structure (PBS) Items</li> <li>• Component Items</li> <li>• Design Domains</li> <li>• New Technology Based Items</li> </ul>
Design Tools	<p>Design tools are a set of product development application programs that enable users to develop a System-of-Interest (SoI).</p>
ECTo	<p>ECTo (Early Concepting Tool) is a tool used for vehicle level concept and prototype development.</p>
Envisioner	<p>Envisioner is a qualitative simulation tool used to aid in early and often confirmation that the specified requirements are fulfilled by the System-of-Interest (SoI).</p>
ESKER	<p>ESKER (Expert-System Knowledgebase Evaluation Reasoner) is a tool used for look-ahead and design space exploration for a System of Interest (SoI).</p>
GEAR	<p>GEAR (Generic Ensemble Archetype Reasoners) is a set of rule-based tools used to develop archetypes for analysis, design, and implementation.</p>
Galileo	<p>Galileo is a tool used to operate on and reason with AMIL data and library information. Galileo automates through orchestration and choreograph the product development cycle of design and test exploration (processes).</p>
Interface Standards	<p>Interface Standards is a set of hardware and software interface documentation that identify and define:</p> <ul style="list-style-type: none"> <li>• Internal interfaces between ARRoW elements (data and tools)</li> <li>• External interfaces to the AIDE</li> </ul>

Block Name	Description
Library	A library set consisting of product component models, project & product development archetypes, and product context models used to develop a System of Interest (SoI).
Master Model	The ARRoW Master Model (MM) is the aggregate of all data, model elements and their relationships that define and document the System of Interest (SoI) under development as well as the management of the SoI development process.
Mechanical CAD Tool	A Mechanical Computer-Aided Design (CAD) tool is a design process and documentation application that supports multi-dimension modeling (e.g., 2 or 3) of physical elements and their materials for a System of Interest (SoI).
Metrics Dashboard	The Metrics Dashboard is the set of user interfaces (UIs) to support the Metrics Framework applications that aid in the development and management of project and product metrics information.
Metrics Framework	Metrics Framework is a set of application programs that aid in the development and management of project and product metrics information that include: <ul style="list-style-type: none"> <li>• Project health</li> <li>• System and cost effectiveness</li> <li>• Problem domain understanding</li> <li>• Design maturity and health</li> </ul>
PLM Repository	A PLM (Product Lifecycle Management) Repository is a set of data stores managed by a PLM Tool for all information that affects a product from concept, through design and release for production, to operations & support and disposal.
PLM Tool	A PLM (Product Lifecycle Management) tool is an application tool that manages and communicates all information that affects a product from concept, through design and release for production, to operations & support and disposal.
Project Health Archetype Library	The Project Health Archetype Library can include the following archetype libraries for: <ul style="list-style-type: none"> <li>• Risk and opportunity management</li> <li>• Cost and schedule management</li> </ul>
Requirements Archetype Library	The Requirements Archetype Library is a repository for templates of requirement types (categories). It is expected that requirement archetype templates will include: <ul style="list-style-type: none"> <li>• Operational requirements</li> <li>• Design constraints</li> <li>• External Interfaces</li> <li>• Budgetary requirements</li> <li>• Product delivery requirements</li> </ul>

Block Name	Description
Sol Analysis Data	System of Interest (Sol) analysis data is investigative methodical information about a product's operational usefulness, support burden, and design. The Sol analytical data can range from the system level to individual domain specific engineering disciplines. Systems level analysis can include cost and system effectiveness analyses, life cycle cost analysis, formal decisions/trade studies, architectural analysis, capability & gap assessments, risk analysis, etc. Domain specific engineering analytical information can include: thermal, shock and vibration, electrical, communications, etc.
Sol ARRoW Infrastructure Data	System of Interest (Sol) ARRoW Infrastructure Data is the information generated as a result of using the ARRoW infrastructure tools on the Sol.
Sol Design Data	System of Interest (Sol) design data is the plan elements of information for the creation of a product that is either a development item or a non-developmental item. Design data can range from conceptual information, through detail design information of domain specific engineering disciplines and "build to print" production information, to "as built" information of a deployed product.
Sol Verification Data	System of Interest (Sol) verification data is confirmation information about a product that determines whether the product is compliant with its requirements. A verification of a product also includes its work products, (e.g., lower level specifications, designs, processes).
SysML Tool	A Systems Engineering (SE) Modeling Language (SysML) tool is a graphical SE modeling application that supports requirements and architecture development, systems analysis & control, and verification and validation for a System of Interest (Sol).
Test Case Archetype Library	<p>The Test Case Archetype Library is a repository for templates of test case types (categories). It is expected that test case archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Operational requirements</li> <li>• Design constraints</li> <li>• External Interfaces</li> <li>• Budgetary requirements</li> <li>• Product delivery requirements</li> </ul>

Block Name	Description
Test Result Analyzer Archetype Library	<p>The Test Result Archetype Library is a repository for templates of test result types (categories). It is expected that test result archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Roll-Up results</li> <li>• Quantitative</li> <li>• Qualitative</li> <li>• Pass</li> <li>• Pass with margin</li> <li>• Fail</li> <li>• Limited operational use</li> <li>• Metrics</li> <li>• Safety</li> <li>• Certification</li> <li>• Acceptance</li> <li>• Sell-off</li> <li>• Qualification</li> </ul>
Verification Tools	<p>Verification Tools are a set of application programs that aid in early and often confirmation that the specified requirements are fulfilled by the System-of-Interest (SoI). Verification application programs provide enabling information to determine corrective actions on non-conformance issues. Verification tools confirm that the SoI and all its elements perform their intended functions and meet the performance requirements allocated to them (i.e., that the system has been built right). Verification tool usage is also influenced by risk management, and safety and mission criticality of the SoI.</p>
Web UI for Component Review	<p>The Web User Interface (UI) includes all Web-based graphical user interfaces (GUI) to support users of the Component Review Database.</p>

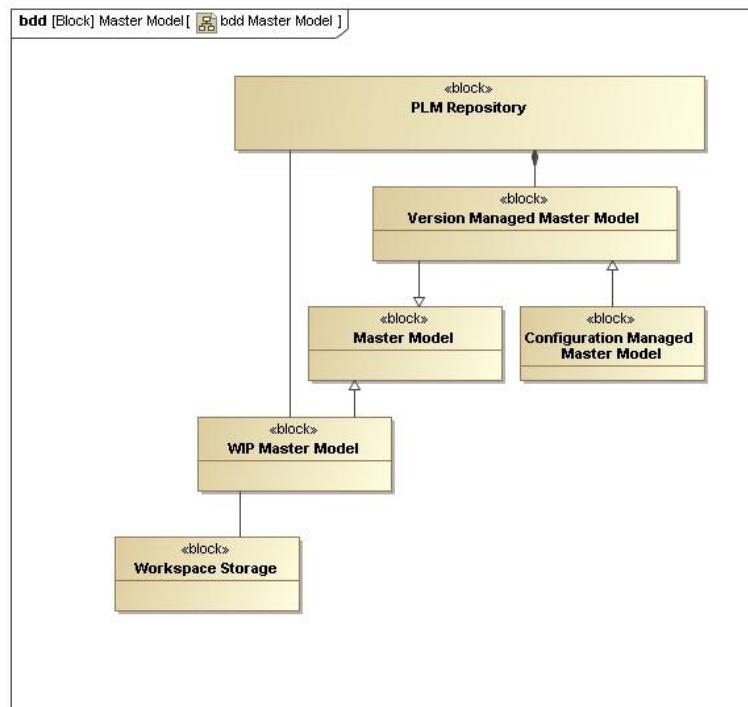
#### 7.1.2.4 ARRoW Master Model

The ARRoW Master Model (MM) is the aggregate of all data, model elements and their relationships that define and document the System of Interest (SoI) under development as well as the management of the SoI development process. The ARRoW development paradigm embraces principles of Model Based Engineering (MBE) and Model Based Systems Engineering (MBSE) whereby, to the maximum extent practical, all aspects of the SoI design and development process are contained in an underlying unified model that can be accessed and manipulated by a diverse set of development tools.

The normal ARRoW process of development entails the import of model elements from the CML and other external sources into the MM, and then a new SoI is developed within the MM by modifying some of the imported elements, creating new elements, and creating new relationships between elements.

Please refer to Figure 7.1-27. Because development of a SoI will normally involve multiple developers working concurrently, the AIDE will need to support this. The evolution and configuration of the MM will need to be managed to the same rigor that data is managed on complex projects today. Thus, we anticipate that the MM will reside in a repository managed by a sophisticated Product Lifecycle Management (PLM) toolset. The AIDE will facilitate

multiple developers concurrently modifying elements of the MM in their local workspace environments and then integrating their changes into the unified MM via a managed process supported by the PLM toolset.



**Figure 7.1-27. Master Model**

Descriptions of the blocks shown in Figure 7.1-27, alphabetically sorted by block name, are provided in Table 7.1-27. The above diagram, and its constituent blocks can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.2 Architecture Development”, with the description text in the documentation metadata field associated with each block.

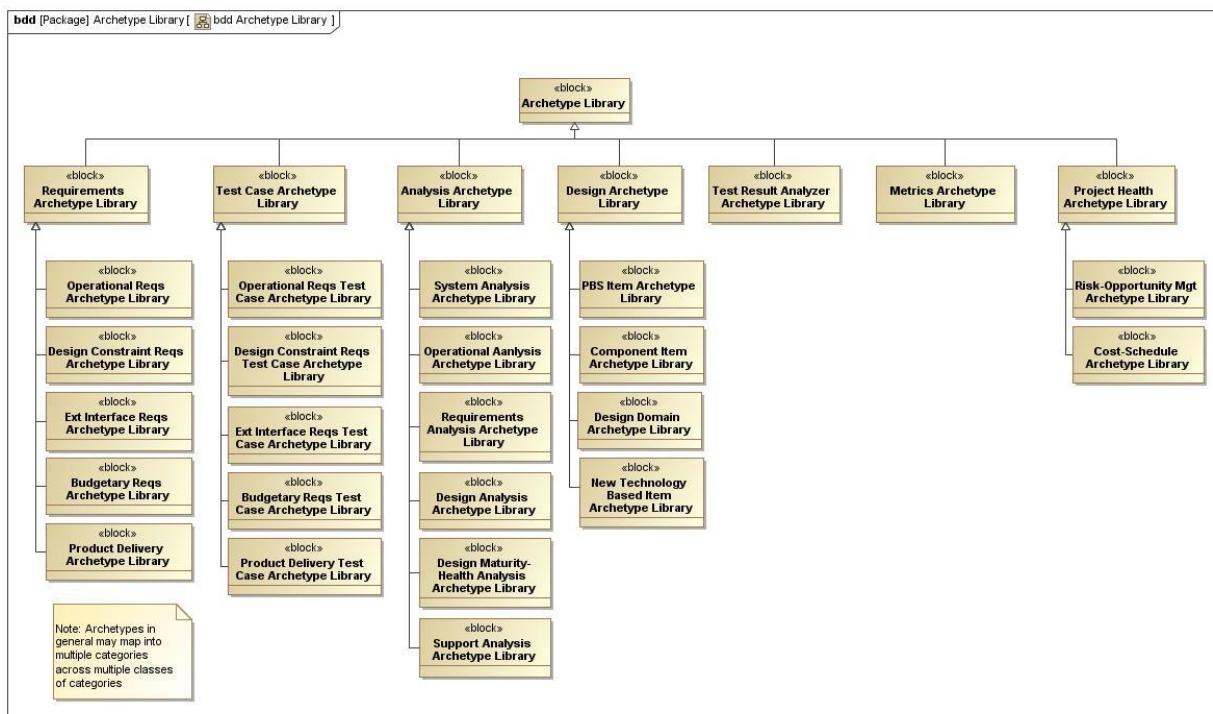
**Table 7.1-27. Master Model Descriptions**

Block Name	Description
Configuration Managed Master Model	This is a specialization of a Version Managed Master Model that not only is version controlled, but additionally is configuration managed.
Master Model	The ARRoW Master Model (MM) is the aggregate of all data, model elements and their relationships that define and document the System of Interest (SoI) under development as well as the management of the SoI development process.
PLM Repository	PLM = Product Lifecycle Management  A PLM Repository is the data store managed by a PLM Tool.
Version Managed Master Model	This is a version of the master model that resides in a repository that enforces version control of the master model and/or its constituent elements.

Block Name	Description
WIP Master Model	<p>WIP = Work in Progress</p> <p>This is a version of the master model that is privately maintained by a particular Sol developer.</p>
Workspace Storage	The memory or storage used by an individual developer that contains the work in progress elements of the master model that the developer needs to access and potentially may need to modify.

### 7.1.2.5 ARRoW Library Elements

In the context of the AIDE, we have identified a number of archetype categories that can potentially contribute to more efficient development of new systems. Refer to section 7.1.2.1 above for a description of archetypes. We have logically organized these archetype categories as if they would each have their own library structure. Whether each of these notional libraries is actually implemented as part of the ARRoW solution is left as a design decision for future consideration. Figure 7.1-28 is a SysML diagram depicting this logical organization of archetype libraries.



**Figure 7.1-28. Archetype Library (Notional)**

Descriptions of the blocks shown in Figure 7.1-28, alphabetically sorted by block name, are provided in

Table 7.1-28. These archetype library blocks can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the nested package chain labeled “6.2 Architecture | Library | Archetype Library”, with the description text in the documentation metadata field associated with each block element.

**Table 7.1-28. Archetype Library Descriptions**

<b>Library Name</b>	<b>Parent</b>	<b>Description</b>
Archetype Library	None	The Archetype Library includes original patterns or models of entities and behaviors from which all things of the same kind are copied or on which they are based.
Budgetary Reqs Archetype Library	Requirements Archetype Library	The Budgetary Reqs Archetype Library can include the following types of requirements that are budgeted to lower level entities of the Sol. <ul style="list-style-type: none"> <li>• Weight</li> <li>• Reliability</li> <li>• Testability</li> <li>• Accuracy</li> <li>• Timeline</li> </ul>
Budgetary Reqs Test Case Archetype Library	Test Case Archetype Library	The Budgetary Requirements Test Case Archetype Library can include the following types of test cases to test the Sol design to the Sol budgetary requirements. <ul style="list-style-type: none"> <li>• Weight</li> <li>• Reliability</li> <li>• Testability</li> <li>• Accuracy</li> <li>• Timeline</li> </ul>
Component Item Archetype Library	Design Archetype Library	The Component Item Archetype Library can include the following building block entities to create component item solution alternatives based an original pattern or model of a component entity. A Component Item Archetype can be use for a reference architecture. Component Item archetypes include: <ul style="list-style-type: none"> <li>• Engines</li> <li>• Transmissions</li> <li>• Chassis/Hulls</li> <li>• Radios</li> <li>• C2</li> <li>• ISR</li> </ul>
Cost-Schedule Archetype Library	Project Health Archetype Library	The Cost-Schedule Archetype Library can include the following cost models and project tracking items to create Cost Model alternatives and Project Tracking alternatives. <ul style="list-style-type: none"> <li>• Development Cost</li> <li>• (AUPC) Average Unit production Cost</li> <li>• (MPC) Manufacturing Production Cost</li> <li>• (VAC) Variance at Complete</li> <li>• Cv &amp; cv (Cumulative &amp; Current Cost Variance)</li> <li>• Sv &amp; sv (Cumulative &amp; Current Schedule Variance)</li> <li>• CPI &amp; cpi (Cumulative &amp; Current Cost Performance Index)</li> <li>• SPI &amp; spi (Cumulative &amp; Current Schedule Performance Index)</li> </ul>

Library Name	Parent	Description
Design Analysis Archetype Library	Analysis Archetype Library	<p>The Design Analysis Archetype Library can include the following types of domain engineering analysis to influence Sol design and determine the maturity of the Sol design.</p> <ul style="list-style-type: none"> <li>• Thermal Analysis</li> <li>• Structural Analysis</li> <li>• Dynamic Analysis: <ul style="list-style-type: none"> <li>– Gun Firing Shock</li> <li>– Firing on Move</li> <li>– Emplaced Firing</li> <li>– Crew Shock &amp; Vibration</li> <li>– Transport Loading</li> <li>– Air Drop</li> <li>– Chassis Vibration Transmission</li> </ul> </li> <li>• Finite Element Analysis</li> <li>• Fluids Analysis</li> <li>• Electrical &amp; Electronics Analysis</li> <li>• Power Analysis</li> <li>• E3/EMC/EMI Analysis</li> <li>• Communications Analysis</li> <li>• Controls Analysis</li> <li>• Timeline Analysis</li> <li>• Equipment Motion Analysis</li> <li>• Signature Management Analysis</li> <li>• Vulnerability Analysis: <ul style="list-style-type: none"> <li>– Shotline Analysis</li> <li>– Ballistic Impact Analysis</li> <li>– Fragmentation Impact Analysis</li> <li>– Shape Charge Impact Analysis</li> <li>– Sympathetic Detonation Analysis</li> <li>– Mine Blast Analysis</li> </ul> </li> <li>• Information Assurance Analysis</li> <li>• Fire Control Analysis <ul style="list-style-type: none"> <li>– Wpn Pointing</li> <li>– Technical Fire Control</li> <li>– Wpn Firing Stationary &amp; On-the-move</li> </ul> </li> <li>• Accuracy Analysis</li> <li>• HSI Analysis</li> <li>• Safety Analysis</li> <li>• Reliability, Maintainability, &amp; Testability Analysis</li> <li>• Assembly &amp; Producibility Analysis</li> <li>• Logistics Analysis</li> <li>• Transportability Analysis</li> </ul>
Design Archetype Library	Archetype Library	<p>The Design Archetype Library is a repository for templates of design types (categories). It is expected that the design archetype templates will include:</p> <ul style="list-style-type: none"> <li>• PBS Items</li> <li>• Component Items</li> <li>• Design Domains</li> <li>• New Technology Based Items</li> </ul>

Library Name	Parent	Description
Design Constraint Reqs Archetype Library	Requirements Archetype Library	<p>The Design Constraint Requirements Archetype Library can include the following types of design constraint requirements imposed on the Sol.</p> <ul style="list-style-type: none"> <li>• Context – Environment</li> <li>• EE, ME, &amp; SW</li> <li>• Specialty Engineering - Safety, HSI, RM&amp;T</li> <li>• Standards - Federal, Military, ESOH, etc.</li> <li>• Operational and Life Cycle – PHST</li> <li>• Cost</li> </ul>
Design Constraint Reqs Test Case Archetype Library	Test Case Archetype Library	<p>The Design Constraint Requirements Test Case Archetype Library can include the following types of test cases to test the Sol design adherence to design constraint requirements.</p> <ul style="list-style-type: none"> <li>• Context – Environment</li> <li>• EE, ME, &amp; SW</li> <li>• Specialty Engineering - Safety, HSI, RM&amp;T</li> <li>• Standards - Federal, Military, ESOH, etc.</li> <li>• Operational and Life Cycle – PHST</li> <li>• Cost</li> </ul>
Design Domain Archetype Library	Design Archetype Library	<p>The Design Domain Archetype Library can include the following building block design domains for use in the design of DMI-PBS Items and Component Items and to create Design Domain alternatives.</p> <ul style="list-style-type: none"> <li>• Physical <ul style="list-style-type: none"> <li>– Includes spatial guidelines, rules, and constraints. Used to create layout alternatives based an original pattern or model of building blocks</li> </ul> </li> <li>• Thermal</li> <li>• Power</li> <li>• Controls</li> <li>• Fire Control</li> <li>• Signal</li> <li>• Computing</li> <li>• Platform Electronics</li> <li>• C4ISR</li> <li>• BattleSpace Communications</li> <li>• Network Ready</li> <li>• EMC/E3</li> <li>• SW</li> <li>• Information Assurance (IA)</li> <li>• Crew/Battle Station</li> <li>• Training</li> </ul>

Library Name	Parent	Description
Design Maturity-Health Analysis Archetype Library	Analysis Archetype Library	<p>The Design Maturity-Health Analysis Archetype Library can include the following types of domain engineering analysis to monitor the progress of the Sol design and determine the maturity of the Sol design.</p> <ul style="list-style-type: none"> <li>• TPMs (Technical Performance Measures): <ul style="list-style-type: none"> <li>- Weight</li> <li>- Start-Up (All Up, Drive Away - Sec)</li> <li>- Energy Conservation (MPG, GPH)</li> <li>- Movement Speeds (CC, Sprint, HWY - MPH)</li> <li>- Engagement Response Time (Sec)</li> <li>- Fires (Ph, RgMax, Min, &amp; Effect, ROF)</li> <li>- Tgt Acq/SA (Rg, Accuracy, Coverage)</li> <li>- Engagement Detection (Point of origin)</li> <li>- Force Protection (Egress, IED)</li> <li>- Reliability (MTBSA, MTBF)</li> <li>- Maintainability (MTTR)</li> <li>- Reserve Capacity (Pwr, Therm, Proc, Mem)</li> <li>- Force Interoperability (Opsn/Logistics)</li> <li>- MOSA (Degree, # of Characteristics)</li> </ul> </li> <li>• Requirement Compliance <ul style="list-style-type: none"> <li>- Performance</li> <li>- Functional</li> <li>- Design Constraints</li> <li>- Interfaces</li> <li>- Safety</li> </ul> </li> <li>• TRL (Technical Readiness Level) <ul style="list-style-type: none"> <li>- Green Energy Propulsion TRL</li> <li>- TRL #n</li> </ul> </li> <li>• MRL (Manufacturing Readiness Level) <ul style="list-style-type: none"> <li>- Green Energy Propulsion MRL</li> <li>- MRL #n</li> </ul> </li> <li>• State of Design Integration <ul style="list-style-type: none"> <li>- Problem Burn Down</li> <li>- Problem Criticality (#, Degree)</li> <li>- Integration Demographics</li> <li>- # of Re_Integrations</li> </ul> </li> <li>• Req-Design Feature Bi-Directional Traceability</li> </ul>
Ext Interface Reqs Archetype Library	Requirements Archetype Library	<p>The External Interface Requirements Archetype Library can include the following types of external interface requirements to the Sol.</p> <ul style="list-style-type: none"> <li>• Mechanical</li> <li>• Electrical</li> <li>• SW</li> <li>• ICDs</li> <li>• Standards</li> <li>• Interface MIL-STDs</li> </ul>

Library Name	Parent	Description
Ext Interface Reqs Test Case Archetype Library	Test Case Archetype Library	<p>The External Interface Requirements Test Case Archetype Library can include the following types of test cases to test the Sol design interface compatibility and adherence with external interface requirements to the Sol.</p> <ul style="list-style-type: none"> <li>• Mechanical</li> <li>• Electrical</li> <li>• SW</li> <li>• ICDs</li> <li>• Standards</li> <li>• Interface MIL-STDs</li> </ul>
Metrics Archetype Library	Archetype Library	<p>The Metrics Archetype Library is a repository for templates of metrics types (categories). It is expected that the metric archetype templates will include:</p> <ul style="list-style-type: none"> <li>• System and cost effectiveness</li> <li>• Problem domain understanding</li> <li>• Design maturity and health</li> </ul>
New Technology Based Item Archetype Library	Design Archetype Library	<p>The New Technology Based Item Archetype Library can include the following newly introduced building block technology item for use in the design of DMI-PBS Items and Component Items and to create New Technology Based Item alternatives.</p> <ul style="list-style-type: none"> <li>• Counter Threat Blast Effects (IED) <ul style="list-style-type: none"> <li>– Blast Chimneys</li> </ul> </li> <li>• Hybrid Electric Drives</li> <li>• Green Energy Sources</li> <li>• Precision Guided Munitions</li> <li>• GPS Based Applications</li> </ul>
Operational Analysis Archetype Library	Analysis Archetype Library	<p>The Operational Analysis Archetype Library can include the following types of operational analysis to determine the operational impacts of the Sol capabilities and design.</p> <ul style="list-style-type: none"> <li>• Effects/Lethality Analysis</li> <li>• Mobility Analysis</li> <li>• Battlefield Communications Analysis</li> <li>• Survivability Analysis <ul style="list-style-type: none"> <li>– Self Defense</li> </ul> </li> <li>• Interoperability</li> <li>• Threat Analysis</li> </ul>

Library Name	Parent	Description
Operational Reqs Archetype Library	Requirements Archetype Library	<p>The Operational Requirements Archetype Library can include the following types of usage requirements for the Sol.</p> <ul style="list-style-type: none"> <li>• Scenario - OMS/MP</li> <li>• Acceleration</li> <li>• Rate</li> <li>• Response</li> <li>• I/O</li> <li>• Capacity</li> <li>• State/Mode Transitions</li> <li>• Functionality</li> <li>• Capability</li> </ul>
Operational Reqs Test Case Archetype Library	Test Case Archetype Library	<p>The Operational Requirements Test Case Archetype Library can include the following types of test cases to test the Sol design performance to meet the operational requirements.</p> <ul style="list-style-type: none"> <li>• Scenario - OMS/MP</li> <li>• Acceleration</li> <li>• Rate</li> <li>• Response</li> <li>• I/O</li> <li>• Capacity</li> <li>• State/Mode Transitions</li> <li>• Functionality</li> <li>• Capability</li> </ul>
PBS Item Archetype Library	Design Archetype Library	<p>PBS = Product Breakdown Structure</p> <p>The PBS Item Archetype Library can include the following product structures for Defense Material Items (DMI) to create materiel solution alternatives based an original pattern or model of a product framework or structure of entities. A PBS Item Archetype can be use for a reference architecture</p> <p>DMI-PBS Item archetypes include product structures for:</p> <ul style="list-style-type: none"> <li>• Ground Vehicle Systems (Surface Vehicle Systems)</li> <li>• Ordnance Systems</li> <li>• Maritime Systems (Sea Systems)</li> <li>• Missile Systems</li> <li>• Aircraft Systems</li> <li>• Electronic/Automated Software Systems</li> <li>• Space Systems</li> <li>• Unmanned Vehicle Systems</li> </ul>

Library Name	Parent	Description
Product Delivery Archetype Library	Requirements Archetype Library	<p>The Product Delivery Reqs Archetype Library can include the following types of requirements that measure and improve the quality, performance, and delivery velocity of the Sol to satisfy our customer's needs, on time, and within cost commitments.</p> <ul style="list-style-type: none"> <li>• Qualification</li> <li>• Acceptance</li> <li>• Certification</li> <li>• Safety</li> </ul>
Product Delivery Test Case Archetype Library	Test Case Archetype Library	<p>The Product Delivery Requirements Test Case Archetype Library can include the following types of test cases to measure and improve the quality, performance, and delivery velocity of the Sol design to satisfy our customer's needs, on time, and within cost commitments.</p> <ul style="list-style-type: none"> <li>• Qualification</li> <li>• Acceptance</li> <li>• Certification</li> <li>• Safety</li> </ul>
Project Health Archetype Library	Archetype Library	<p>The Project Health Archetype Library can include the following archetype libraries for:</p> <ul style="list-style-type: none"> <li>• Risk and opportunity management</li> <li>• Cost and schedule management</li> </ul>
Requirements Analysis Archetype Library	Analysis Archetype Library	<p>The Requirements Analysis Archetype Library can include the following types of problem domain analysis to define the customer needs, required functionality, and the complete problem: operations, cost and schedule, performance, training and support, test, manufacturing, and disposal.</p> <ul style="list-style-type: none"> <li>• Capability &amp; Gap Analysis</li> <li>• Functional Analysis</li> <li>• Use Case/Scenario Analysis</li> <li>• Object Oriented Analysis</li> <li>• Requirements Maturity</li> <li>• Requirements Stability</li> <li>• Requirements Bi-Directional Traceability</li> <li>• External Interface Definition</li> <li>• SWaPC-C Analysis</li> </ul>
Requirements Archetype Library	Archetype Library	<p>The Requirements Archetype Library is a repository for templates of requirement types (categories). It is expected that requirement archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Operational requirements</li> <li>• Design constraints</li> <li>• External Interfaces</li> <li>• Budgetary requirements</li> <li>• Product delivery requirements</li> </ul>

Library Name	Parent	Description
Risk-Opportunity Mgt Archetype Library	Project Health Archetype Library	<p>The Risk-Opportunity Management Archetype Library can include the following example risk and opportunity items to create Cost Model alternatives and Project Tracking alternatives.</p> <ul style="list-style-type: none"> <li>• If PwrPk Perf is not obtained, then Dash Speed is not met</li> <li>• If R occurs, then consequence C - Risk n</li> <li>• If alternative energy density is obtained, then energy conservation is exceeded</li> <li>• If O occurs, then benefit B - Opportunity n</li> </ul>
Support Analysis Archetype Library	Analysis Archetype Library	<p>The Support Analysis Archetype Library can include the following types of life cycle sustainment analysis to influence Sol design and determine the logistics support package to sustain over the life cycle of the Sol.</p> <ul style="list-style-type: none"> <li>• Sustainment Analysis</li> <li>• LCEP Analysis (Life Cycle Environmental Profile)</li> </ul>
System Analysis Archetype Library (1 of 2)	Analysis Archetype Library	<p>The System Analysis Archetype Library can include the following types of effectiveness, and capability assessment &amp; gap analysis on the Sol design.</p> <ul style="list-style-type: none"> <li>• Cost Effectiveness Analysis: <ul style="list-style-type: none"> <li>- Sys Effectiveness/LCC</li> <li>- Availability/LCC</li> <li>- Sys Capacity/LCC</li> <li>- System Benefit/LCC</li> </ul> </li> <li>• System Effectiveness Analysis: <ul style="list-style-type: none"> <li>- Measures of Effectiveness (MOEs) <ul style="list-style-type: none"> <li>o Pk, Pra, (Mobility, Firepower)</li> <li>o Effects on Tgt (Destroy, Neutralize, Suppress)</li> <li>o # Objectives Seized</li> </ul> </li> <li>- Measures of Performance (MOPs) <ul style="list-style-type: none"> <li>o Range</li> <li>o RoF</li> <li>o Tgt Acq (Detect, Tracking)</li> <li>o Position &amp; Heading</li> <li>o Response Time</li> <li>o ToF</li> <li>o Movement Speeds</li> <li>o Terrain Negotiation</li> </ul> </li> <li>- Measures of Usage (MOUs) <ul style="list-style-type: none"> <li>o Engagement Duration (FM, Direct, Tgt Acq)</li> <li>o # of Rds Fired/per Msn/kill</li> <li>o # of Supplies Transferred</li> <li>o # Tgts Defeated</li> <li>o # Msn Executed</li> <li>o # of Moves</li> </ul> </li> </ul> </li> </ul>

Library Name	Parent	Description
System Analysis Archetype Library (2 of 2)	Analysis Archetype Library	<ul style="list-style-type: none"> <li>- Measures of Suitability (MOSs) <ul style="list-style-type: none"> <li>o Weight</li> <li>o Ease of installation</li> <li>o Interoperability</li> <li>o Adaptability</li> </ul> </li> <li>• Capability &amp; Gap Assessments <ul style="list-style-type: none"> <li>- Key Performance Parameters (KPPs) <ul style="list-style-type: none"> <li>o Survivability</li> <li>o Force Protection</li> <li>o Sustainment (Availability)</li> <li>o Net-Ready/Interoperability</li> <li>o System Training</li> <li>o Energy Efficiency</li> </ul> </li> <li>- Key System Attributes (KSAs) <ul style="list-style-type: none"> <li>o Command &amp; Control (Contact, Info, Planning)</li> <li>o Intelligence (Coverage, METT, Tgt Acq Rg)</li> <li>o Fires (Capacity, Response Time, Pk, Rg)</li> <li>o Movement &amp; Maneuver (Speed, Terrain, Transport)</li> <li>o Protection (Jam Resistance, IA)</li> <li>o Sustainment (Reliability &amp; Ownership Cost)</li> </ul> </li> </ul> </li> <li>System Effectiveness—A probability measure that the system solution can successfully meet an overall operational demand within a given time when operated under specific conditions.</li> <li>Reflects the technical characteristics of the system solution (e.g., performance, availability, supportability, dependability). The ability of the system solution to do the job for which it was intended.</li> <li>• Cost Effectiveness - A measure of a system solution in terms of mission fulfillment (system effectiveness) and total life-cycle cost (LCC). Reliability is a major factor in determining the cost effectiveness of a system solution <ul style="list-style-type: none"> <li>- Find the most effective solution with the least cost by determining the cost-effectiveness differences between alternatives of solutions</li> </ul> </li> </ul>
Test Case Archetype Library	Archetype Library	<p>The Test Case Archetype Library is a repository for templates of test case types (categories). It is expected that test case archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Operational requirements</li> <li>• Design constraints</li> <li>• External Interfaces</li> <li>• Budgetary requirements</li> <li>• Product delivery requirements</li> </ul>

Library Name	Parent	Description
Test Result Analyzer Archetype Library	Archetype Library	<p>The Test Result Archetype Library is a repository for templates of test result types (categories). It is expected that test result archetype templates will include:</p> <ul style="list-style-type: none"> <li>• Roll-Up results</li> <li>• Quantitative</li> <li>• Qualitative</li> <li>• Pass</li> <li>• Pass with margin</li> <li>• Fail</li> <li>• Limited operational use</li> <li>• Metrics</li> <li>• Safety</li> <li>• Certification</li> <li>• Acceptance</li> <li>• Sell-off</li> <li>• Qualification</li> </ul>

#### 7.1.2.6 ARRoW Requirements to Test Case Flow Architecture

The architecture described in this section is that portion of the AIDE related to functionally how requirements are ingested into the master model, how test cases relate to the verification of those requirements, how run time testing of requirements is executed, and how test result verdicts are determined and flowed to other elements within the AIDE. We refer to this overall process as the Requirements to Test Case (RTTC) Flow.

Figure 7.1-29 shows a simplified view of the static architecture of ARRoW elements related to the RTTC flow. These elements are shown as they relate within both the Master Model (MM) and the Component Model Library (CML). Figure 7.1-30 shows a simplified view of the entities and their interfaces that are involved in the run time execution of the test and verification process. Please refer to both of these diagrams as part of the following discussion.

Refer to section 7.1.2.1.2 for a more comprehensive discussion of requirements, requirement archetypes, and requirement archetype sets. A Requirement Archetype Set (RAS) contains multiple Requirement Archetypes (RAs) and optionally other RASs. The CML contains both RASs and RAs. RAs are indirectly bound to Design Archetypes (DAs) in the CML via their mutual Test Case Archetypes (TCAs). When DAs are imported into the MM, references to the associated TCAs and RAs are also automatically imported. Multiple TCA-RA pairs will quite possibly be associated with a particular DA, since multiple requirements are commonly allocated to components. However, mutually exclusive TCA-RA pairs may also be associated with a particular DA element. For example, for US Marine Corps use of a component, one TCA-RA pair might apply, but for US Army use of that same component, a different TCA-RA pair might apply. By importing the appropriate RAS(s) from the CML, the imported RA references can be de-conflicted within the MM, and consequently the appropriate TCAs can be determined as well since each RA has exactly one corresponding TCA.

At this point in our discussion, the MM has RAS(s), RAs, TCAs, and DAs – all archetypes. Archetypes must generally be refined to instances that specifically apply to the System of Interest (SoI) being developed.

Each DA will be refined to a Design Component (DC) within the MM. This might be accomplished, for example, by assigning values to design parameters, selecting specific equipment models from the CML, or even selecting optional architectural relationships from the CML. Note that in the context of this discussion, a DA or a DC can be at any level within a product structure hierarchy (e.g., system, subsystem, assembly, or component).

Once refined, a DC will have one or more Component Models (CMs). A CM is a model of a component that potentially has “white box” knowledge of the component design. This knowledge is frequently necessary for all but low fidelity models of components. Since details of the design may be modeled, the construction of a CM may be proprietary. CMs are generally written and published in the CML for the purpose of supporting some or all of the suite of tests that are expected to be applied to the DC. A DC might support multiple CMs differing in fidelity and performance that are compatible with the same Test Case (TC). The DC itself provides a standard interface between the CMs and TCs that orchestrate the test of the DC. This standard interface support includes both run time data exchange interfaces as well as configuration/setup and status reporting interfaces.

A TC is an executable that stimulates the DC for the purpose of verifying a requirement. The DC has a corresponding CM that reacts to the stimuli and produces output that can then be either compared to a required output or routed to some other element within the MM. A TC does not have visibility to the internal structure or behavior of the DC, including its associated CM. In other words, it has a “black box” interface to the DC. This standardized test mechanism can thus be reused against many alternative design solutions and revisions without needing to change the structure/logic of the TC.

A TC may be a modification of a TCA to make it an executable for supporting test of a specific DC, or it may be the unmodified TCA itself if that TCA can appropriately be applied to the run time test of the DC.

As mentioned above, importing the DA will result in the importation of one or more RAs associated with the DA and its refined DC. Each of these RAs may or may not be refined to a corresponding requirement. The decision to create a requirement is based on whether a requirement is deemed to be needed at any particular level within the SoI product structure. Requirements will minimally be needed at the system level, since in general they will need to trace to the customer provided source requirements. However, at lower levels in the product structure, the choice to create a requirement will be based on whether a test result at that level needs to be compared against some expected result, threshold value, or utility function. If it does not, then the test result can instead be routed up the product structure hierarchy for further processing so that it can support verification of a higher level requirement.

If a requirement *is* created by refining it from its parent RA, then it needs to be verified, of course. Requirements are verified with the addition of one more element: a Test Result Analyzer (TRA). Generally, a requirement is verified by the following process: a test case stimulates the DC, the DC reacts with a resultant output, this output is then compared to the required output by the TRA, and the result of this comparison (the verdict) is then dispatched or published to the appropriate consumers such as verdict loggers, the metrics framework, or an element of the ARRoW user interface.

It is anticipated that common patterns for the construction of TRAs will be defined, including the use of common, reusable code. As an aid to the development of TRAs, we allow for the

definition of TRA archetypes. TRA Archetypes reside in the CML and can be imported into the MM and refined to create TRA instances specific to the verification tests needed.

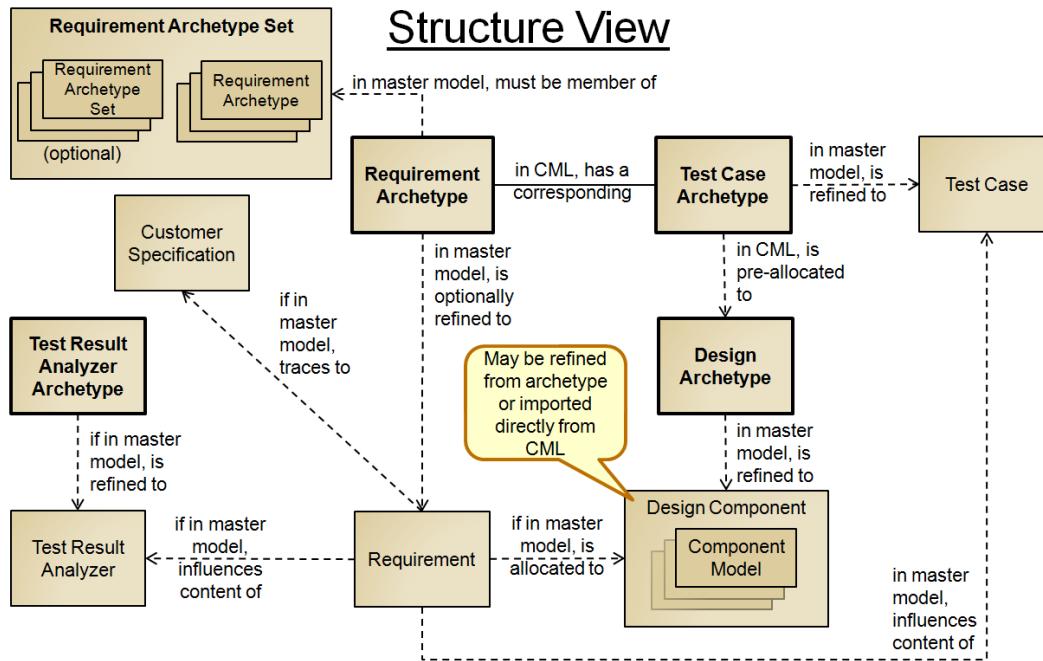


Figure 7.1-29. RTTC Entities - Structure View

## Verification Run-Time Interfaces View

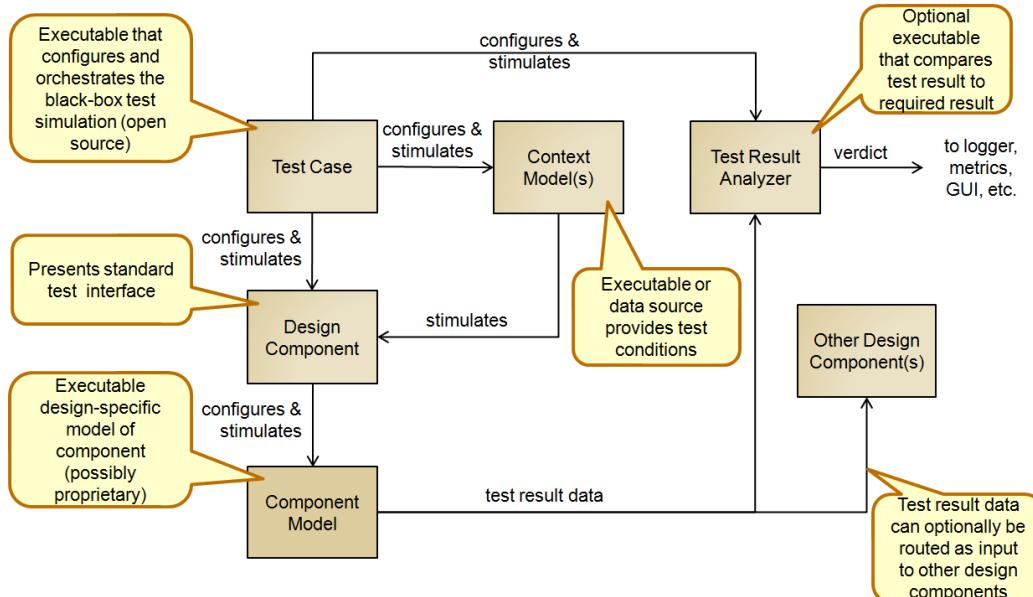
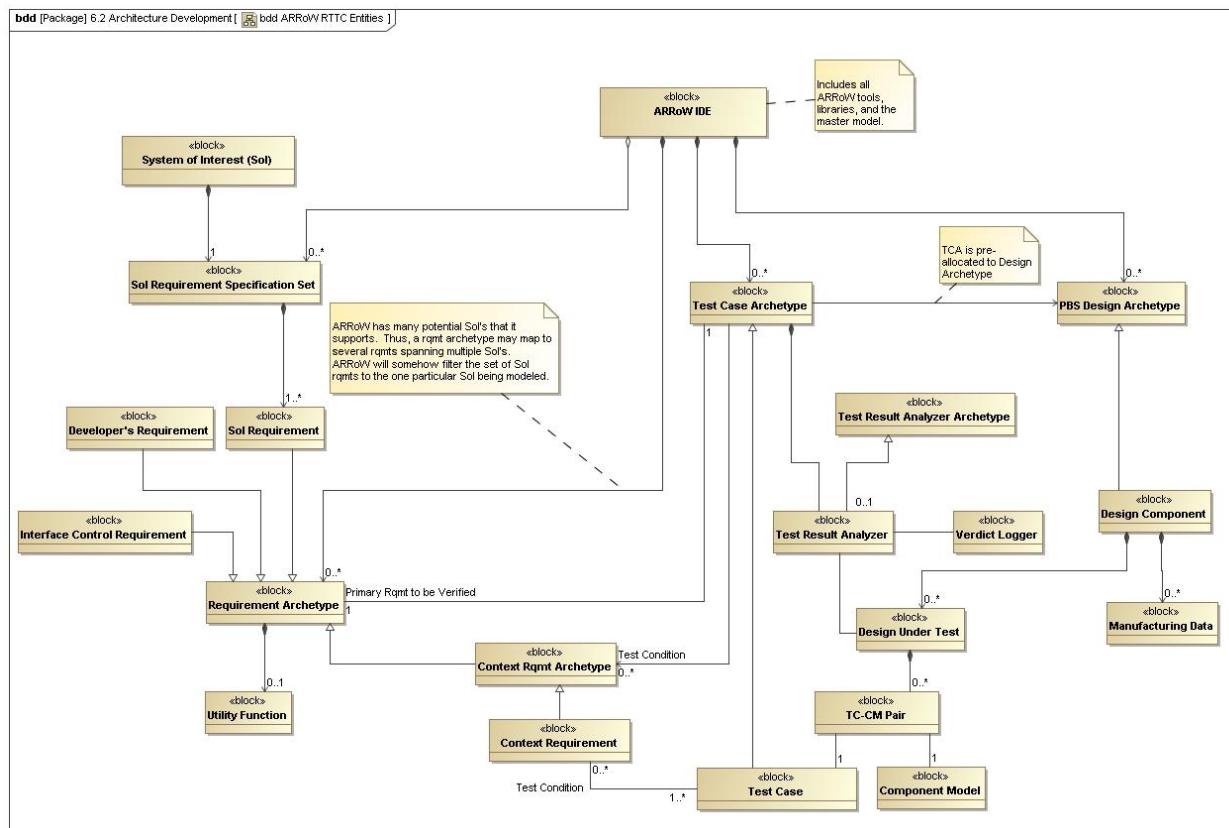


Figure 7.1-30. RTTC Entities - Run-Time Interfaces View

**Figure 7.1-31. ARRoW RTTC Entities**

Descriptions of the blocks shown in Figure 7.1-31, alphabetically sorted by block name, are provided in Table 7.1-29. The above diagram, and its constituent blocks can be additionally found in the MagicDraw file “META\_Project.mdzip”, in the package labeled “6.2 Architecture Development”, with the description text in the documentation metadata field associated with each block. This diagram is essentially a SysML metamodel of the ARRoW RTTC architecture.

**Table 7.1-29. ARRoW RTTC Entities Description**

Block Name	Description
AIDE	The AIDE includes all ARRoW tools, libraries, and the master model.
Component Model	<p>A component model models the DUT in the context of a specific test case. In general, it is constructed to react to stimuli from the test case and provide response output representative of what the real component (that realizes the design) would do.</p> <p>This component model supports use of legacy tools that might include white box insight into the DUT as well as external environmental factors.</p>
Context Requirement	<p>A Context Requirement (CR) is a requirement that specifies the external environment to which a design will be exposed. It includes natural or induced environments such as temperature or electromagnetic effects, as well as interoperability/interface requirements with external systems.</p> <p>A CR is sometimes not tested directly, but rather is verified by executing a test case for each requirement that uses that context requirement as part of its set of test conditions.</p>
Context Rqmt Archetype	A Context Requirement Archetype is simply an archetype for a Context Requirement.
Design Component	A design component is a specific implementation that conforms to a PBS Design Archetype. It can be at any level in a product breakdown structure.
Design Under Test	A Design Under Test (DUT) is that portion of a Design Component that can simulate the design in order to verify that the design conforms to the requirements allocated to it.
Developer's Requirement	This includes business development goals, budgets (tolerance/power/etc.). This is requirements not validated by customer but used by contractor management to influence/constrain the design. Is problem space related. Does not include design rules, design guidelines, etc. that are solution space related.
Interface Control Requirement	<p>Legacy interface management approaches that manage interfaces such as using Interface Control Documents (ICDs) will employ an interface requirement structure in the ARRoW environment.</p> <p>Note that interfaces will include both design integration interfaces as well as ARRoW environment interfaces. An example of an s ARRoW environment interface would be a test case configuration interface.</p>
Manufacturing Data	This block is included to illustrate that a design can include more elements than just virtual test related things.
PBS Design Archetype	<p>PBS = Product Based Structure</p> <p>This is a template for similar design types (architectures). A PBS Design Archetype facilitates reuse via abstracted levels in a reference design architecture.</p>

Block Name	Description
Requirement Archetype	<p>This is a template for similar requirement types (categories). It is expected that many types will exist. A requirement Archetype facilitates reuse and mapping to abstracted levels in a reference design architecture.</p> <p>Requirement Archetypes generally conform to the following:</p> <ol style="list-style-type: none"> <li>1. It has a template expression of a requirement, written in a way that can be ultimately refined to be a well expressed requirement. For example, this could be written in natural language text conforming to well known systems engineering best practice characteristics or even a formal requirements expression language. It can support parameterization if necessary.</li> <li>2. It supports default traceability to multiple (0..*) reference architecture and/or design entities. TBR: this needs to be captured in the block definition diagram (bdd).</li> <li>3. It has a one-to-one mapping to a specific test case archetype.</li> <li>4. It supports multiple (0..*) references to requirement archetypes on which the associated test case results depend.</li> <li>5. It supports (0..1) utility functions.</li> </ol>
Sol Requirement	This type of requirement must be validated with the customer, and its verification must be validated by the customer.
Sol Requirement Specification Set	This includes all specifications that are part of the Sol's spec tree (e.g., A-spec, system spec, subsystem specs, critical item development specs, etc.). This includes customer supplied specs as well as developer created specs.
System of Interest (Sol)	The System of Interest (Sol) is the system that is being developed using the ARRoW toolset. In this context a Sol can be an entity at any level within a product breakdown structure.
TC-CM Pair	This is simply a container that shows that there is a one-to-one relationship between a specific test case (TC) and its corresponding component model (CM).
Test Case	<p>A test case is an executable that stimulates the DUT for the purpose of verifying a requirement. The DUT has a corresponding component model that reacts to the stimuli and produces output that can then be compared to the required output.</p> <p>A Test Case does not have visibility to the internal structure or behavior of the DUT including its associated component model. This standardized test mechanism can be reused against many design solutions without needing to change the structure/logic of the test case.</p> <p>A test case may be a modification of a test case archetype to make it an executable for supporting verification of a specific requirement against a specific DUT, or it may be the unmodified test case archetype itself if that TCA can appropriately be applied to the DUT in this context.</p>

Block Name	Description
Test Case Archetype	<p>This is an archetype for similar test case types (categories). It is expected that these archetypes will map one-to-one to requirement archetypes. A test case Archetype facilitates reuse and mapping to abstracted levels in a design reference architecture.</p> <p>Certain classes of requirements will have a set of recurrent constraint (context) requirements that apply during the verification test as test conditions.</p> <p>For example, ambient temperature will normally influence mobility performance, so the verification of, say an acceleration requirement, would be done using test conditions of the required ambient operational temperature range. Verification of the acceleration requirement is therefore dependent on the ambient operational temperature requirement.</p> <p>Note: the dependent RAs could optionally be copied into the Master Model when the parent RA is copied, with ARRoW guiding the user to fill in appropriately. Include notion of shallow copy vs. deep copy.</p>
Test Result Analyzer	<p>Generally, a requirement is verified by the following: a test case stimulates the DUT, the DUT provides resultant output, and this output is then compared to the required output by the Verifier. In some cases, the Test Case may execute without the need to execute the Test Result Analyzer (TRA). For example, the DUT output may be routed to a higher assembly level Test Case or DUT, where the summary test results are verified at that level only.</p>
Test Result Analyzer Archetype	<p>This is a template for similar Test Result Analyzer (TRA) logic classes. A TRA Archetype facilitates reuse of verification patterns.</p>
Utility Function	<p>This is a way of relating the DUT performance/capability/behavior to perceived value to the customer or management. Perceived value can be binary (e.g., pass/fail), enumerated (e.g., fail, threshold, objective, exceeded), continuously scaled (e.g., linear, exponential, s-curve, etc.), or any other function deemed applicable. Note: include default utility function support in some RAs.</p>
Verdict Logger	<p>This presents verification results to the user and/or saves the results to persistent storage.</p>

### 7.1.3 Bibliography

[AMSC05] Army Materiel Systems Command (2005), MIL-HDBK-881A: Work Breakdown Structures for Defense Materiel Items, available at <http://www.acq.osd.mil/>

[BF90] Blanchard, B. S., & Fabrycky, W. J., (1990), Systems Engineering And Analysis, Engle Cliffs, N.J., Prentice-Hall

[CJCS07] Chairman of the Joint Chiefs of Staff, 2007, OPERATION OF THE JOINT CAPABILITIES INTEGRATION AND DEVELOPMENT SYSTEM, available at [http://www.dtic.mil/cjcs\\_directives/cdata/unlimit/m317001.pdf](http://www.dtic.mil/cjcs_directives/cdata/unlimit/m317001.pdf)

- [CJCS09] Chairman of the Joint Chiefs of Staff, 2009, JOINT CAPABILITIES INTEGRATION AND DEVELOPMENT SYSTEM, available at [http://www.dtic.mil/cjcs\\_directives/cdata/unlimit/3170\\_01.pdf](http://www.dtic.mil/cjcs_directives/cdata/unlimit/3170_01.pdf)
- [DC11] Dictionary.Com (2011), <http://dictionary.reference.com/browse/>
- [HOF11] Hofmann, G. (2011), “What Is the Cost Of Poor Requirements Management,” available at <http://www.reqdb.com/mediawiki/>
- [INC10] International Council on Systems Engineering. (2010), Systems Engineering Handbook, version 3, available at <http://www.incose.org/>
- [IBM11a] IBM (2011), "Rational DOORS," available at <http://www-01.ibm.com/software/awdtools/doors/>
- [IBM11b] IBM (2011), "Rational RequisitePro," available at <http://www-01.ibm.com/software/awdtools/reqpro/>
- [MD11] MagicDraw (2011) "Cameo Requirements+," available at <https://www.magicdraw.com/cameoreq>
- [MRL10], OSD Manufacturing Technology Program. (2010), Manufacturing Readiness Level Deskbook, available at [http://www.dodmrl.com/MRL\\_Deskbook\\_30\\_July\\_2010.pdf](http://www.dodmrl.com/MRL_Deskbook_30_July_2010.pdf)
- [SEF01] (2001) Systems Engineering Fundamentals, Defense Acquisition University Press, [http://www.chassis-plans.com/PDF/DOD\\_Systems\\_Engineering\\_Fundamentals.pdf](http://www.chassis-plans.com/PDF/DOD_Systems_Engineering_Fundamentals.pdf)
- [WS11] “Suspension (Vehicle),” [http://en.wikipedia.org/wiki/Suspension\\_\(vehicle\)](http://en.wikipedia.org/wiki/Suspension_(vehicle))

# META Adaptive, Reflective, Robust Workflow (ARRoW)

## Phase 1b Final Report

### TR-2742

## Appendix 7.2 - Tool Design

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.2 Tool Design .....</b>	1
<b>7.2.1 AMIL.....</b>	2
7.2.1.1 Introduction .....	2
7.2.1.2 AMIL Structure .....	3
7.2.1.3 Dynamic Nodes .....	5
7.2.1.4 Ontology .....	7
<b>7.2.2 Component Model Library.....</b>	8
7.2.2.1 CML Architecture .....	8
7.2.2.2 CML Search .....	10
<b>7.2.3 Conceptualization – The Early Concepting Tool.....</b>	11
7.2.3.1 ECTo Architecture.....	13
7.2.3.2 ECTo Models.....	16
<b>7.2.4 Co-Analysis and Exploration.....</b>	18
7.2.4.1 Principles Behind GEAR .....	18
7.2.4.2 ESKER .....	19
<b>7.2.5 Tool Plug-ins.....</b>	21
7.2.5.1 Magic Draw/SysML.....	21
7.2.5.2 Pro/Engineer (Creo) Plug-In .....	27
7.2.5.3 Incorporating Lightweight, Open Source, Freeware Tools into the Tool Chain .....	27
<b>7.2.6 Metrics .....</b>	29
7.2.6.1 Metrics Framework.....	29
7.2.6.2 Specific Metrics .....	30
7.2.6.3 Metrics Dashboard .....	32
<b>7.2.7 Cloud Deployment .....</b>	34
<b>7.2.8 Bibliography .....</b>	35

## List of Figures

<b>Figure 7.2-1. AMIL Structure .....</b>	3
<b>Figure 7.2-2. Example Requirement in AMIL.....</b>	5
<b>Figure 7.2-3. Fragment of Executable Model.....</b>	6
<b>Figure 7.2-4. Data Flow for Semantic Queries .....</b>	8
<b>Figure 7.2-5. CML Query for Data in Artifactory .....</b>	10
<b>Figure 7.2-6. ECTo Layout and Views .....</b>	12
<b>Figure 7.2-7. ECTo Exploration Panes .....</b>	13
<b>Figure 7.2-8. Zulu (ECTo's 3D Visualization) .....</b>	14
<b>Figure 7.2-9. System Hierarchy and CML Panel .....</b>	15
<b>Figure 7.2-10. Supporting System and Component Panel .....</b>	16
<b>Figure 7.2-11. Hull Shaper Model .....</b>	17
<b>Figure 7.2-12. ESKER .....</b>	19
<b>Figure 7.2-13. Opening Block Dialog Box.....</b>	22
<b>Figure 7.2-14. AMIL Representation of Requirement Ranges of Values.....</b>	23
<b>Figure 7.2-15. Browsing Query Results .....</b>	25
<b>Figure 7.2-16. Relevant Commercial and Open Source Tools .....</b>	28
<b>Figure 7.2-17. A Generic Metric Composition .....</b>	30
<b>Figure 7.2-18. Total Weight Head Node Connectivity Computation.....</b>	31

Figure 7.2-19. Dashboard Structure in AMIL.....	33
Figure 7.2-20. Example Dashboard Configuration.....	34

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
AMIL	ARRoW Model Interconnection Language
API	Application Programmers Interface
AWS	Amazon Web Service
CAD	Computer Aided Design
CML	Component Model Library
DSE	Design Space Exploration
ECTo	Early Concepting Tool
ESKER	Expert-System Knowledgebase Evaluation Reasoner
FEA	Finite Element Analysis
GEAR	Generative Archetype Reasoning
IFV	Infantry Fighting Vehicle
JSON	JavaScript Object Notation
MD	Magicdraw
OWL	Web Ontology Language
PCC	Probabilistic Certificate of Correctness
RDF	Resource Description Format
SPARQL	Simple Protocol and RDF Query Language
SVN	Subversion
UML	Unified Modeling Language

## 7.2 Tool Design

This table documents META tool components that we have developed over the past year. Each of the components has met one or more needs outlined in the original META proposal. The underlying technology shows the innovative approaches that we have come up with to make the tools work well.

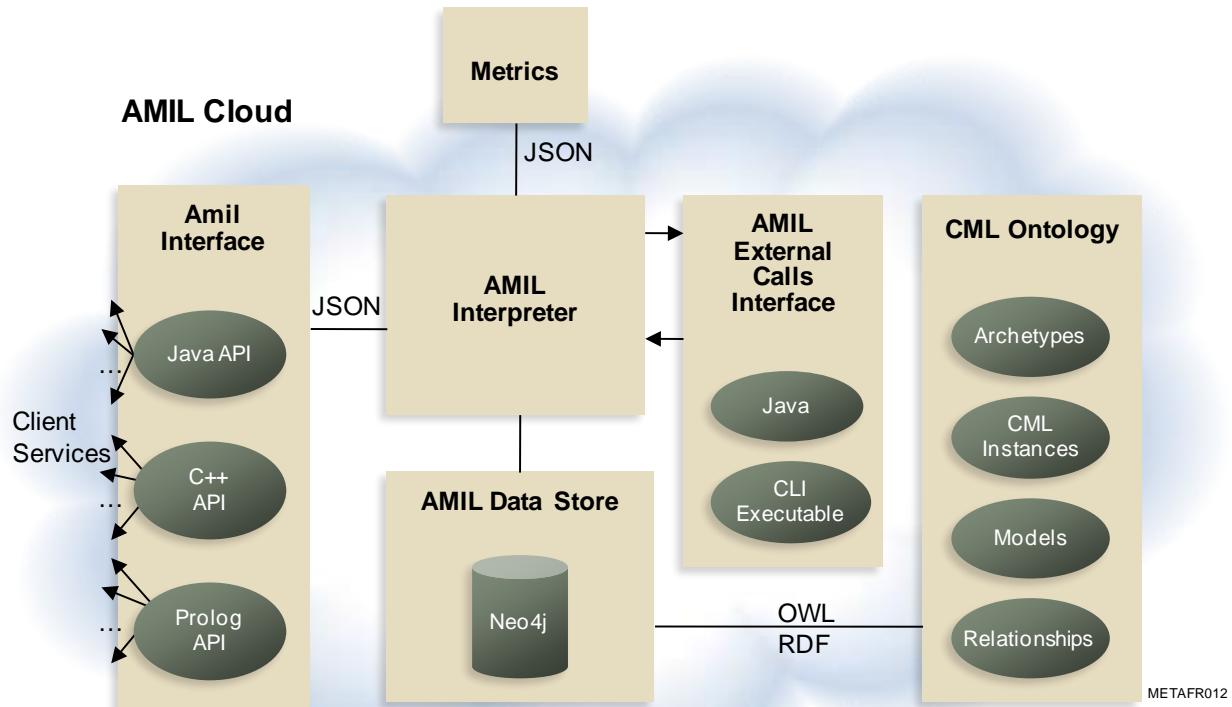
Tool Component	Meets Need	Underlying Technology	Demo
<b>ARRoW Integrated Development Environment (AIDE) Interface and Dashboard</b>	Demonstration capabilities, system launch and control	Eclipse/SpringHTML, , Maven, Subversion, Tomcat, Java, Metrics, more	Jan, Mar, May, Jul, Sep
<b>Metrics Suite</b>	Design progress, Complexity measures	Entropy-based methods, Design Curvature algorithms, contention models (BBN), (also see metrics docs)	Jan, Mar, May, Jul, Sep
<b>CAD (Pro/Engineer plug-in)</b>	Provide interface and access between ARRoW tools and Pro/Engineer	Pro/Engineer API, C++	Mar, May
<b>AMIL</b>	Heterogeneous model and tool interconnect	Graph database, Java/Prolog/C++ API, persistence and caching control, graph viewer	May, Jul
<b>Galileo T&amp;V</b>	Probabilistic Certificate of Correctness, Diagnostics, Language	Monte Carlo and Importance Sampling, Context Model PDF's and Ratio distributions, Reach-set Analysis (MIT), K-means clustering, Expert system	May, Jul
<b>ESKER</b>	Look-ahead and Design Space Exploration, Adaptability via set-based concurrent engineering, Levels of Abstraction, Language	Expert system state expansion and search, Rule-based design structure matrix, AMIL-aware, variable fidelity modeling, partial decomposition, subjective/qualitative rankings	May, Jul
<b>Envisioner</b>	Qualitative Simulation	Lisp	May
<b>SysML (MagicDraw plug-in)</b>	Requirements	MagicDraw API, AMIL-interconnected	May, Jul
<b>CML/Master Model</b>	Abstraction Control, Component Reuse	Ontology-based search, Maven/Artifactory delivery mechanism	Jul, Sep

Tool Component	Meets Need	Underlying Technology	Demo
ECTo	Vehicle-level concepting and prototyping	C++ object hierarchy of generic domain models	Jul, Sep
Metrics Infrastructure and Dashboard	Integrated metrics analysis and calculation services, role and interest-configurable graphical user interface	AMIL-integrated service architecture using Java Standard Object Notation (JSON text metric definition files)	Jul, Sep
Generative Archetype Reasoning (GEAR)	Synthesis, Component Reuse, Domain-specific reasoners (design, analysis, ...), Language	Semantic Web technologies such as OWL, Description Logic and Declarative Logic Programming, Lisp, SPARQL, Protégé	Sep
Cloud Deployment	Provide mechanism to support wide distribution and crowd participation	Amazon Cloud deployment mechanisms	Sep

## 7.2.1 AMIL

### 7.2.1.1 Introduction

The ARRoW Model Interconnection Language (AMIL) is used to represent the models that ARRoW works with—either directly, when the models are very abstract, or as a proxy, when the actual model is represented in a specialized tool like Simulink® or CREO™. More important, it represents the links among those models. AMIL is deployed as a web service, and provides a basic API for creating and manipulating nodes, links, and their properties; it is most useful to think of it as a network of models, rather than as a computer language. Java clients are expected to use the AmilLib wrapper library, which presents an object model of nodes and links, and hides the web service interactions; C++ clients like the Early Concepting Tool (ECTo) use a similar wrapper. The textual representation of AMIL data in JavaScript Object Notation (JSON) (Introducing JSON, 2011) or Resource Description Format (RDF) (World Wide Web Consortium, 2004), and the exact set of operations provided by the web service, are only of interest within ARRoW components. An overview of the AMIL structure is shown in Figure 7.2-1.



**Figure 7.2-1. AMIL Structure**

AMIL was not designed to represent every component that might be needed to build a fighting vehicle. Rather, it provides an abstraction that supports reasoning about basic structures and connections in a consistent way, without attempting to provide services for which robust, high-performance implementations are already widely available.

AMIL is built on the open-source graph database Neo4j ([neo4j.org](http://neo4j.org)). The implementation uses elements of Tinkerpop Blueprints ([tinkerpop.com](http://tinkerpop.com)) to support the importation and use of OWL (World Wide Web Consortium, 2004) to represent a formal ontology, and to support Simple Protocol and RDF Query Language (SPARQL) (World Wide Web Consortium, 2008) queries against the ontology.

In what follows, we will first describe AMIL’s basic structure, then the use of dynamic nodes to support metrics and the Component Model Library (CML). Finally, we will discuss the formal ontology embedded in AMIL’s repository, and its relationship to the other structures managed by AMIL.

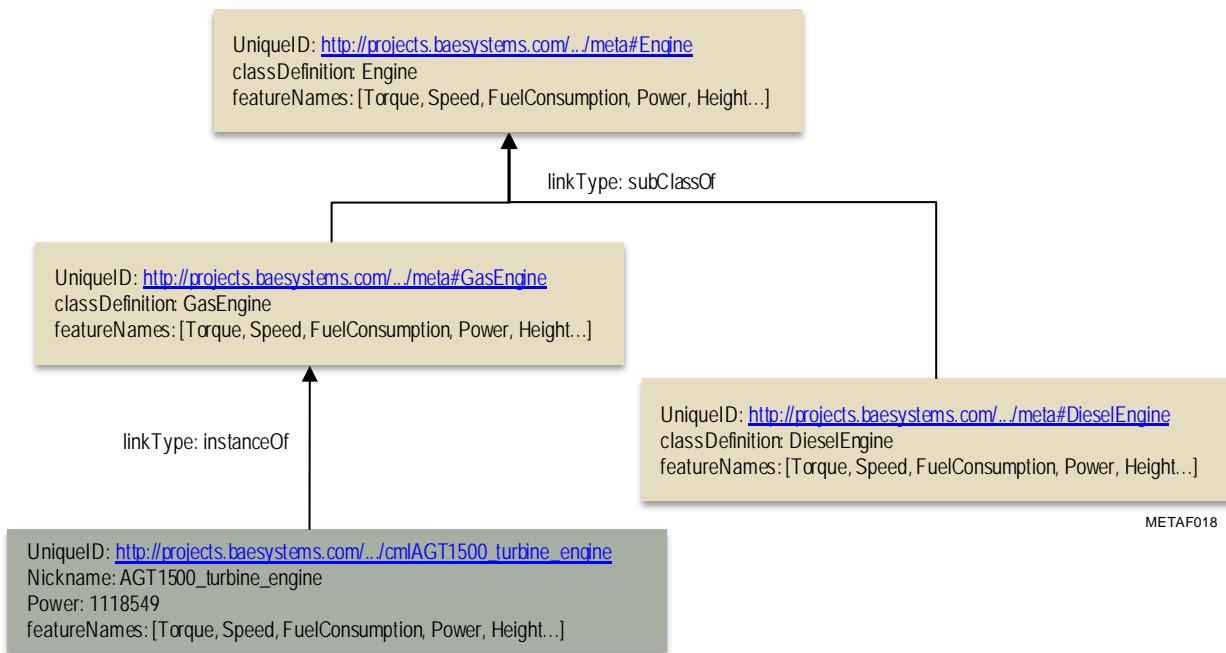
### 7.2.1.2 AMIL Structure

The AMIL repository contains nodes and links. It is not quite a conventional triple store, because both types of objects can contain an arbitrary number of named attributes, as opposed to the standard representation of RDF statements as SUBJECT-PREDICATE-OBJECT. As will be discussed below, the AMIL repository contains RDF in that form, with the more complex AMIL nodes and links overlaid on the RDF structures.

Within AMIL, there are two types of nodes: “immediate” and “dynamic,” described below. Either type can be accessed by name, by retrieving one of the ends of a link, or by query; both types are represented in the data store by a unique ID and an arbitrary set of named attributes, where the values can be strings, numbers, or arrays of strings or numbers. Links between nodes have an arbitrary type—new types are created on request—as well as their own sets of attributes. While nodes have a unique ID, the database key for a link consists of its type and the unique IDs of its start point and its end point.

Although nodes can have arbitrary sets of attributes, there are some attribute names that AMIL assigns specific meanings to. For example, any node associated with a class definition in the formal ontology will have a “classDefinition” attribute; the set of attributes associated with a particular class will be stored by AMIL in the “featureNames” attribute of the class node. These reserved names are documented more fully in the Software Users Manual. Similarly, certain link types may be given specific interpretations by AMIL in some contexts. For both nodes and links, the reserved names are used either to support dynamic nodes, or to support access to the ontology by AMIL clients. As shown in the following figure, and discussed more fully below, AMIL stores a compact representation of the ontology to allow it to hide the details of the database structure from its clients.

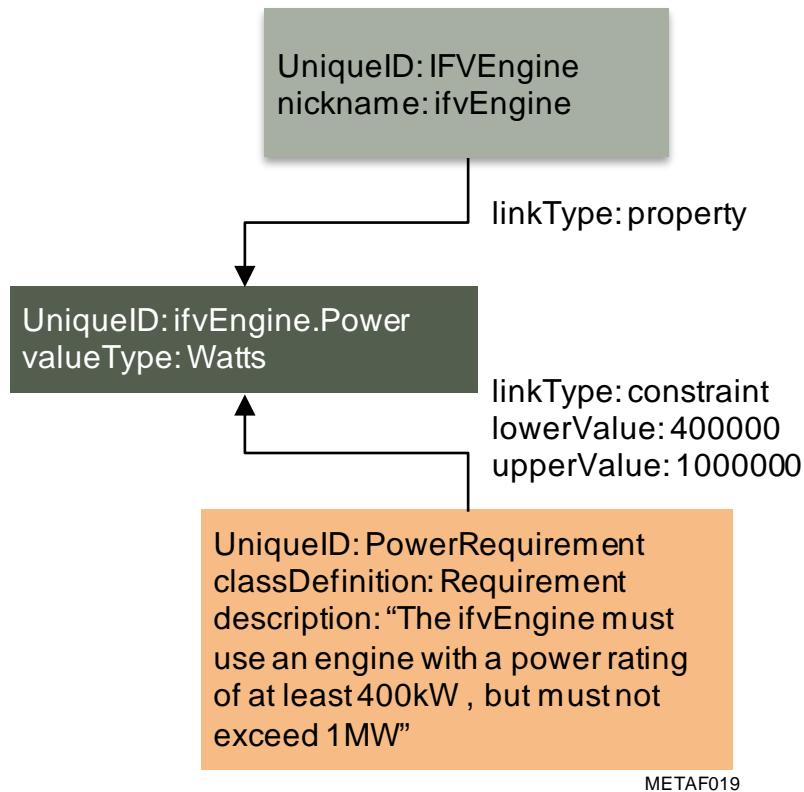
This shows a small piece of the ontology:



The three upper nodes represent the Engine class and two of its subclasses, which are connected to the parent by “subclassOf” links. The bottom node is a specific model of gas engine, tied to its class by an “instanceOf” link. The set of featureNames identifies specific items that are declared in the ontology, so will be available for semantic queries; the engine’s power rating is one.

Archetypes, requirements, metrics, and components in the CML are all represented by similar groups of nodes and links, with their own specific link types and node attributes. Additional subsystems can easily be supported.

Requirement authoring in SysML is an example of an AMIL client application that capitalizes on the ability to create arbitrary link types and properties in AMIL, as shown in Figure 7.2-2.



**Figure 7.2-2. Example Requirement in AMIL**

As discussed previously, requirements are an important concept in development with META. In order to allow requirements to effectively drive design space exploration, authoring a requirement in SysML creates a new type of link, "constraint", to carry relevant constraint data, as well as creating its own set of attribute names to carry requirement-specific information.

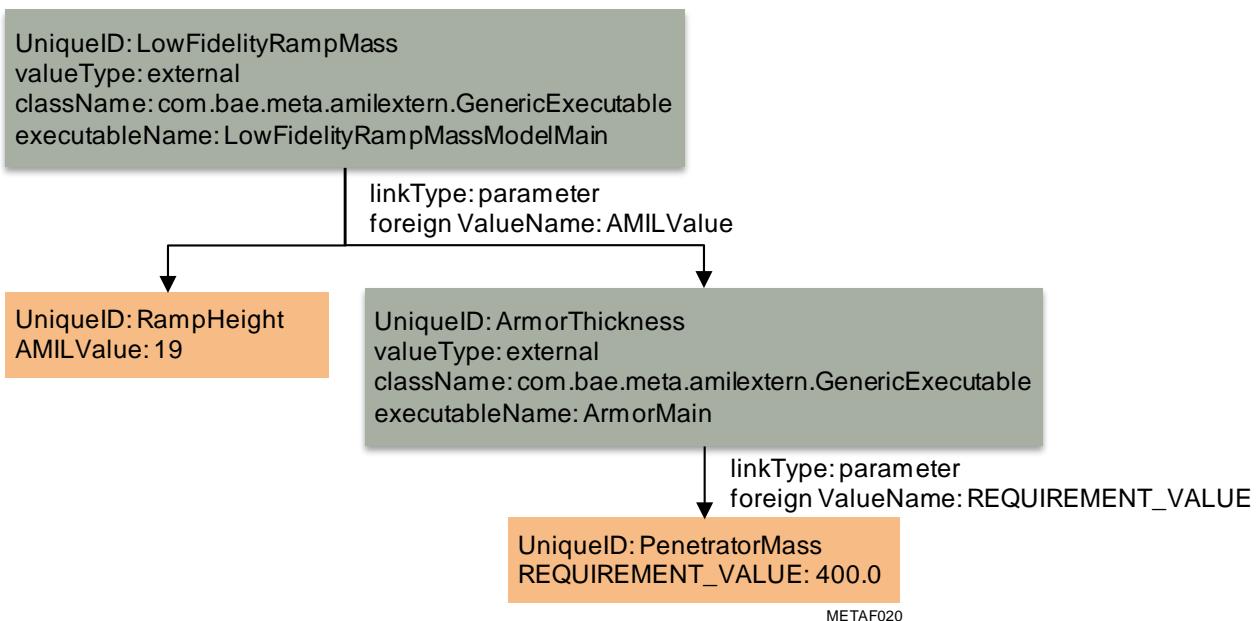
### 7.2.1.3 Dynamic Nodes

The nodes shown so far are all “immediate”: when an immediate node is retrieved by an AMIL client, the client simply receives a representation of the name-value pairs associated with the node. Many of the things represented in the AMIL graph are executable models; dynamic nodes allow these executable models to be invoked transparently by any AMIL client. The code associated with a dynamic node is executed on the server, so there is no need for additional client software installation, and the results of the execution are presented as a set of name-value pairs, just as if the node were immediate.

Dynamic nodes are stored in the repository with a set of named attributes. The “valueType” attribute either labels the node as immediate, or identifies a Java class accessible to the server that will be used to evaluate the node. The set of such classes is extensible, but not dynamic; changing it requires a server restart. The attribute set from the repository is passed to the evaluation class, which may then do whatever is needed to return the node’s values: it can

perform a straight Java computation, but it can also run an executable, invoke a web service, or return the contents of a file.

Figure 7.2-3 shows a subset of the nodes and links associated with an executable model to estimate the mass of the exit ramp on an infantry fighting vehicle (IFV), based on the ramp's dimensions and the survivability requirements of the IFV. The ramp mass model itself is a legacy C program; the valueType property “external” tells AMIL that this node will be evaluated by a class that allows the execution of arbitrary external code; there are more structured evaluators for metrics and the component model library. The evaluation class, in turn, will use the GenericExecutable class to run the legacy C code, as specified in other attributes of the node.



**Figure 7.2-3. Fragment of Executable Model**

The model's parameters will, in this case, be passed to it on the command line. The parameters are identified by links from the node representing the model to other nodes in the graph: the ramp's height, in this case, is stored as a simple value. The “parameter” link from the LowFidelityRampMass node causes that value to be pulled in for the model. The “ArmorThickness” parameter, on the right, is a little more complicated: in this case, ArmorThickness itself is another executable model, which in turn requires its own set of parameters. The parameter shown, PenetratorMass, was published from a SysML application as a requirement.

When a client application retrieves the LowFidelityRampMass node, AMIL begins by invoking its “external” node type handler, which then invokes the GenericExecutable class. Each of these invocations has full access to AMIL, of course; as GenericExecutable gathers the parameters for the executable, it retrieves the “ArmorThickness” node, which causes a recursive invocation of GenericExecutable. The second invocation retrieves the parameters for that the armor thickness model, runs it, and returns an attribute set, one of whose members is “AMILValue.” The first invocation of GenericExecutable can then proceed.

As described, this can be a very expensive process. Each of the executable models could take a very long time to execute, and the chain of models required to evaluate a given node is potentially quite large. Although there has been no need to implement it in the current system, there is nothing to prevent caching of dynamic node evaluations, either in the server’s memory or in the database: the handler for the node has full access to the database, including the ability to modify existing nodes and create new ones.

AMIL is central to the implementation of metrics and to the implementation of the prototype CML. For metrics, each metric definition is a dynamic node, of type “metric,” which can be retrieved with parameters that specify the model for which the metric is to be computed; the code has full access to the AMIL graph, so can retrieve parameters from other nodes, or run arbitrary external computations to determine metric values. For the CML, the bulk of the data is stored externally to AMIL; an index is maintained in AMIL to facilitate searches against the CML’s content—for example, find all engines in the CML with a power output greater than 350 kw. AMIL was not designed to store all of the details of those engines, which might include full Computer-Aided Design (CAD) drawings, very precise thermal models, torque curves, and so on; rather, it helps client applications identify the appropriate models to retrieve from the CML repository, and provides the coordinates within the repository for those models.

#### **7.2.1.4      Ontology**

AMIL and AmilLib provide some very basic search capabilities, and others can be added. A primary requirement is to search highly structured, semantically rich data associated with models in the component model library. We also have a need to represent archetypical structures, where a component of the archetype might start out as “Engine,” and eventually be refined to a specific model of a specific brand of gas turbine engine, based on size, power, thermal characteristics, and so on. This kind of search is best supported by using a formal ontology, which allows queries based on the meaning of the data. A free-text search, or even a Google-style search, would be much too imprecise, missing matches because of differing vocabulary terms or misspellings, or returning spurious matches because it couldn’t match numerical size, weight, and power (SWAP) parameters.

At database initialization time, AMIL loads a predefined set of OWL files to populate the ontology. These include class definitions as well as instance definitions, which, as discussed in Section 7.2.2.2, act as an index for the Component Model Library. The nodes and edges created by this process are stored in the same repository as the rest of the AMIL database, but the form of the data is rather different: where AMIL supports arbitrary, extensible sets of attributes on nodes and edges, the ontology has a well-defined and very small set of link types and attributes. In AMIL’s model, the weight of a component would be stored as a “Weight” attribute on the node representing the component; in the ontology, the weight is stored as the “value” attribute of a node that can be reached from the component node by following links of specific types.

In order to support both models, AMIL post-processes the ontology during database initialization, in order to identify all of the classes, instances, and *features*—which correspond to AMIL’s attributes—that were created. Using this information, AMIL adds new links, and adds new attributes to class and instance definition nodes; at the end, a component’s weight will be stored both as an attribute of the component node, for AMIL clients, and as a feature associated with the node, for ontology searches.

Because AMIL is the only thing accessing the repository, it is feasible for it to maintain consistency despite all of the duplicated data. When an AMIL client changes an attribute that

was defined by the ontology, AMIL can find and update the feature value as well; if an AMIL client creates a new instance of an ontology class, AMIL will create all the required attributes and links to support its full use.

The ontology exists primarily to be searched; searches are supported using the SPARQL query language. Queries can find instances of particular classes, and can filter based on instance features; thus, a simple request like “find all engines whose weight is less than 250 kg and whose power output is greater than 300 kW” is easily expressed, and can be evaluated very efficiently. Figure 7.2-4 shows a snippet of the formal ontology, its representation in AMIL, and a query to retrieve some elements defined by the ontology.

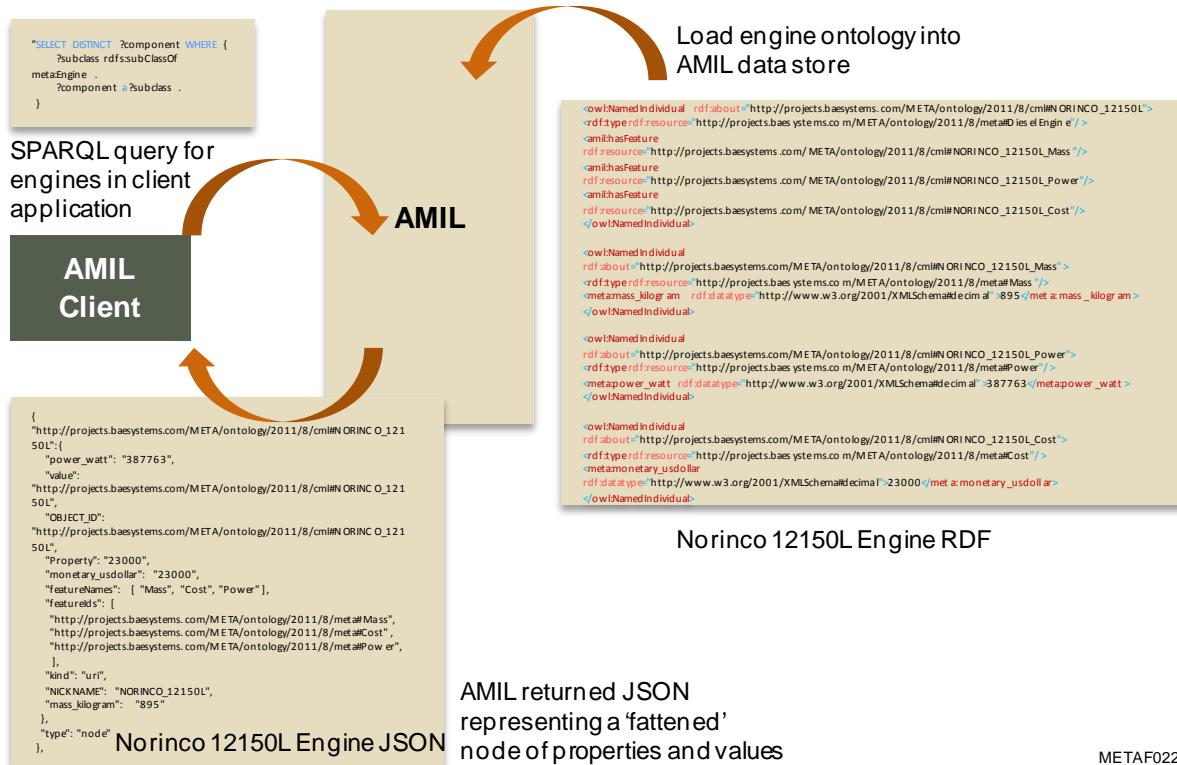


Figure 7.2-4. Data Flow for Semantic Queries

## 7.2.2 Component Model Library

The Component Model Library (CML) has two primary purposes. First, it is a repository that stores technological knowledge and facilitates its sharing and communication between work threads and components. Second, it encourages re-use of artifacts and makes it easy to do so in a reliable and consistent manner. A centralized component library supports distributed design, because it is available anywhere, and facilitates design evolution, because it is always available.

### 7.2.2.1 CML Architecture

#### Maven repository managers

The CML semantics borrows heavily from the semantics Apache Maven<sup>[MVN11]</sup> uses to reference software artifacts and identify dependencies. The overlap allows us to leverage existing Maven tools to build the CML infrastructure. The CML uses a Maven repository

manager as a back-end store for components and models. Repository managers generally provide an interface between developers and online repositories; for the CML, Artifactory [ART11] acts as a content management system for storing resources associated with a given component or model (e.g., MagicDraw files, Matlab files, photos, videos, etc.). The following repository managers are popular in the Maven community: Nexus [NXZ11], Artifactory, and Apache Archiva [ARC11]. Artifactory was selected for its stability, usability, and active user/developer community. It is available in several distributions with varying capabilities and license restrictions; for the CML, the free open source version is sufficient.

#### *Artifactory feature set*

Despite its novel application in CML, Artifactory addresses several concerns out of the box:

1. User management – restricting and permitting access to CML
2. REST [RST11] interface – providing programmatic access to CML resources
3. Web GUI – providing an easy-to-use browser interface to CML resources
4. System logging – monitoring CML usage
5. Backups – data loss prevention
6. Dependency management – maintenance of artifact dependency trees

#### *Artifactory usage in CML*

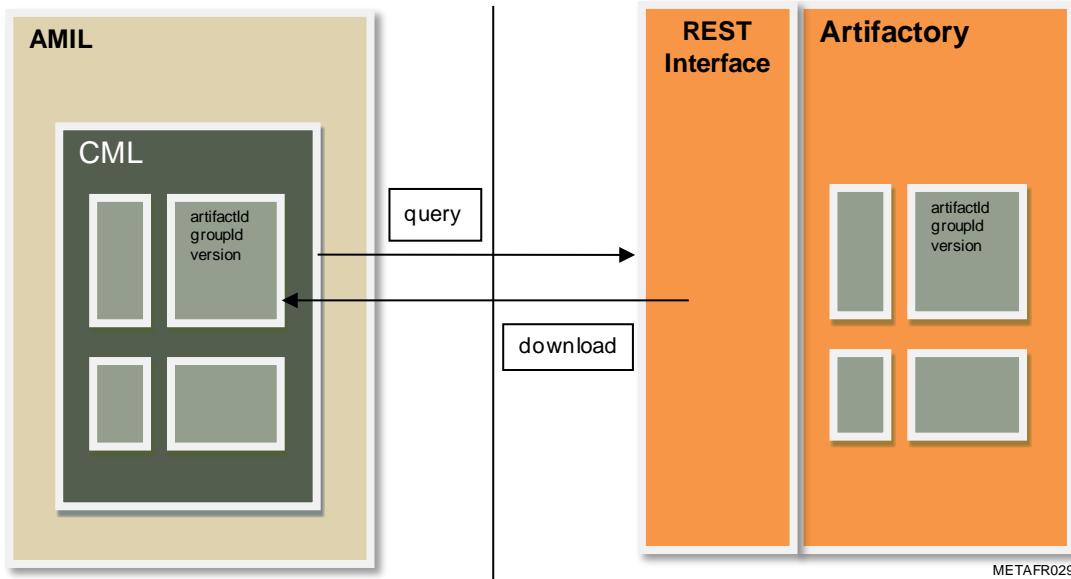
In CML design, Artifactory saves bandwidth by delaying the transfer of large data files into the ARRoW workspace. For example, the user might search for an engine with a given set of parameters, and select a single engine from the search results. Not until the user needs to use the data files associated with that engine will they be brought into the local cache by Artifactory.

Artifacts—CAD drawings, spec sheets, executable models, and so on—can be uploaded to the Maven repository via Maven’s command-line interface or through Artifactory’s web interface. Artifact versions are categorized as snapshots or releases: a release artifact will not change, while a snapshot artifact can change. Snapshots and releases are uploaded to separate locations in the Maven repository.

AMIL serves as the index for the CML, allowing semantic searches to retrieve the metadata for a specific component model. The metadata includes the Maven “coordinates,”<sup>1</sup> which the AMIL “cml” dynamic node type will use to retrieve artifacts from Maven via Artifactory’s REST interface (Figure 7.2-5). A traditional Maven repository would set the packaging coordinate to be an archive format such as jar, war, or zip; CML overloads it to be a more flexible file extension setting to account for the wide variety of system engineering file formats. Artifactory provides a basic search capability as well, but it is focused on retrieval of versions of software packages, so is not easily extended to support the queries required for system design.

---

<sup>1</sup> The Maven coordinates are group ID, artifact ID, version, and packaging. For example, com.tinkerpop.blueprints is the group ID for all of the Tinkerpop code used to support the AMIL ontology; an artifact ID is blueprint-neo4j-graph, the library for interfacing Tinkerpop to neo4j; a version would be 1.0-BAE, and packaging would be jar. These four values uniquely identify the library version in the world of Maven; a copy of the library can be cached locally, or retrieved from any Maven repository that has it, without further thought.



**Figure 7.2-5. CML Query for Data in Artifactory**

### 7.2.2.2 CML Search

Key to using the CML is the ability to search for artifacts that met specific needs. In cyber-physical design three major use cases for CML search are in design space exploration, analytical compositions and design verification.

#### *Design space exploration*

In META, a design starts with a set of requirements, which are essentially constraints on the design space. In design space exploration, we can use these constraints to limit the space of possible component configurations. However, it is often the case that design requirements are contradictory, so the problem cannot be solved automatically, for example by using constraint-based solvers. The goal instead is let designers use their domain knowledge, providing support through the CML's search facility. For example, the size of the engine compartment for a vehicle might already be known, but performance requirements require an engine with particular capabilities, whether in terms of torque, power output or fuel efficiency. At this point a designer can search for engines in the CML that can meet the space and performance requirements. If no results are returned, then it is clear that there are problems with the requirements and that the design is over-constrained, unless developing a new engine is within scope. If results are returned, the engineer can select one, or, if his design software supports it, retain all matching engines as candidates until subsequent requirements, such as weight claims, allow the set to be further reduced.

#### *Analytical Compositions*

System designers regularly perform repetitive sets of analysis in the course of system design, often as part of test and verification of a design. These analyses consist of a common set of tools and methods applied in an analytic workflow, with the design data flowing from one analytical tool to the next. There are a wide variety of design modeling methods available to engineers (CAD, Finite Element Analysis [FEA], Thermal, etc). There are also a number of general purpose modeling tools that have a wide user base, such as Simulink and OpenModelica. For an engineer, finding the most appropriate model to use in a particular workflow is often a time

consuming task. A CML query service can aid them by helping them find design models which meet their requirements, in terms of input and output parameters, or in terms of what type of analysis is to be performed.

#### *Design verification*

In design verification, the design task (which is a synthetic task) has been completed to some level of detail, and the goal is to analyze the design to confirm that it still fulfills all the requirements and design assumptions. Perhaps later on in a design, a weight claim for the drivetrain was imposed that made the current engine choice inappropriate. At this point the engineer can perform another search of the CML with the updated set of engine requirements. Tradeoffs might have to be made at the margins—relaxing one requirement slightly could allow for a consistent design.

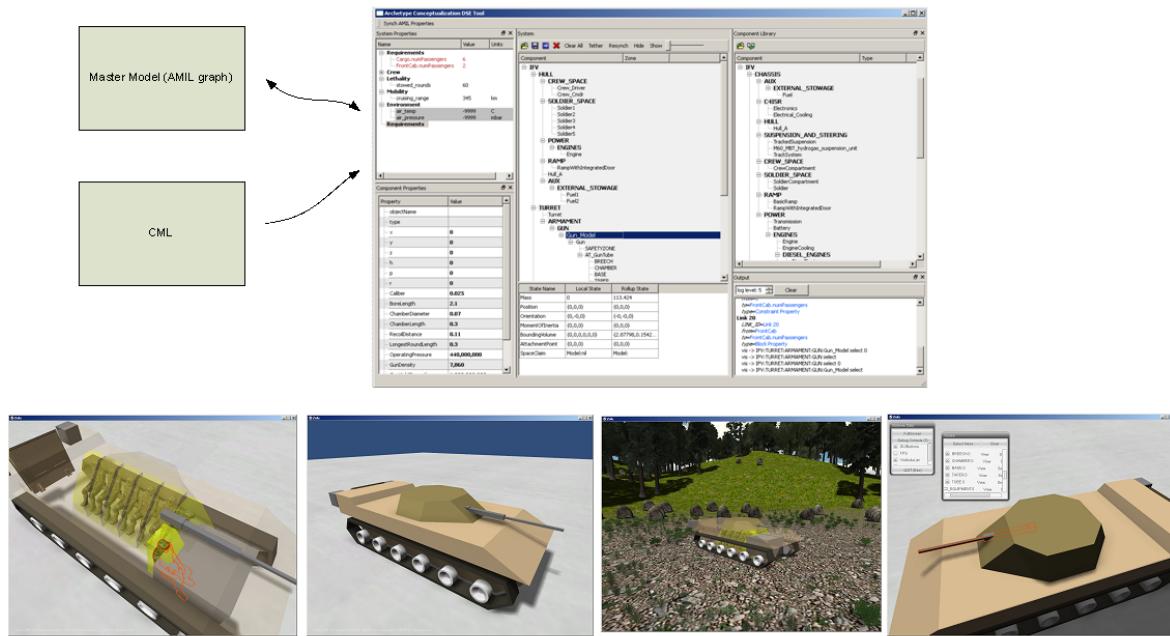
As discussed in Section 7.2.1.4, AMIL provides a formal ontology and supports semantic searches of it using a standard language. We believe this allows the best combination of high precision and expressive power in queries, which are critical to the use of the CML. A goal of future work would be to provide support both in the AMIL/CML API and in the user interface for the types of searches supported by SPARQL, without requiring knowledge of its relatively difficult syntax, or knowledge of the relatively complex naming conventions that are used in OWL ontologies.

### 7.2.3 Conceptualization – The Early Concepting Tool

The Early Concepting Tool (ECTo) guides a vehicle design process by applying abstract components that fit together *a priori* as a result of applying archetypal rules. Any down-select process that excludes incompatible components is made possible by doing DSE as a successor stage. The ECTo concepting reasoner captures spatial representations, complex space claims, and articulations, which are difficult to represent and reason with in pure logic.

As a system design tool ECTo enables editing of a master model primarily through the hierarchical assembly and manipulation of components from the CML. It is focused primarily on empowering a designer in the early design phase to be able to incorporate and manipulate major design drivers and rapidly assess the qualities of system concepts. The resultant concepts can be used as the basis for more detailed design.

The ECTo includes a 3D viewer called Zulu to represent the vehicle design concept and to aid in initial spatial layout and rudimentary packaging without the burden of a commercial CAD tool. ECTo was developed as a tool which could be used independently or that can fit naturally into the ARRoW toolchain and interact closely with the projects AMIL graph. Figure 7.2-6 highlights the key layout of the tool.



**Figure 7.2-6. ECTo Layout and Views**

The ECTo’s logic reasoner works on vehicle archetypes at specific levels of modular abstraction, incorporating encoded knowledge of interactions and design rules of how the modules fit together. The intermodule influence diagrams at block levels are selected from composites of lower level archetypes. The first pass that ECTo makes is at the conceptual stage, in which we can then realize a concept system<sup>2</sup>.

**Synthesis.** Synthesis at the concepting level is the art of realization based on merging intent (requirements) with possible embodiments (components from the model library). ECTo synthesizes a numerical subsystem/component space and weight claim topology with a visual representation and an accompanying AMIL blueprint that another tool or engineer can use. The design may not be complete either—as when alternative design options are available, and an identifier tag can be used to indicate sets of components which can provide inputs to another decision support system.

<sup>2</sup> ECTo thus creates “ectypes” which the conceptual realization or instances of a vehicle archetype

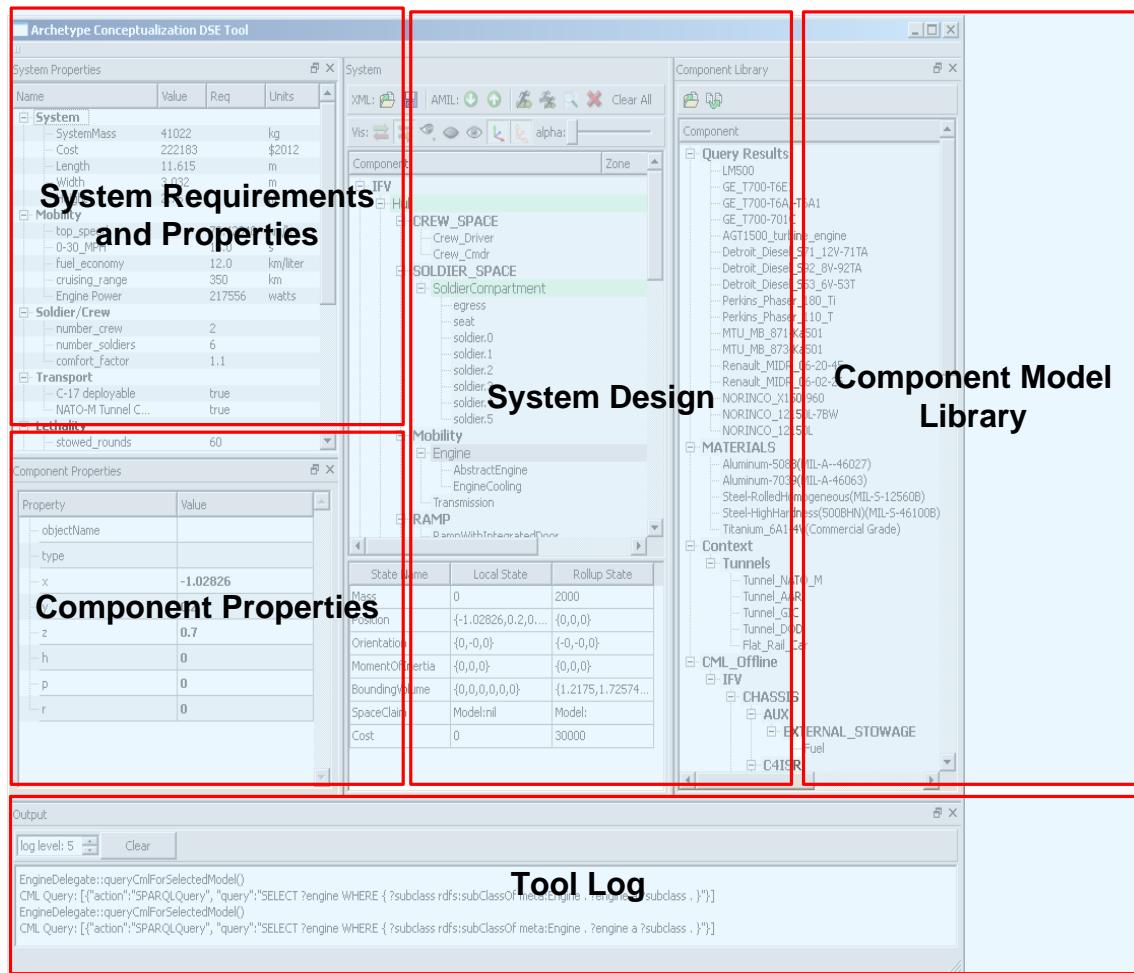


Figure 7.2-7. ECTo Exploration Panes

### 7.2.3.1 ECTo Architecture

The overall architecture of ECTo is based on the premise of editing a master model primarily through the hierarchical assembly and manipulation of components from the CML. To this end, both system design data and CML data can be stored or retrieved interchangeably using either local XML files or the Arrow Web Services AMIL graph. The interface between ECTo and the Arrow Web Services is intended to be very transparent and provides an example for how a design tool would interface with these services.

The ECTo is written in C++ and build on top of the Qt application framework using Visual Studio. The ECTo uses a C++/Qt based AMIL client library to facilitate all its interactions with the AMIL graph. Zulu is built using the Unity Engine and communicates with ECTo using UDP messages.

**ECTo Main Window.** ECTo is made up of several configurable panels with the candidate system as shown Figure 7.2-7 in a typical layout.

**Zulu.** The interactive 3D visualization component to ECTo, called Zulu, runs as a separate process. No state is stored in the visualization and it is not required to be run for ECTo to function, though this provides the easiest way to manipulate components and generate a spatial view of the system. Figure 7.2-8 illustrates the Visualization capability.

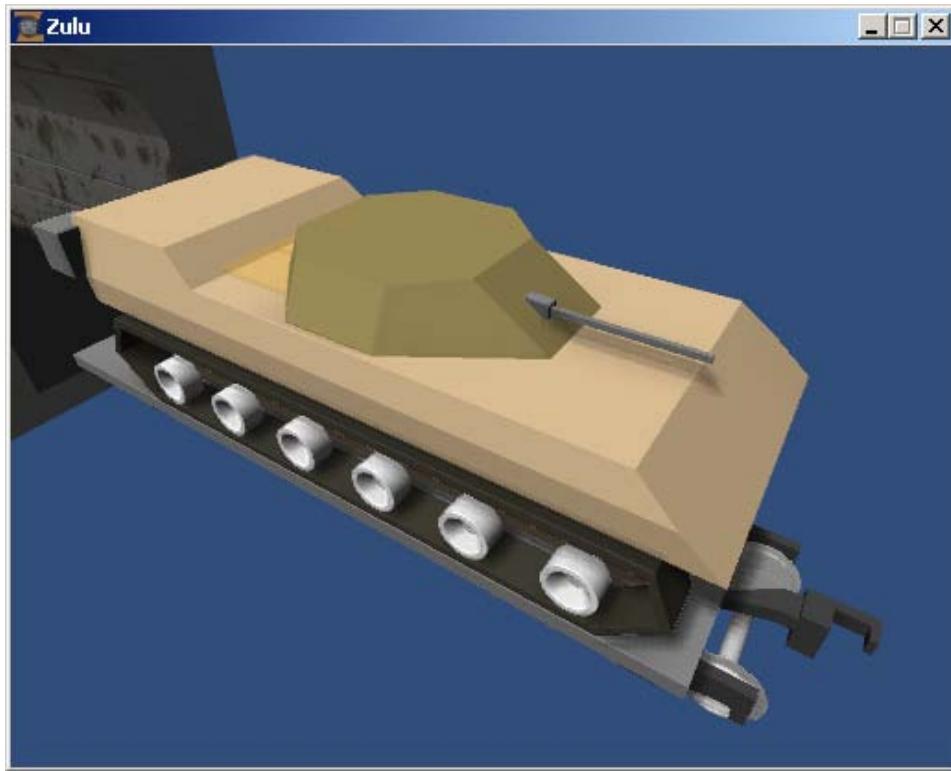
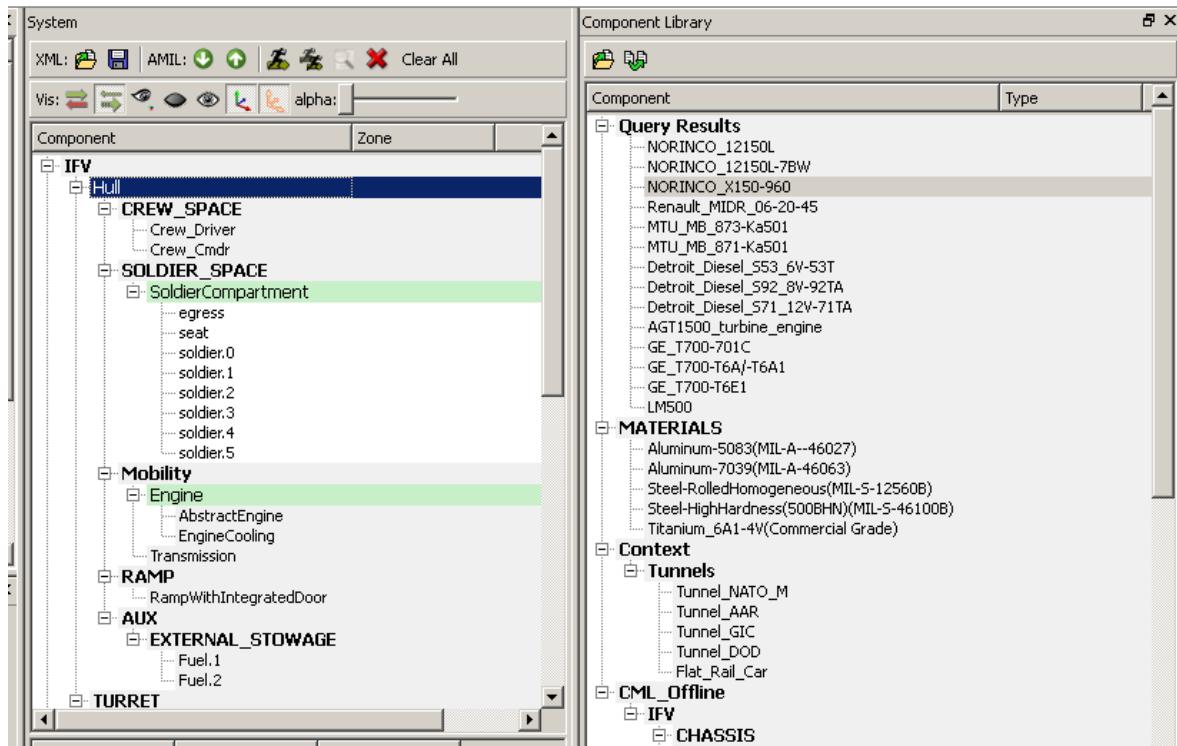


Figure 7.2-8. Zulu (ECTo's 3D Visualization)

**System Design Toolbar.** ECTo maintains a list of all the elements it is operating with so it is meant to work as an exploration tool with its own private set of data. This means that ECTo essentially assumes it owns the system hierarchy while it is editing it. For checking design consistency, models can be run explicitly by selecting a model to execute or by executing all models. Refer to the ECTo user's manual for more information.

Query CML creates a SPARQL query for refinements of a selected component and queries for items in the ontology that are valid refinements of abstract item selected.

**System Hierarchy.** The hierarchical representation of the system is displayed in the System Design panel. The values of the major states are displayed in a table. The 'Local State' column is the state values for *only the component selected*. The 'Rollup State' column displays a rollup of values for the selected state and all that state's children. System level rollups for Mass, Cost, Length, Width, and Height are always displayed in the System Properties panel.



**Figure 7.2-9. System Hierarchy and CML Panel**

**System Requirements and Properties Panel.** This panel stores all the system level properties or any properties that need to be exchanged between component models. The ‘Value’ column indicates the current systems estimated performance and the ‘Req’ column is used to maintain the system requirement or derived requirements for that property. Most key system inputs are considered requirements but they can be adjusted by a designer so they can assess how particular inputs drive a design. Figure 7.2-10 shows the supporting system and component panel.

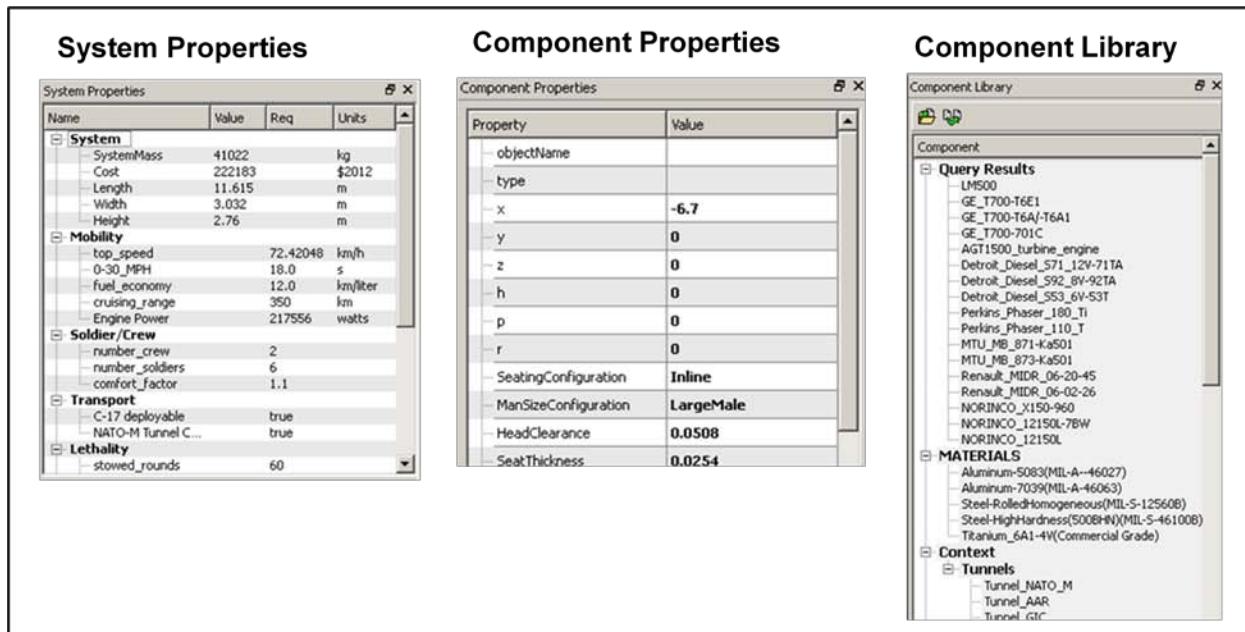


Figure 7.2-10. Supporting System and Component Panel

**Component Panel.** This panel shows and allows one to edit the properties of a specific component. Components with associated models will frequently have additional properties that are specific to that model. For example, the SoldierCompartment model uses additional properties to construct the soldier compartment such as seating arrangement, squad size assumptions, clearances, etc.

**Component Model Library Panel.** The CML panel shows components from the CML which can be included into the system hierarchy by dragging a component from this panel to the parent in the system you want to attach this component to.

Components can be loaded into this panel either through XML files using or as a result of a query to the CML service in Arrow Web Services.

As additional ways to interact with and search a CML are established, this panel can evolve to support additional approaches. Additionally, CML repositories should be able to easily export indexes of components into a compatible XML file. The file CML.xml in ECTo's working directory provides an example of various CML components used to build up the IFV in the demo walkthrough below.

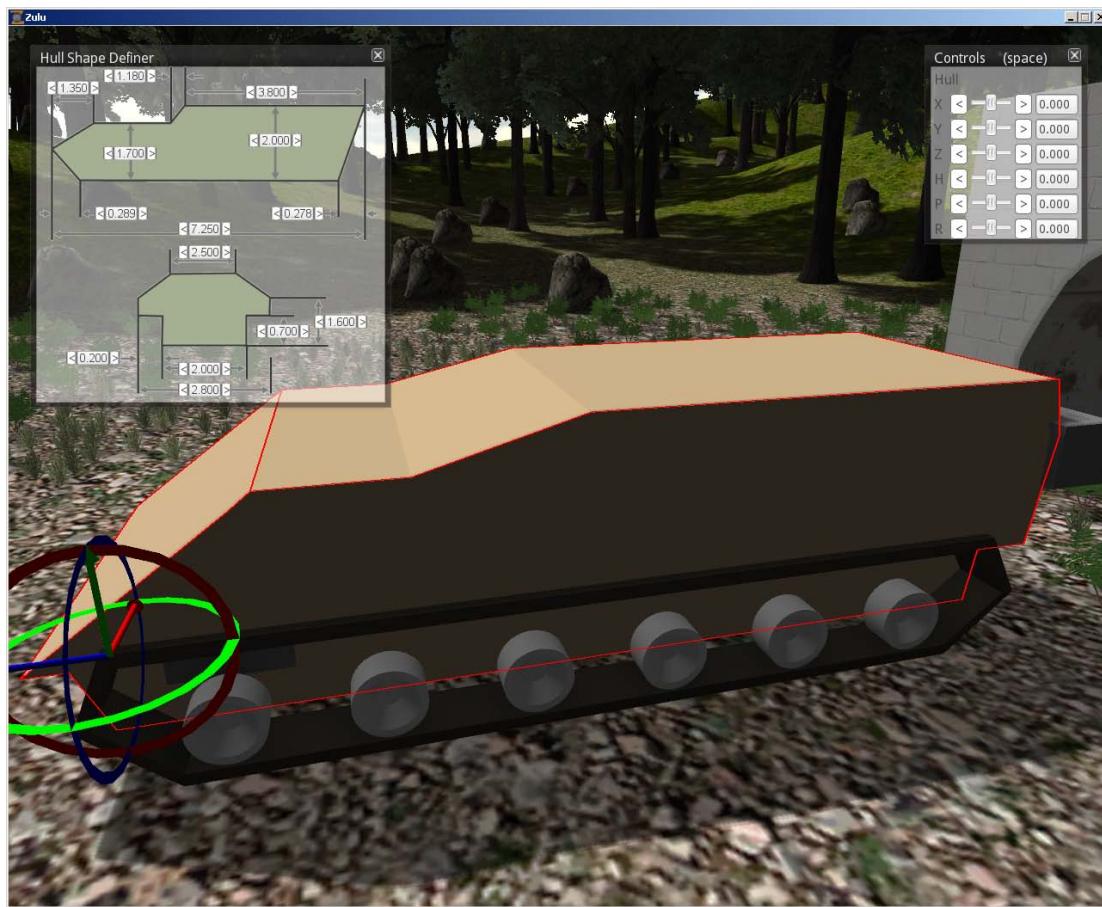
### 7.2.3.2 ECTo Models

Certain nodes, indicated by having a green background in the System Hierarchy, have Models associated with them. These are typically parametrically driven components or systems that change their properties or are ‘built’ based on inputs to those models. For example the ‘SoldierCompartment’ Model, when executed takes the requirement from the System Properties panel for ‘number\_soldiers’ and various other inputs in the Component Properties panel to construct the space claim and mass for a the Soldier Compartment. If the number\_soldiers requirement is changed and run, this model will see the number of soldiers and the various properties of the compartment, bench, egress volumes, etc., change to reflect this.

The ECTo models are run from the System Design Toolbar, either individually on a selected model or by executing all models in the system. Running all models traverses the tree and

executes models from children models up, models under the sub-tree of another model node are executed before the ancestor node's model. Executing Models can retrieve and set data in the System Properties widget as they run, so one model's output can be used as input or modified by another model. This data flow can also be used to control the execution order of models or to establish an implicit causal network of models.

Another different kind of model included in ECTo is the Hull Shaper model. If you select the 'Hull' Model component, you should see a Hull Shaper UI popup in the Zulu visualization. By modifying these fourteen dimensions a designer can capture the essential shape of most traditional combat vehicles. As the Hull is modified, estimates of weight are calculated and cost if calculated using the cost per pound input parameter. If this model is insufficient to represent a design, a different hull model or component can be used in its place. Figure 7.2-11 shows a hull shaper model.



**Figure 7.2-11. Hull Shaper Model**

In the System Properties panel, a quick look calculation is made as modifications occur to generate a minimum Engine Power that would be required for a vehicle of this weight class to achieve the required maximum speed (as shown in the System Properties as top\_speed). If the system is modified to make it heavier, the required Engine Power will go up. Additionally, if the top\_speed requirement is modified in the System Property panel, this will also change the minimum required Engine Power. This calculated, or derived, requirement for minimum Engine Power is used when querying for viable refinements of Engine.

One way to quickly change the weight of the vehicle is to modify the ‘AvgThickness’ component property for the ‘Hull’ Model component. If the average hull thickness is changed from 0.02 m to 0.03 and the Hull model is run, the new model will represent a heavier armored vehicle.

**Query for viable engines in CML.** If we selected an ‘Engine’ model node in system hierarchy, then we can ‘Query CML for Candidate Component Refinements’, and observe a SPARQL query in the Output Log panel. The Query Results is added to the CML panel, and only engines returned that exceed the minimum Engine Power calculated in the System Property panel.

**Replace abstract engine with specific engine.** Delete an ‘AbstractEngine’ and replace it with a concrete engine from the library.

### Path Forward

ECTo is a prototype of an early system conception and analysis tool and has the potential to evolve into a very powerful system engineering asset.

### Architectural modifications

- Migrate IFV specific delegates and models into dynamically loadable libraries, these libraries can then be stored and brought into ECTo from the CML.
- Include Ability to do multidimensional sweeps across input values or enumerated sets. Add output tables and data export utilities to facilitate analysis of this data.
- As statistical models are incorporated allow for execution of Monte Carlo runs.
- Fully incorporate a Design Set node that can be used to represent alternatives and any level in the hierarchy and tools.

Enhance the System Properties panel to allow for a designer to adjust values and still maintain original requirements perspective. Add a third column for actual requirement, a column for value to use as system input and a value for actual estimated output.

#### 7.2.4 Co-Analysis and Exploration

This section summarizes Generic Ensemble (GEAR) and Expert-System Knowledgebase Evaluation Reasoner (ESKER). For more details and examples of use, refer to Appendix 7.6.

##### 7.2.4.1 Principles Behind GEAR

The idea of aligning archetypes with reasoners leads to the term GEAR to describe these capabilities. Reasoners, patterns, templates, design rules, and archetypes are essentially synonyms for this generic capability. The prominent idea behind GEAR is to apply similar rule-based semantics in the context of developing archetypes for analysis, design, and implementation. The goal is to extend the information laid out in AMIL and leave it in a

symbolic format, suitable for mapping into more concrete representations. The symbolic representation thus forms an “archetype” for the specified behavior.

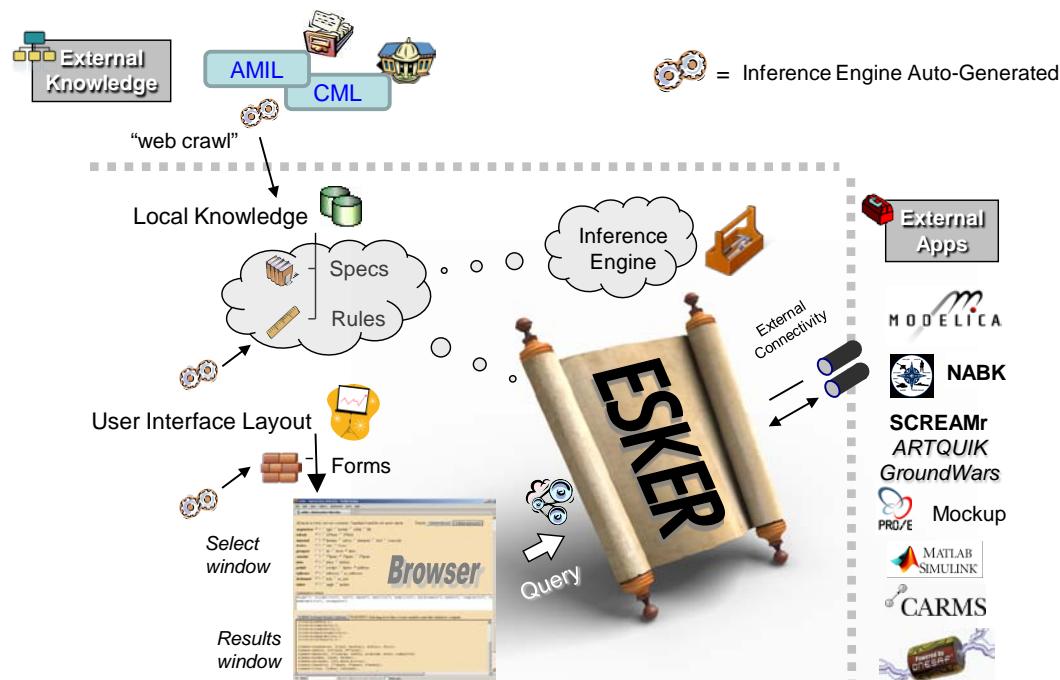
The first step is to start out with a domain model of some sequence (potentially concurrent) of steps that may build into a cyber-physical realization. This sequence is referred to as a plan, a use case, a scenario, a thread, or any term providing essentially a narrative description. The main connecting theme is that it forms a set of behaviors that would typically reproduce a human’s action (automation) or improve on some already automated realization. In practice, these sequences draw from typical or archetypal behaviors that have stood the test of time. The key is to not reinvent the wheel each time the engineering development process needs to implement a behavior. Instead behavioral recipes can be extracted from the repository and applied to a start-up design task that reduces development time. This requires a description that can generate concrete realizations based on the behavioral archetypes and requirements.

See Appendix 7.6 for a detailed treatment of GEAR.

#### 7.2.4.2 ESKER

The ESKER is the Expert-System Knowledgebase Evaluation Reasoner tool for design space exploration which ties together AMIL and logical semantic reasoning to facilitate Design Space Exploration (DSE). ESKER contains the engine that drives the search. The semantic web reasoners available use a similar inference engine (Prolog). ESKER also uses a declarative form, making it very compatible with triple-store and description logic.

The ESKER evaluates utility criteria for a given set of components selected from a *set of* variants. We initially assume that the model components would fit together; a precursor archetypal model actually establishes the specification for components that *can* get integrated together, which is also what ECTo does from vehicle structural design rules.



**Figure 7.2-12. ESKER**

System optimization has historically remained a challenging problem because the complexity involved in simply choosing between alternatives of any significant number makes a purely quantitative approach prohibitive. Although algorithmic automation approach can alleviate the bookkeeping, several challenges remain, especially in terms of integrating results from a set of tools that provide the intermediate decision support.

**Concepting and Design Phases.** The general statement of the problem is concisely framed in a basic two-dimensional design space. The scenario typically occurs with the design of any sufficiently detailed product, such as a ground vehicle or a weapon system and it involves selecting alternatives with respect to some set of criteria. Within the first dimension, a set of concept or design alternatives exists. Some examples may include:

- Capacity of vehicle in terms of different count of troops
- Tracked vs. Wheeled
- Gun Caliber
- Engine Type
- *Etc.*

In the second dimension, a set of optimization criteria exists to arrive at the best choice of element alternatives. The criteria can have various requirements and constraints associated with their description and typically fall into a set of established categories, such as:

- Cost
- Reliability
- Performance
- Weight
- *Etc.*

The system engineering puzzle is to choose which alternatives fit best together within a given set of criteria. The major difficulty in doing this from a *global* perspective is that both the product and optimization categories cross a broad spectrum of disciplines and will likely integrate a number of disciplines and analysis tools together to provide the most effective solution. That is the nature of system engineering, and why a cross-disciplinary approach is vital.

The results of this implementation show that an expert system backed by a dynamic knowledge base is well suited for the optimization task. These objectives can provide a formal mechanism to rationalizing the engineering decisions made:

- Declarative Knowledge
- Structured Decisions
- Human still in the loop
- Generate a narrative for explanation and regression (i.e., a *provenance* capability)

**A search optimization problem.** The problem boils down to optimizing among the alternatives considering constraints, requirements, and various measures of effectiveness. Most of these measures either come about through heuristics, analysis models, or simulation of the

alternative being studied. An approach is needed that will selectively lock choices to prevent an explosion of alternatives<sup>3</sup>.

Originally, the expert system organization was predicated on a two-stage process. The first stage included heuristics and straightforward calculations (cost lookup, first-order rules, etc.). The second stage would feature more elaborate simulations, via connections to external tools. The plan was to eventually allow the second stage outcomes to get adopted as first stage heuristics as the tacit knowledge matures.

A detailed treatment of ESKER is provided in Appendix 7.6. ESKER has been implemented in the functions of Design Space Exploration and PCC calculation.

### 7.2.5 Tool Plug-ins

Plug-in architectures are implemented by distinct but cooperating modules. We describe each plug-in separately in the following paragraphs.

#### 7.2.5.1 Magic Draw/SysML

To demonstrate an integrated SysML capability for ARRoW, we developed Magic Draw plug-in extensions via the Java-based plug-in architecture of the Magic Draw tool suite. These tools were then used to support modeling of a ground vehicle reference architecture model. This section describes the various features of the extensions that were made to the Magic Draw tool.

##### 7.2.5.1.1 Magic Draw Plug-In Architecture

Magic Draw is a Unified Modeling Language (UML)/SysML tool written in Java, and provides a plug-in based architecture that allows developers to write Java code to extend or modify its behavior. Magic Draw exposes an API to allow plug-ins access to the user interface and the model database. Refer to the Magic Draw documentation for more information.

##### 7.2.5.1.2 ARRoW Magic Draw Plug-In

We developed a single ARRoW plug-in for Magic Draw, with a variety of capabilities. The ARRoW plug-in provides the following functionality:

- Initialize the plug-in
- Collect requirements in a way that can be integrated into AMIL without language parsing.
- Read and writes from/to AMIL graph database
- Solve parametric equations
- Generate custom reports (e.g., OWL schemas)
- Query, read data, load models, and publish models into the component model library
- Select a refinement of a design archetype
- Provide model element information
- Extend SysML model elements with ARRoW stereotypes

<sup>3</sup> A spreadsheet-based approach, although table-driven, is untenable since it lacks: (1) Large-scale maintainability, with the “if-then” rules particularly difficult to implement and (2) Customizable extensibility to outside tools. The latter strongly suggests that flexible reasoners could play a vital role. Interesting to note that, despite decades of development of decision support systems and methodologies, spreadsheets are still popular as primary tools for decision making.

These capabilities are described further in the plug-in software documentation. All source code is found in Subversion (SVN) under the mdplugin project.

#### 7.2.5.1.2.1 Initialization

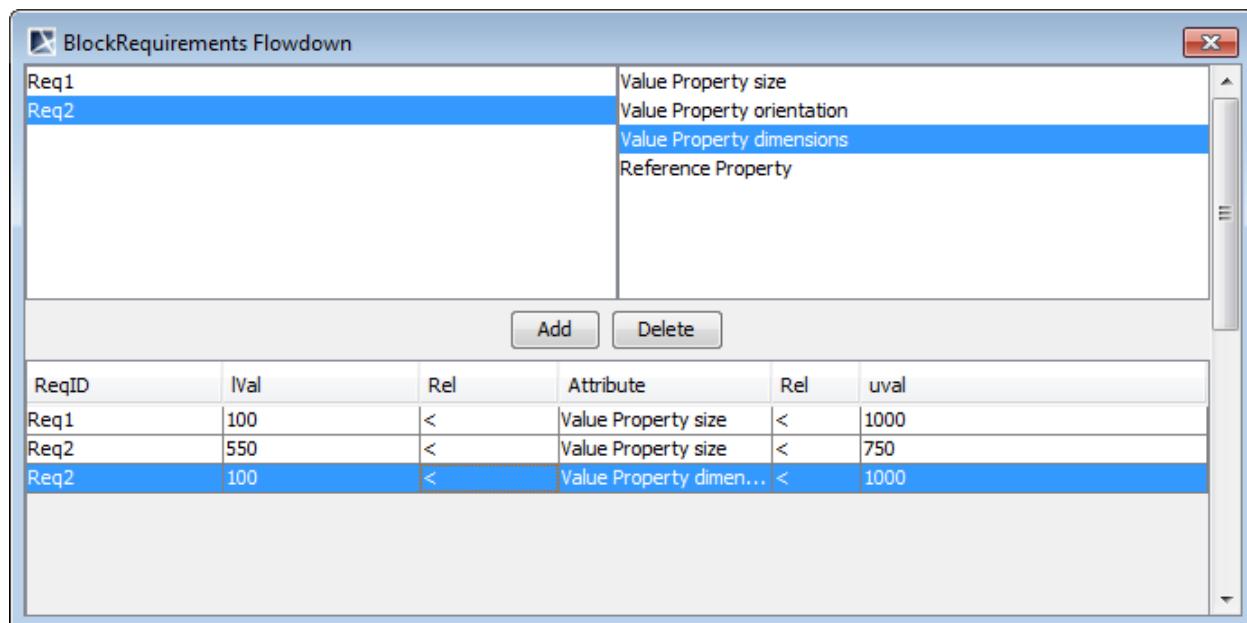
On startup, Magic Draw checks a certain directory for plug-ins. The mdplugin project, when Maven Install is run, build the plug-in and installs it the proper directory under the MDUML\_HOME environmental variable. A file, plugin.xml tells Magic Draw about the plug-in, including the name of the root Java class to initialize it, and JAR dependencies the plug-in has. The ArrowPlugin class is responsible for initialization and installation of all ARRoW capabilities within Magic Draw.

#### 7.2.5.1.2.2 Requirements Capture

Requirements are captured as SysML requirements model elements, which are really just wrappers around the text of the requirement. For ARRoW, we need to get at the meat of the requirement and how it impacts the architectural elements of the system. To achieve this, we created a custom dialog that would capture the relationship between blocks, requirements, and properties, store it within the model, and publish it to AMIL.

A dialog box, implemented in the MD plug-in, is used to manage the requirements mapped onto a block and the properties they affect. The dialog captures the range of values constraint that the requirement places on the block property.

Figure 7.2-12 shows a screen shot of the dialog box for the Opening block, with requirements Req1 and Req2 placing constraints on the size property. The prototype captures ranges of values, in the form lVal < property < uVal. This can change to deal with other syntax, tolerances, or more general expressions as we evolve the solution.



**Figure 7.2-13. Opening Block Dialog Box**

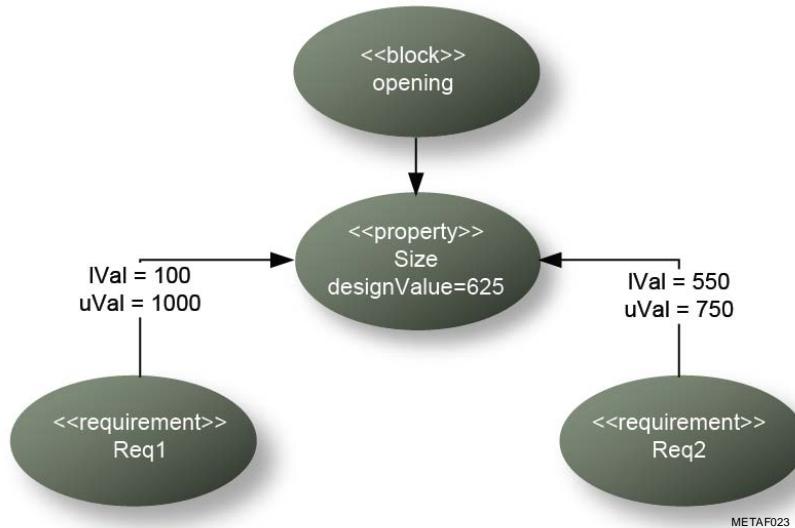
#### 7.2.5.1.2.3 AMIL

## Write

The MD plug-in adds a menu item to the main File menu to Publish to AMIL the model. All requirements and all Blocks with the <<RequirementValues>> stereotype are published to AMIL. The following diagram documents the pattern of SysML elements as they are mapped to AMIL nodes. This happens within the RequirementValues.java class.

Blocks, instances, properties, and requirements are each mapped to their own AMIL node. The arc from block to property represents the ownership of the property by a block. Attributes of the property, such as type, description, etc., are not shown.

The arcs from requirements to property represent the constraints or desired values of the property, and the range expression is mapped to attributes of the arc. As illustrated in Figure 7.2-13, the IVal and uVal ranges are included on the arcs.



**Figure 7.2-14. AMIL Representation of Requirement Ranges of Values**

Design values are assigned to a property and mapped to the AMIL graph using a *designValue* property on the SysML property's AMIL node.

## Read

The ARRoW allows reading back out of the AMIL graph for a single node. The ARRoW SysML tools support reading back *designProperty* values back from the AMIL graph, allowing tools to exchange design information. Since SysML is the authoritative source for requirements in our application of ARRoW, requirement values are not read from AMIL. This feature will also read back properties in ECTo's format of name-value pairs on the AMIL node. Named AMIL properties that do not align with SysML properties are ignored.

This feature is implemented in PopulateFromAMIL.java.

### 7.2.5.1.2.4 Parametric Models

We used parametric diagrams to mathematically relate block properties with one another. While we did not extend Magic Draw to achieve this capability, we do note that other Magic Draw add-ons are required. ParaMagic and either OpenModelica, Modelica, or Mathematica are required for full parametric support.

### **7.2.5.1.2.5 Report Templates**

Magic Draw uses the open source *velocity* tool as the means for generating reports from the models. While not part of the Java plug-in extensions, we have created custom reports for ARRoW. These templates are in SVN in the mdplugin/src/main/resources/templates directory.

The owl.txt report generates an OWL schema from the SysML models.

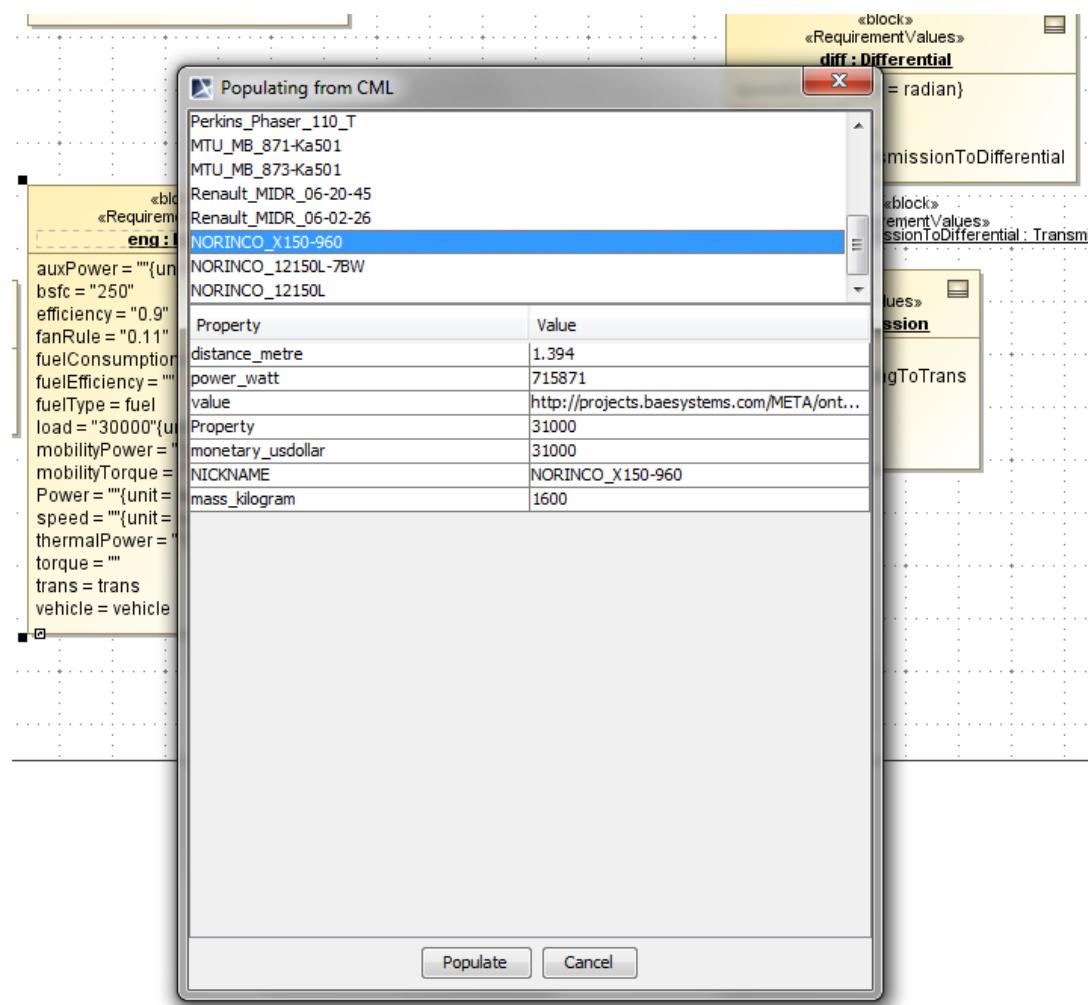
SysMLStructuralReportTemplate.rtf is a fix to Magic Draw’s default report template that includes blocks in the report.

Refer to the Magic Draw documentation for how to use custom reports from the Tools>Report Wizard menu option.

### **7.2.5.1.2.6 CML Query**

The ARRoW can query the CML database for components of a particular type, like engine, and can even build more complex queries to find engines with power in a specific range. As illustrated in Figure 7.2-14, the results are shown in a dialog box, allowing the user to browse the results. A single result can be selected, which loads the CML property values into the model element’s SysML properties.

This occurs within the PopulateFromCMLAction.java and PopulateFromCML.java classes.



**Figure 7.2-15. Browsing Query Results**

#### 7.2.5.1.2.7 CML Publish

A Magic Draw package, containing an archetype, can be published to CML from within Magic Draw. This adds the mdzip file to the CML and Artifactory, which supports the CML.

#### 7.2.5.1.2.8 Archetype Refinement

The reference architecture includes decision points in selecting among design patterns or components within the design. An ARRoW plug-in class, RefineDesignArchetypeAction.java, reads from an AMIL node that triggers a reasoner within the AMIL graph to run, collect design data, rank alternatives, and provide the top ranked option. The selected option is then automatically added to the design.

#### 7.2.5.1.2.9 Node Info

As an exploration and understanding tool, the ARRoW plug-in allows model elements on diagrams or in the containment (tree) view to be queried about their type and other information. This is in the NodeInfo.java class.

### **7.2.5.1.2.10 ARRoW Stereotypes**

The ARRoW defines stereotypes to extend the Block and Requirement SysML model elements. These are described in the Language section of this Final Report.

### **7.2.5.1.3 Tool Considerations**

For this work, we chose to use the Magic Draw UML/SysML tool, version 17.0, service pack 2. There are many other tools that support UML/SysML, including some open source tools. While we made some extensions to the tool, via a plug in architecture that Magic Draw exposes, we kept those extensions to a minimum. Other tools have similar extendibility approaches, so these plug ins can be adapted to other tools.

The profiles, extensions to the SysML language, should be interoperable with other tools, as should the UML and SysML models of the reference architecture.

ParaMagic is a tool from InterCax that extracts SysML parametric models and makes them available to a solver, such as Open Modelica. ParaMagic has no open source equivalent currently. However, it would be possible to write a SysML-tool-independent utility to extract the parametric model to AMIL, then write a solver on top of the AMIL graph.

The following lists the features of a UML/SysML tool that we are using and dependent on within ARRoW.

- Tool extension architecture, like a plug in capability, to add custom menu items, dialog boxes, etc., to the tool.
- Ability to use the extension mechanism to create an interface to AMIL.
- Supports the UML 2.X and SysML 2.X standards
- Supports profiles
- Ability to partition models into separate files, at least package level
- XMI interoperability
- Support for all diagrams and constructs used in the Reference Architecture.

### **7.2.5.1.4 Future Work**

#### **7.2.5.1.4.1 Separating the Archetypes into Separate Files**

Currently, all design archetypes live in one file, under models/SysML/IFV.mdzip in SVN. Each archetype should be broken out into its own package and file for storage in CML.

#### **7.2.5.1.4.2 Auto-Requirement Mapping**

Requirement archetypes are mapped onto the reference architecture components they impact. A corresponding relationship happens at the design level with requirements attached to design elements. This process could be automated, to create and attach requirements from requirement archetypes as design elements are created.

#### **7.2.5.1.4.3 Design Under Test Configuration for Test/Co-Simulation**

Using the internal block diagrams, behavioral models can be composed to support a co-simulation. This SysML diagram can then be leveraged into executing the composition, extracting models from CML and deploying it on the appropriate platforms.

#### **7.2.5.1.4.4 Synchronization with ECTo**

Both ECTo and SysML cover the design space from different viewpoints. We could combine the reference architecture and ECTo to create a Domain Specific Language (DSL), a language

that supports the capabilities and expressiveness of UML/SysML, but with the visual interface of ECTo. Constraints can be enforced by warning the user that the engine usually doesn't go inside the squad compartment. The DSL would also ease the mapping of requirements onto the design elements.

Ideally, ECTo would evolve into a tool that is plug-and-play, meaning that new component types could be added without having to change the ECTo code base itself. Parametric relationships that are hard coded into ECTo now could be authored in SysML and exported into AMIL for use by ECTo. Likewise, design archetypes, components, subcomponents, and peer connection patterns could be packaged for use by ECTo.

Ultimately, one of the desired features of the META project is to support the analysis of parametric models outside the context of MagicDraw. MagicDraw could continue to be used for authoring parametric relationships between components, just as it would be used for developing the ontology interface for CML. The parametric models would also be published into AMIL. However, instead of running the ParaMagic plugin from within MagicDraw, a similar tool living in the cloud could be launched from within AMIL and used to solve the parametric equations. This way, MagicDraw, ECTo, and other tools could benefit from the use of tool independent parametric models, and the mathematical analysis would move from the client to the cloud.

### **7.2.5.2 Pro/Engineer (Creo) Plug-In**

#### **7.2.5.2.1 Introduction**

The Pro/Engineer (Pro/E) was recently renamed Creo, but is referred to in this paper as Pro/E. This plug-in dynamically provides parametric information for generating mass, moments of inertia, model structure, and baselines of Pro/E objects.

The information generated uses JSON (JavaScript Object Notation) for lightweight data-interchange. The JSON provides an easy text format that is language independent but has conventions that are familiar to programmers. In addition, since the AMIL language is a derivative of JSON, it allows for easy input into the META systems.

#### **7.2.5.2.2 Benefits**

As the main worker, the Pro/E Plug-in does the heavy lifting of calculating the parametric information and producing the necessary output. The plug-in consumes information placed on the queue, with the JMS provider ensuring that items in the queue are only processed once.

The architecture design for the Pro/E plug-in is configured to leverage cloud technology. For example, if the demand for processing Pro/E objects increases additional workers can be started. This offers parallel processing for on demand resource utilization while the plug-in dynamically analyzes and calculates mass properties of any Pro/E object.

#### **7.2.5.2.3 Conclusion**

The Pro/E Plug-in provides critical parametric-constraint properties to allow automated design decisions.

### **7.2.5.3 Incorporating Lightweight, Open Source, Freeware Tools into the Tool Chain**

#### **7.2.5.3.1 Engineering Design Tools Investigation**

The BAE Systems META tool chain is founded upon the notion of heterogeneous tool and technology integration and lightweight, unobtrusive data integration mechanisms. This approach attempts to enable the integration of fast and automated abstract design

methodologies, but retains the capability to access high-fidelity domain-specific tools and capabilities where needed and appropriate for the development of a complex, safety-critical, weapons platform like a combat vehicle.

An analysis was completed to evaluate lightweight tool capabilities in the design chain. An evaluation found existing open source tools with similar functionality matched to commercial tools as shown Figure 7.2-15. An initial investigation assessed CAD due to its cost and complexity.

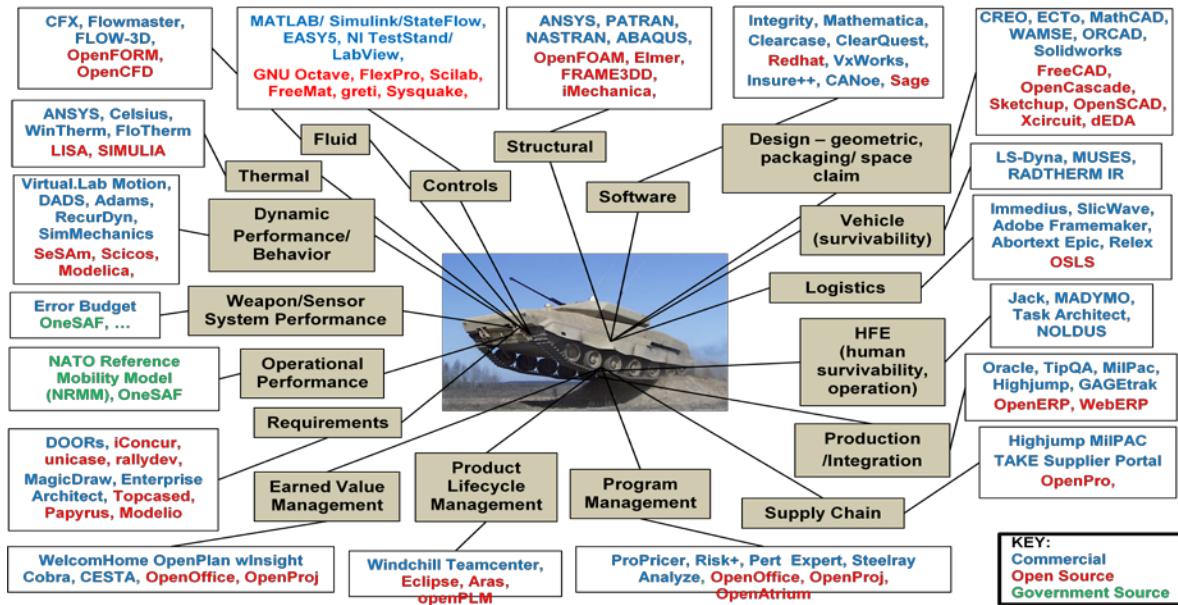


Figure 7.2-16. Relevant Commercial and Open Source Tools

### 7.2.5.3.2 Capability Sampling

A capability assessment was concluded evaluating low-cost or freely-available CAD tools for use in the META program. Some of the things evaluated include ease-of-use, flexibility, ability to create and modify geometry, and any APIs to allow automation through scripting or other software interfaces.

Low-cost or free (commercial or open-source) CAD tools can be an important part of early product development. These CAD tools with an easy but powerful user interface coupled with a richly-populated component model library will allow a wide user community to try out different early concept layouts of a system.

The CAD tool capabilities to focus on at this early stage in conceptual design (of a large system layout) include easy placement and movement of component geometry (singly or in groups), easy geometry creation and modification, and presentation ability. Less important capabilities include top-down design tools, drafting/annotation modules, or other detail-design capabilities. Another important characteristic of a useful CAD tool is the ability to deal with importing/exporting geometry from other CAD systems while still allowing full modeling capabilities of the CAD tool (i.e. no loss of functionality or geometry).

Later, during preliminary design, the CAD tools can shift to more powerful (presumably more expensive) CAD tools to support a more robust development activity.

### 7.2.5.3.3 Conclusion

Generally, there are a lot of unknowns (such as performance with large models) with the CAD software packages evaluated. Additional evaluations are necessary to conclude where and how best to use open source and freeware software.

## 7.2.6 Metrics

Metrics provide the ARRoW designer with valuable calculations for measuring design performance. In addition, metrics support credible decisions within the limited resources of design space exploration. By designing a controlled process for assimilating metrics given design goals, assumptions, and limitations, design space exploration proves more efficient.

In order to facilitate evaluation and exploration of design space in META, the purpose of metrics is threefold:

1. provide the ARRoW designer with innovative metrics used for measuring design performance
2. offer an extensible framework for integrating and developing metrics
3. allow the ARRoW designer to inspect and analyze metric calculations through a comprehensive user interface

### 7.2.6.1 Metrics Framework

The metrics framework provides the mechanism for integrating metrics for design evaluation. The purpose is to facilitate the evaluation of a design by providing quantitative measures, flexible metric selection and grouping. The metrics framework also provides the ability to rapidly prototype and integrate new metrics.

The framework is designed to use metrics consisting of at most two types of sub-elements; evaluators and statistics. In addition, statistics incorporate a third type of element: measures. Each component is defined as follows:

#### *Measures*

Measures are quantities that can be directly observed from the design under development. For example, the weight of a vehicle's hull is a quantity that is directly measured from the design, without requiring further reasoning or analysis. Measures, however, often times are coupled with running a simulation in order to measure quantities calculated after the simulation.

#### *Evaluators*

Evaluators orchestrate the execution of measures used to compute a metric. For example, the metric 'Total Weight of the Vehicle' requires the use of an evaluator to execute many measures responsible for measuring the weight of individual vehicle components.

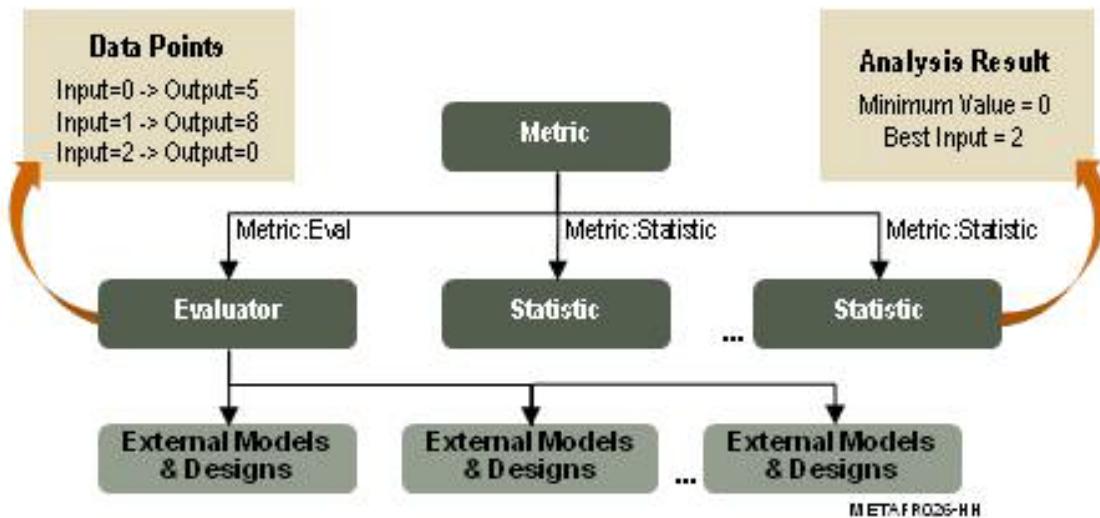
#### *Statistics*

Statistics perform a summarization of data. A statistic may be used to compute an average or maximum calculation. A statistic may, however, return the entire set or a subset of values collected by an evaluator as a form of summarization.

Metrics, evaluators and statistics are represented by AMIL nodes and are associated through AMIL links. Modeling metric elements in AMIL facilitates analysis and reasoning over the

design characteristics represented in AMIL (additional discussion on the use of AMIL for metrics is given in Section 7.2.1.3).

Figure 7.2-17 illustrates the structure of a metric. Metrics directly reference an evaluator and at least one statistic element. The metrics framework evaluates a metric by first evaluating the metric's evaluator. By evaluating the evaluator, data is collected from the various referenced design and model properties. Subsequently the statistics nodes are used to process all of the collected data for statistical analysis.



**Figure 7.2-17. A Generic Metric Composition**

The metrics framework also maintains some key features listed below:

- Metrics used within the framework are consistent and repeatable. Consecutive executions of a metric maintain the same analysis when no changes are made to simulation or model data.
- Provides a convenient abstraction for interfacing metrics with design data in AMIL
- Manages node versions over time
- Capable of easily integrating executable models with limited concern about ownership or proprietary data

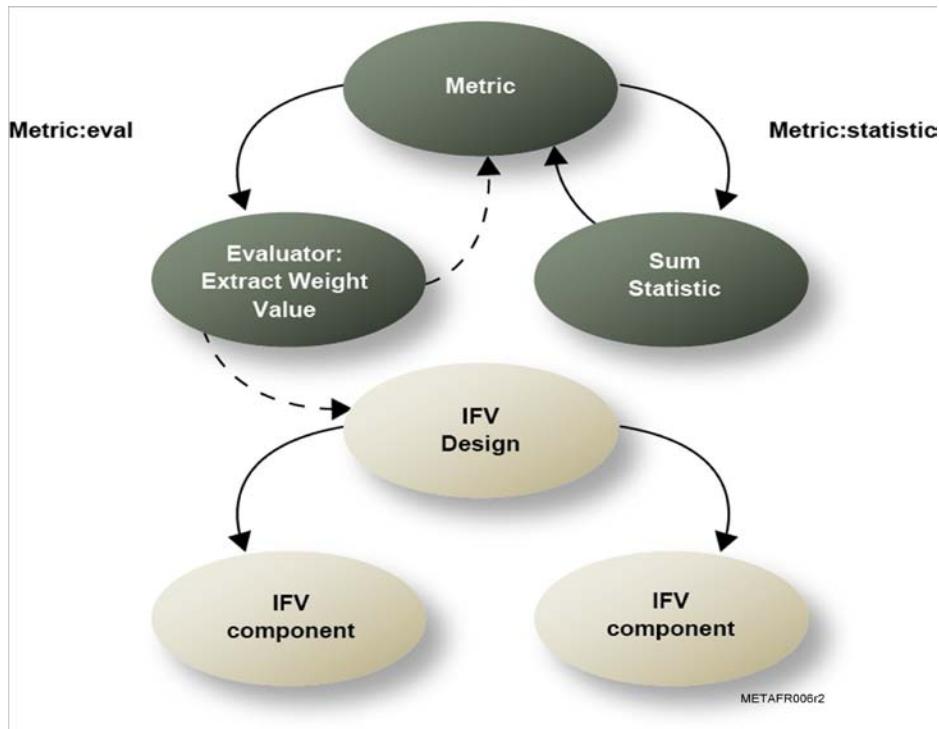
### 7.2.6.2 Specific Metrics

Using the structure of the generic metric model designed within the framework, a number of metrics were built and integrated for this effort. Two of these metrics were developed external to BAE, and were integrated within days. For a more mature system, we believe this time would be just hours.

#### 7.2.6.2.1 Weight

As shown in Figure 7.2-17, system designers can calculate the total weight of the proposed design by accessing weight information provided to by ECTo. The metric evaluator walks the AMIL graph for the vehicle design, collecting weight values in support of the metric computation. The metric statistic contains the information to sum these values. The value for

the weight metric can be used as input to Matlab Simulink mobility simulations in support of computing other metrics such as top speed, fuel efficiency and time to accelerate to 20 MPH.



**Figure 7.2-18. Total Weight Head Node Connectivity Computation**

#### 7.2.6.2.2 Graph Complexity

Complex designs are known to be, on average, more fragile than simple ones, and feature more dependencies; longer integration, testing, and maintenance times; and greater susceptibility to abstractions leakage “at the seams.” Because the AMIL graph provides the “glue” for the design under consideration, we are able to generate simple graphical measures to understand how the complexity of one design compares to another. We have implemented the following metrics: total node count, count of incoming and outgoing links to each node, and maximum link counts to/from a node. These metrics are simple enough that they also support testing for new ARRoW users to verify operational server and database, and successful loading of metric data.

#### 7.2.6.2.3 Signal Complexity

This metric calculates the signal entropy of a time series to indicate complexity. This metric was developed by team member BBN and is documented in Section 7.7.

Because the metrics framework is an extension of AMIL, the same mechanism for integrating 3rd party tools into AMIL is applied to metrics. A standard evaluator node is provided by AMIL to call an externally authored AMIL node with "overridden" parameters passed in and extract "user configured" output values associated with those input parameters. For the statistic node, 3<sup>rd</sup> party tools or metric authors can use a pre-defined statistic node (such as max, min, sum, count, etc), build a new statistic node, or omit one altogether. The integration is completed by pointing the metric node to the defined evaluator and statistic nodes.

#### **7.2.6.2.4 PCC Calculation**

This metric utilizes the PARC Envisioner to generate a PCC comparison across two designs for a ramp use case. This metric was integrated similar to the BBN metric integration above.

#### **7.2.6.2.5 Design Decision Support**

The MagicDraw Plug-in (discussed in Section 7.2.4) employs the metrics framework to reason a best platform type based on comparisons between constraints of the design and constraints of the candidate platform types. In this case, the evaluator collects the requirements stored as AMIL nodes, evaluates all candidate design measurements against all criteria and returns the minimum distance result. The minimum statistic used here passes back both the numeric result along with information about the candidate design for which that result was computed.

#### **7.2.6.2.6 Max Torque Variation**

This metric measures the sensitivity of a model to variations in its inputs. The amount that each of the simulator input variables is to be varied is defined in the evaluator. A mesh grid of multivariate inputs is created by the evaluator and assigned to the measurement node, to result in a sensitivity analysis to the amount of change in the result of Simulation 3’s output for a determined amount of change of input.

#### **7.2.6.2.7 Procurement Cost**

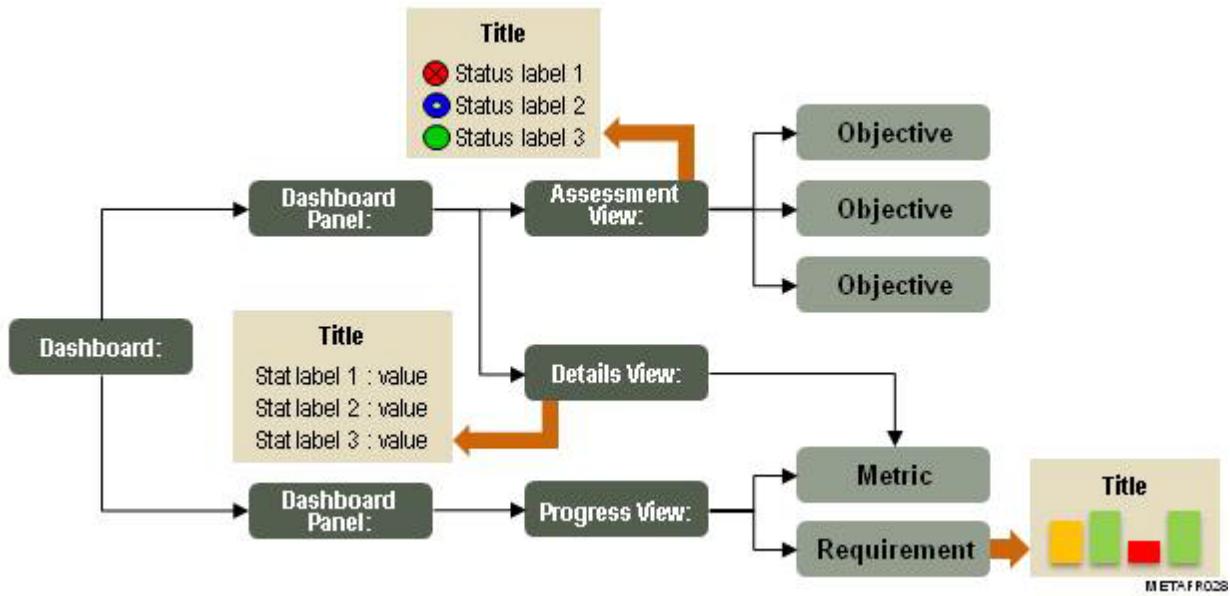
In order to support a procurement cost metric, we used an accepted model that mapped weight to procurement cost and used rough cost per pound data, in 2011 dollars, for several classification types. These costs were based on the average unit production of 5000 units. The 17 classifications in the model are Automatic Fire Emergency System (AFES), Armor, Chassis Structure, Crew Station, Defensive Armament, Dismountable, Electronic Control System (ECS), External Lighting, Fuel, Hit Avoidance, Hydraulics, NBC, PDM, Platform Electronics, Propulsion, Signature Management, Suspension.

#### **7.2.6.3 Metrics Dashboard**

Feedback to users on design performance is provided by the metrics dashboard. Metrics themselves have many consumers, so the dashboard is easily configurable to support those consumers. Top sponsor leadership and company management may be interested in cost and system effectiveness, and capability and gap analysis, whereas the upper project management are more interested in the health of the design and risk mitigation metrics. Direct line management will be interested in reviewing tracking book metrics.

To support the varying needs of the different metrics consumers, there are three demonstration dashboards implemented: Demo, Design, and Requirements. The Demo dashboard is used to demonstrate a sample of a top level cost analysis and capability assessment. The Design dashboard relies on the outputs of an ECTo design exported to AMIL and presents the results of a mobility model and other design metrics such as calculated total cost and total weight. The Requirements dashboard provides a table of the integrated PARC and BBN Signal complexity results.

Each dashboard is composed of a combination of views. As shown in Figure 7.2-18, these views pull data from AMIL and use HTML, JSON, and JavaScript. Each view specifies one or more AMIL nodes from which to retrieve data. The views then request the AMIL node results, glean the desired properties from them and display them.



**Figure 7.2-19. Dashboard Structure in AMIL**

The dashboard views developed to date include:

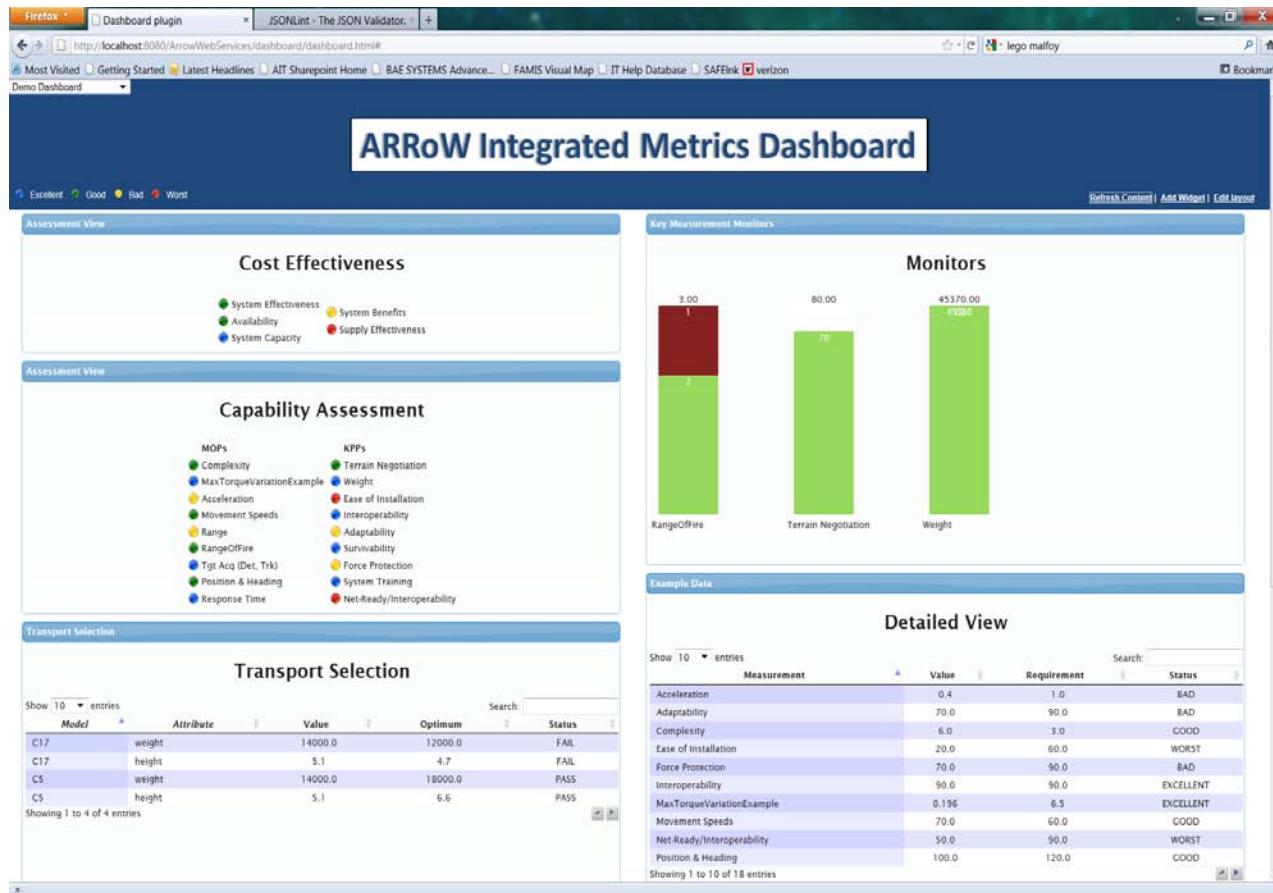
1. Assessment view: dynamically constructed “gumball” rollups of data received via AMIL. An assessor compares calculated values to minimum and required values and determines, according to a rule, which region the value falls into. An example of the quadratic and linear assessors of the value of the System Weight metric is shown below. The quadratic specifies a characteristic length from the requirement to where the assessment is no longer excellent while the linear specifies a percentage of minimum below which the assessment is deemed as worst. Using an assessor determines the color displayed for a metric in the assessment panel.
2. Progress Bars/Bar charts: dynamically evaluates a node value against an objective measure (which is likely referencing a requirement)
3. Details View: Dynamically construct a table of data values received via AMIL
4. Metric Details View: Dynamically construct a table of evaluator data values (inputs and outputs for each data point)
5. Summary View: Dynamically construct an HTML summary of the statistic results contained within a metric

The dashboard is configurable both in terms of data displayed and widget placement on the screen.

Additional data nodes represent content for the dashboard panels. For example, a necessary metric for a quality IFV design may be that the tank has wheels or tracks and a turret. A sufficient condition for a good design may be that the cost and weight are within desired ranges. To support display of such information, the content nodes can reference other AMIL nodes (indicator for having a turret, or ranges for cost) for which we can extract a value to analyze and present.

An example of the Demo dashboard is shown in Figure 7.2-19. The various panels within the dashboard are provided by the different views that have been developed, and include: metrics

values, assessment of metrics values (excellent, bad, worst), comparison of design alternatives against requirements, and monitoring graphics.



**Figure 7.2-20. Example Dashboard Configuration**

The work of Phase 1B has extended a vehicle designer's ability to make decisions based on objective feedback on the design. The ARRoW metrics framework seamlessly integrates metrics with simulation and design tools, hides the back-end AMIL details, is easily re-configurable and supports straight-forward metric creation.

### 7.2.7 Cloud Deployment

Use of the Amazon Web Service (AWS) cloud provided a simple mechanism for deploying the ARRoW Web service for external use. A cloud computer instance is similar to any other computer that can be bought off the shelf. AWS provides a barebones Windows or Linux system and the user installs software required to run the application. The strengths of cloud computer instances are their ability to be readily available on the Web, their ability to scale in storage and computational power as necessary, and its ability to store and reuse instance images.

AWS incorporates a Security Group capability to control access to the cloud instance. The Security Group is basically a list of IP addresses and ports through which access is established. Through this list, Administrators of the cloud instances establish rules for governing which external IP addresses have access to the cloud instance, and which types of connections are allowed (e.g., SSH, HTTP, VNC, ...). Each cloud instance is externally identified through an

IP address. When an instance is started or rebooted, it is assigned a new IP address. AWS incorporates an Elastic IP capability that provides a static IP address for external access so that users only have to remember the one address.

Cloud instances are scalable through computational power, RAM amount, and disk storage size. When an instance is launched, AWS provides an option for selecting the size of the computer in terms of number of processors and size of RAM. After the instance is launched, any amount of disk storage can be added.

AWS provides a capability for creating and storing images of cloud instances. This provides a convenient configuration management mechanism for maintaining baseline versions of the application and testing enhancements to that application. An image is a snapshot of the computers software configuration. This includes the operating system, application software, user accounts, environmental variables, data files, and anything else required to reestablish the running instance. Once the application is installed and successfully verified in a running instance, an image of the instance provides a means for getting back to this exact running state at any time. This then provides the freedom to change the instance to tryout potential enhancements without worry of losing the known good state.

During Phase 1B of the contract, two cloud instances were successfully configured to run the ARRoW Web service. One instance utilized a Red Hat Enterprise Linux 64 bit operating system and the other a Microsoft Windows 2008 R2 SP1 64-bit architecture operating system. The procedures for establishing these two instances are included in the User’s Manual that accompanies this report.

### 7.2.8 Bibliography

- [ARC11] Archiva documentation (2011), available at <http://archiva.apache.org/>
- [ART11] Artifactory documentation (2011), available at <http://www.jfrog.com/>
- [BPT11] Blueprints documentation (2011), available at  
<https://github.com/tinkerpop/blueprints/wiki>
- [HY10] Hong Yang, R. C.-q. (October 2010). “A Metrics Method for Software Architecture” 5 (JOURNAL OF SOFTWARE, Issue 10).
- [INC95] *Metrics Guidebook for Integrated Systems and Product Development*, INCOSE-TP-1995-002-01 (originally INCOSE TM-01-001).
- [JSN11] *Introducing JSON*. (2011). Retrieved 9 26, 2011, from <http://json.org>
- [META11a] META Adaptive, Reflective, Robust Workflow (ARRoW) Phase 1A Final Report, TR-2683, 13 April 2011.
- [MVN11] Maven documentation, (2011), available at <http://maven.apache.org/>
- [NEO11] neo4j: The Open Source, NoSQL Graph Database. Retrieved 9 26, 2011, from neo4j.org: <http://neo4j.org>
- [NXS11] Nexus documentation (2011), available at <http://nexus.sonatype.org/>
- [OWL11] *OWL Web Ontology Language Reference* (2011), available at <http://www.w3.org/TR/owl-guide/>
- [PRT11] Protege documentation (2011), available at <http://protege.stanford.edu/>

[RDF11] World Wide Web Consortium, OpenRDF documentation (2011), available at  
<http://www.openrdf.org/>

[RST11] REST documentation (2011), available at  
<https://www.ibm.com/developerworks/webservices/library/ws-restful/>

[SQPR11] World Wide Web Consortium, *SPARQL Query Language for RDF* (2011), available at <http://www.w3.org/TR/rdf-sparql-query/>

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.3 – Modeling Language**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.3 Modeling Languages in ARRoW .....</b>	1
7.3.1 Languages and Their Purpose .....	1
7.3.1.1 Roles of Languages in Designing Complex Systems .....	1
7.3.2 Reference Architecture.....	4
7.3.2.1 SysML as the Umbrella for Multi-viewpoint Systems Integration.....	4
7.3.2.2 Engineering Activities.....	6
7.3.2.3 Archetypes.....	6
7.3.2.4 Archetype Development.....	10
7.3.2.5 Abstract Components .....	14
7.3.2.6 Design Refinement .....	15
7.3.2.7 Context Components .....	17
7.3.2.8 Viewpoints .....	17
7.3.2.9 Parametrics.....	18
7.3.2.10 Relationship with CML.....	18
7.3.2.11 Behavioral Models .....	19
7.3.2.12 SysML Usage.....	19
7.3.3 AMIL.....	29
7.3.3.1 Uses of AMIL Analysis and Verification.....	29
7.3.3.2 Design Support.....	29
7.3.3.3 Chaining of Tools .....	29
7.3.3.4 Analysis Reuse.....	30
7.3.3.5 Semantics of AMIL.....	31
7.3.3.6 AMIL Syntax .....	32
7.3.4 Qualitative Modeling Language .....	33
7.3.4.1 Qualitative Simulation in Early Design.....	34
7.3.4.2 Qualitative Representations .....	34
7.3.4.3 Specifying Qualitative Models .....	34
7.3.4.4 Example: Analysis of IFV Door Opening and Closing .....	37
7.3.4.5 Envisionment and PCC Computation .....	39
7.3.4.6 Modelica to QML Translation .....	40
7.3.4.7 Qualitative Simulation Semantics .....	43
7.3.5 Bibliography .....	43

## List of Figures

Figure 7.3-1. Three Worlds of Engineering.....	2
Figure 7.3-2. Categorization of Languages .....	3
Figure 7.3-3. Roles of Languages in Design.....	3
Figure 7.3-4. Reference Architecture .....	4
Figure 7.3-5. SysML Integration with Domain Specific Engineering Areas .....	5
Figure 7.3-6. Requirement Archetypes and Design Archetypes .....	8
Figure 7.3-7. Acceleration Requirement Archetype .....	9
Figure 7.3-8. Acceleration Test Archetype.....	9
Figure 7.3-9. Acceleration Test Sequence Diagram .....	10
Figure 7.3-10. Design Archetype Applicability Ranges .....	12
Figure 7.3-11. Ground Vehicle Design Instances .....	13
Figure 7.3-12. SolidComponent Block and Core Relationships .....	14
Figure 7.3-13. Power Train Design Instances.....	15
Figure 7.3-14. Behavioral Models.....	16
Figure 7.3-15. Example Engine Power and Torque Curves .....	16
Figure 7.3-16. Ground Vehicle Viewpoints .....	18
Figure 7.3-17. Combining Behavioral Models into a Co-Simulation.....	19
Figure 7.3-18. Block Diagram Integrating Multiple Viewpoints.....	21
Figure 7.3-19. Physical Interfaces of an Engine.....	22
Figure 7.3-20. Engine Torque Constraint Equation .....	23
Figure 7.3-21. Usage of Engine Torque Constraint Equation.....	23
Figure 7.3-22. Power Train Design Instances.....	25
Figure 7.3-23. Engine Torque Parametric Diagram .....	26
Figure 7.3-24. Requirement Values Stereotype .....	27
Figure 7.3-25. ARRoWRequirement Stereotype .....	27
Figure 7.3-26. AMIL Representation of Requirement Ranges of Values .....	28
Figure 7.3-27. Relationships Among Models and Design Elements.....	30
Figure 7.3-28. Example of Inferring a Link.....	32
Figure 7.3-29. Definition of an Ideal Resistor including Both Qualitative and Quantitative Equations .....	35
Figure 7.3-30. Defining an ideal diode requires landmarks to divide the quantity space into additional intervals, and modes to model discrete transitions.....	36
Figure 7.3-31. Composition of components in QML defining a system involving a diode (shown graphically on the right). .....	37
Figure 7.3-32. Composition of components defining a system in which a door is given a command to open and close. ....	38
Figure 7.3-33. (left) Directed graph describing the envisionment of the door system included for concreteness. (right) A simplified version of the envisionment graph to describe the example.....	39
Figure 7.3-34. Modelica and QML Models of a Capacitor.....	41
Figure 7.3-35.Translating RLC Model in Modelica To QML.....	42

## List of Symbols, Abbreviations, and Acronyms for Appendix

Symbol, Abbreviation, Acronym	Definition
AMIL	ARRoW Model Interconnection Language
ARRoW	Adaptive, Reflective, Robust Workflow
ASIC	Application-specific integrated circuit
BDD	Block Definition Diagram
CAD	Computer Aided Design
CBD	Contract Based Design
CDRL	Contract Data Requirements List
CML	Component Model Library
DARPA	Defense Advanced Research Projects Agency
DUT	Design Under Test
ECTo	Early Concepting Tool
FEA	Finite Element Analysis
IBD	Internal Block Diagrams
IFV	Infantry Fighting Vehicle
JSON	JavaScript Object Notation
OWL	Web Ontology Language
PCC	Probabilistic Certificate of Correctness
PID	Proportional Integral Derivative
QML	Qualitative Modeling Language

Symbol, Abbreviation, Acronym	Definition
SWAP	Size, Weight, and Power
UML	Unified Modeling Language

## 7.3 Modeling Languages in ARRoW

### 7.3.1 Languages and Their Purpose

This appendix surveys the requirements for languages that have emerged from developing ARRoW in phase 1B. This includes describing the properties of and uses for two languages developed as part of this effort: AMIL (ARRoW Model Interconnection Language) and QML (Qualitative Modeling Language). Additionally, we describe the use of SysML (an existing language) in ARRoW. An unlimited number of additional languages potentially have a role in ARRoW as means to express models. During Phase 1B, Modelica, MatLab/Simulink, C++, and Java models were used, as well as the CAD modeling system Pro/Engineer. These languages are not discussed here, as their role in modeling components and systems is not unique to ARRoW.

SysML is used in ARRoW to represent system requirements, to support aspects of requirements analysis as part of the conceptual design process, and to provide one source of archetypes to support the design process. Its use as a repository is the primary focus of 7.3.2 which describes its use in expressing a “reference architecture” for Infantry Fighting Vehicle (IFV) design.

Section 7.3.3 describes the role of AMIL in capturing the relationships among heterogeneous elements in ARRoW. In focusing on language requirements, this document describes AMIL’s role in providing this foundation for ARRoW. Implementation details for AMIL can be found in the Tool\_Design appendix. And the user’s manual also provides descriptions of how AMIL tools can be used to support design.

Finally, section 7.3.4 describes QML, a language for describing the behavior of components in terms of transitions among qualitatively similar states. Models in QML are necessary in order to perform analysis based on qualitative simulation, a technology investigated in phase 1B of this project.

The remainder of section 7.3.1 presents thoughts about the various roles languages play in the design of complex systems, the need for multiple languages to fill these roles. And it attempts to orient the various languages presented against a framework of language role.

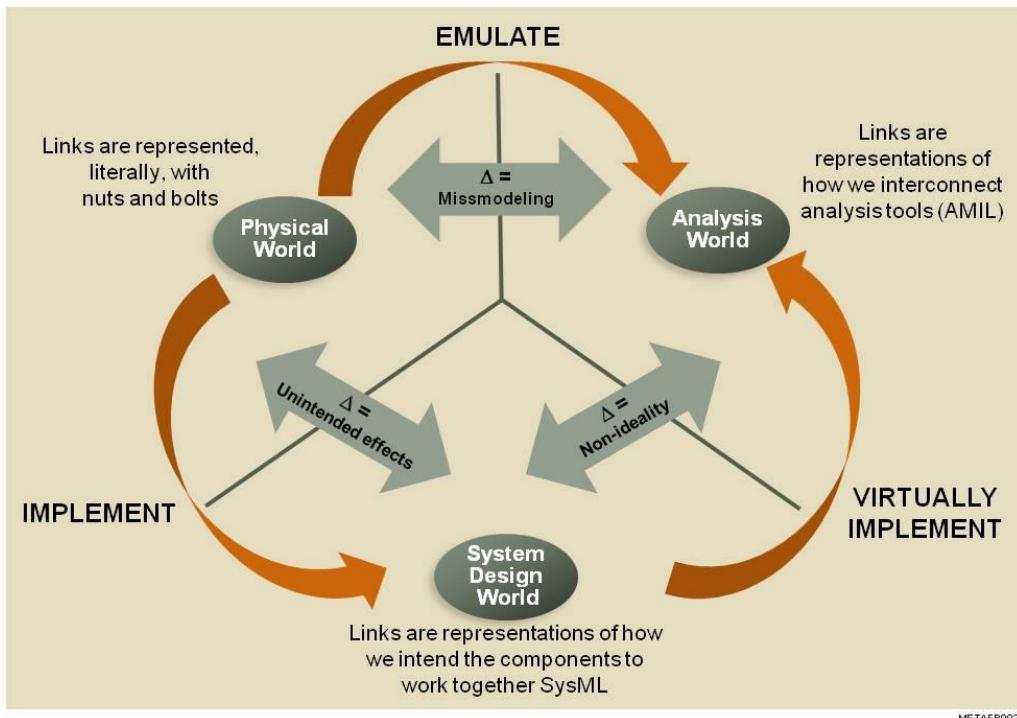
#### 7.3.1.1 Roles of Languages in Designing Complex Systems

Models are used throughout the process of designing, analyzing, verifying, diagnosing, and manufacturing a ground combat vehicle. Some models serve a specific role within the process. For example, a block diagram in SysML may be used to design the vehicle from the perspective of the functional decomposition of the system. On the other hand, you may have a CAD model of a component that is used to design the shape and structure of it, but this model also may be used as the basis for structural analyses using a Finite Element Analysis (FEA) tool.

The diversity of both tasks and information to be represented implies that designers will utilize a diversity of languages. Further, no model is a perfect and accurate representation of the final product or the natural phenomena. Figure 7.3-1 shows some of the differences between the domains of representation engineers must utilize.

While different in topic there are parallels between the domains of design and analysis. Design effort is focused on the decomposition of the problem and defining the function and bounds of a system or a component. Analysis is more focused on the composition of models resulting from

that decomposition to verify that system behavior or performance is as designed or planned in relation to a representation of its environment. Design is not ignorant of these relations and analysis also must deal with decomposition, but that their primary focus does differ.



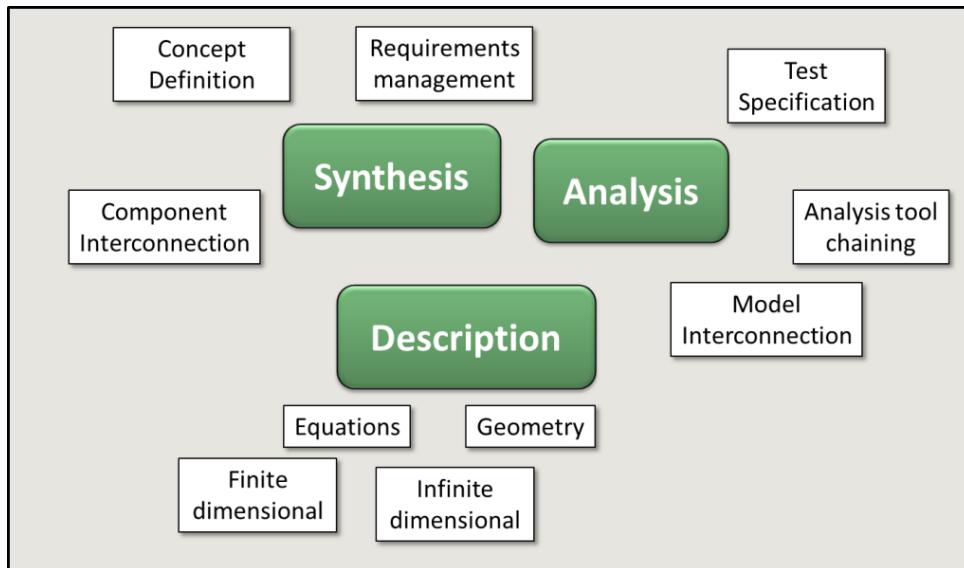
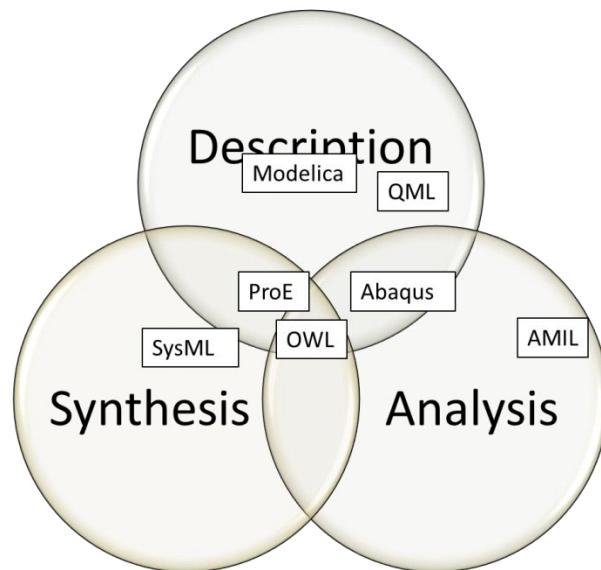
**Figure 7.3-1. Three Worlds of Engineering**

Throughout the history of engineering, design and analysis has been to larger or lesser degrees put on paper. Different languages, informal at first, then standardized, and to some extent formalized have been created to record these aspects of engineering. These languages for analysis and design of cyber-mechanical systems can be catalogued and organized in multiple ways, e.g., by physical domain or engineering discipline they cover, by the type of computational model they embody, or by the stage of the design process they are usually invoked in.

For purposes of describing the roles languages play, we identify three broad phases of the design process:

- **Description** languages characterize the behavior of a component, or a collection of components, by representing, for example, the equations that govern it or its geometry.
- **Analysis** languages describe the processes and protocols used to test and verify whether the system behavior is as desired.
- **Synthesis** languages define how components are interconnected and for what purpose.

Figure 7.3-2 roughly lays out finer grained tasks in relation to these broad phases. In Figure 7.3-3, we list the languages created as part of ARRoW and place them in a Venn diagram according to their most dominant aspects. For context and comparison, we also add other engineering design languages.

**Figure 7.3-2. Categorization of Languages****Figure 7.3-3. Roles of Languages in Design**

The rest of this appendix describes all of the language advances we have made during phase 1B. The flow of this appendix tries to follow the typical design process as it flows from requirements to design and analysis.

### 7.3.2 Reference Architecture

Development of new vehicles, planes, and other complex systems engineering products has languished by comparison to electronic design. ASIC capacity has increased according to Moore's Law, doubling every 18 months. ASIC design has learned to make complex designs out of composable components, but those operate with interfaces of ones and zeros, with some well known physics in support. Their tools have formalized the low level knowledge of transistor layout and allow ASIC designers to operate at higher levels of abstraction. This has led to the exponential increase in productivity.

While the design space of systems engineering is orders of magnitude more complex, the ability to capture the rules, patterns, limitations, interfaces, and components of systems in all dimensions is key to raising the level of abstraction for systems engineers and capturing the benefits that have been realized for other design domains.

One of the key obstacles in this effort is the integration of domain-specific engineering efforts at the system level. Systems engineers are constantly balancing trade-offs in weight, power, range, cost, schedule, capability, etc., across domains. Multiple engineering domains relevant to IFV designs have evolved separately, and have their own vocabulary, notation, and tools that continue to evolve. This makes it difficult for systems engineers to understand all the domain specific models and make design trade-offs.

This section describes our use of SysML to create a reference architecture that brings together product knowledge with engineering domain viewpoints to create an engineering knowledge base. This captures product architecture, design patterns, archetypes, and relationships in a way that can be shared among all stakeholders. And by this means, the experience of systems engineers is captured in machine readable form.

Figure 7.3-4 shows a top level view of the reference architecture for a ground vehicle along with the context with which the vehicle interacts. Domain viewpoints are integrated into the architecture, and trade-off calculations are formalized using sets of parametric equations built from a library of physics and mathematical relationships.

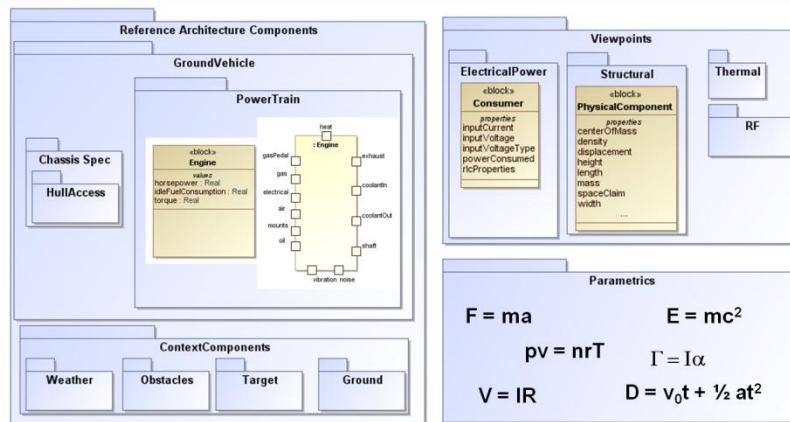
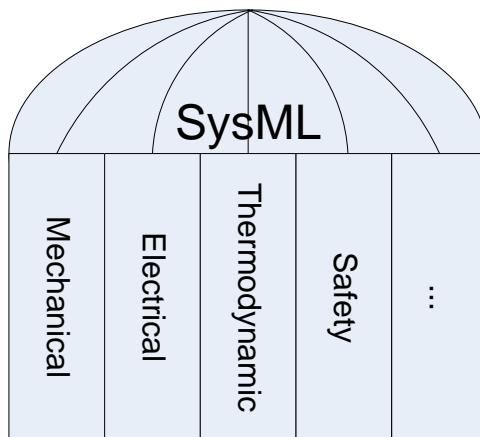


Figure 7.3-4. Reference Architecture

#### 7.3.2.1 SysML as the Umbrella for Multi-viewpoint Systems Integration

ARRoW uses SysML to describe the reference architecture of a military ground vehicle. SysML is a general purpose language for describing systems architecture and design. However,

it does not extend well into domain specific engineering areas. You cannot make a CAD drawing with SysML, nor a circuit diagram. So, SysML based systems engineering models must sit above the domain specific efforts, to link them together. The systems model identifies the key structures, properties, interfaces, and behaviors between the components and between the domain engineering efforts. Figure 7.3-5 shows this relationship.



**Figure 7.3-5. SysML Integration with Domain Specific Engineering Areas**

Designers often rely on their skill and experience to recognize and keep track of the interactions among the domains. This does not scale well to increasingly complex systems, nor can it support domain experts that need to understand how and why the decisions they make affect other areas. SysML provides the infrastructure to relate the multiple aspects of system development and prevent the design leakage that occurs at the seams between components and between domain engineering groups.

The SysML models of the reference architecture tie into the SysML models of the domain viewpoints. These domain viewpoints contain the high level properties that are typically part of the trade-off analysis for systems engineers. From the viewpoint models, system architecture decisions flow down to the domain engineers and tools, and results flow back up to the system architecture for impact analysis and exploration of side effects. The SysML models bridge the gaps between the engineering domains.

One could argue that SysML could be the “one language” to express all aspects of vehicle development. All engineering tools use some kind of data format which can be expressed in UML/SysML. Behaviors that allow the tools to execute mathematical computations and graphical visualizations can be expressed, completely and at various levels, in UML/SysML. (UML/SysML includes a model-level programming language called (UML). However, creating a universal language for all engineering domains would wipe out years of experience developing those domain specific languages. Instead, in the ARRoW language toolset, UML/SysML is used as a glue language to integrate the domain specific languages together.

Each of the domain specific engineering areas is considered a *viewpoint* in the reference architecture. In modeling terms, a viewpoint addresses the questions and concerns of a set of *stakeholders* that have an interest in the system. Each viewpoint is isolated from its counterparts. While the electrical viewpoint is integrated with other components and subsystems, such as communications, computing, etc., the viewpoint allows the entire vehicle electrical system to be

taken as a whole. Domain specific tools and languages can then be used from here to complete the analysis of power consumption, generation, transient effects like power surges, and assessments of cross-coupling interference.

The SysML model of the reference architecture only needs to capture the architecture of the vehicle, including property values, interconnections, and behavioral patterns to pass this information down to domain specific (viewpoint) models and engineering tools for analysis. This interface to the other tools is facilitated by AMIL.

### **7.3.2.2 Engineering Activities**

Reference architecture supports the engineering activities of synthesis, analysis, and description.

#### **7.3.2.2.1 Synthesis**

The reference architecture describes how components can be connected by capturing the idea of connection patterns. For example, moving mechanical energy from one place to another, on a motorcycle from the engine to the wheel, can be done with a driveshaft, chain, or belt. There are different tradeoffs involved – a driveshaft is less efficient, but requires little maintenance. These tradeoffs are also captured in the reference architecture.

Reasoners are created that operate from the reference architecture. The reasoners rank design alternatives by comparing the component context in the design with applicability data associated with each option. For example, a reasoner to choose between a tracked and wheeled vehicle evaluates the design’s mass, desired speed, acceleration, and maneuverability.

#### **7.3.2.2.2 Analysis**

The reference architecture captures test archetypes, which are templates for testing certain types of requirements. For example, a test archetype for acceleration defines interfaces to the design under test—a throttle control and a speedometer readout—and the test driver behavior—floor it until the desired speed is reached. Then a verifier compares the how long it took to reach that speed with the required acceleration.

#### **7.3.2.2.3 Description**

The reference architecture can use any of the behavioral SysML models to describe behavior. State machines describe the behavior of a component. Sequence and activity diagrams describe component interactions as well. Internal block diagrams describe the interfaces of other non-SysML models, such as Simulink models. These block diagrams can then be composed using ports and connectors to build the behavioral model of the entire system.

#### **7.3.2.3 Archetypes**

Archetypes are patterns or templates that can be used as a starting point for developing a product. The reference architecture includes a collection of archetypes that capture the design patterns, constraints, and possible combinations of components to build a system. Any phase of engineering, from requirements through test, can make use of archetypes.

### 7.3.2.3.1 Types of Archetypes

The following types of archetypes are supported in the reference architecture.

#### 7.3.2.3.1.1 Requirements Archetypes

A *requirement archetype* defines a pattern for writing a requirement. The archetype includes systems engineering best practices for phrasing the requirement to make it clear, complete, precise, and testable. The requirement archetype is written in such a way that the requirements archetype just needs to fill in the blanks. The example below shows a requirement archetype for vehicle acceleration.

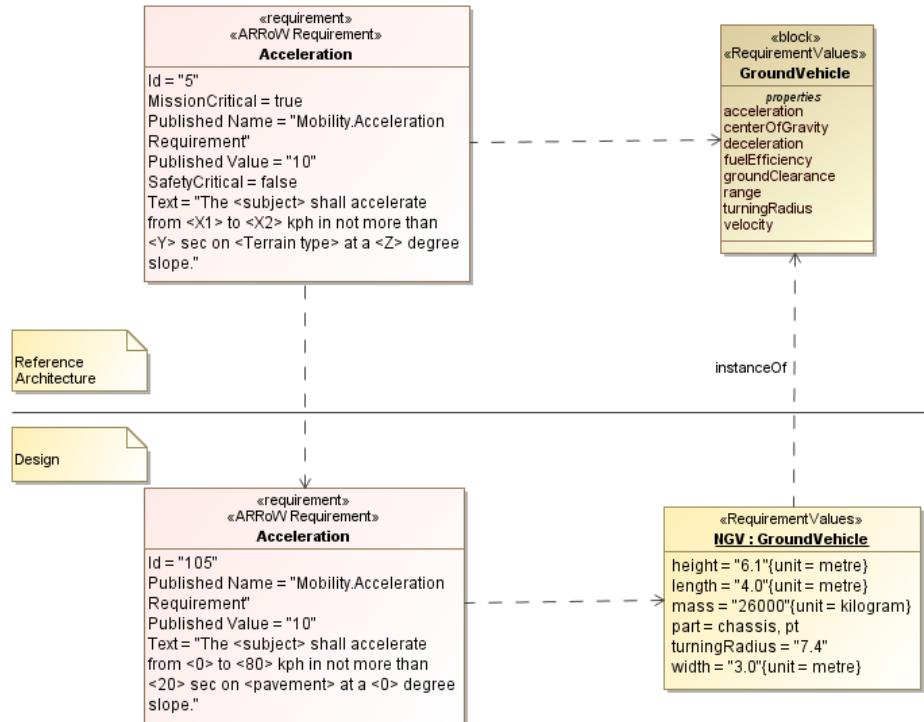
The <subject> shall accelerate from <X1> to <X2> kph in not more than <Y> sec on <Terrain type> at a <Z> degree slope.

A *requirement archetype set* packages related requirement archetypes together. For example, mobility related requirement archetypes, such as acceleration, maximum velocity, turning radius, and obstacle negotiation are packaged together. These archetypes form the core set of requirements that must be considered when creating a requirements document.

Current military vehicle procurement is heavily focused on the requirements documents to define the contract as to what will be delivered. Requirement archetypes provide a head start in developing these requirements with a template of all the aspects to be considered, providing guidance of issues to be considered, based on systems engineering experience, for use by the crowd.

The future of military vehicle development using the META toolset may be less focused on requirements and more on use cases and executable test cases. However, requirement archetypes provide a framework for defining non-functional requirements, such as safety and reliability.

Figure 7.3-6 shows a requirement archetype allocated to a component in the reference architecture. Here, acceleration is a typical requirement of a ground vehicle. The ground vehicle, NGV, is an instance of the GroundVehicle in the system design. The Acceleration requirement is an instance of the Acceleration requirement archetype, with the data and context filled in.



**Figure 7.3-6. Requirement Archetypes and Design Archetypes**

Requirements can automatically be allocated to design elements based on the relationship between requirements archetypes and components in the reference architecture.

### 7.3.2.3.1.2 Design Archetypes

Design archetypes capture the design patterns involved in a typical system. The design archetype describes the subcomponents, relationships, properties, interfaces, and behavioral models of a component. The archetype provides guidelines to designers in decision and trade-offs, helping to determine the requirements for components and aiding in the selection of appropriate components from the CML.

The core of the design archetype is the block definition diagram, where the components are defined. An internal block diagram defines the physical interfaces to the component. Parametric diagrams can capture the mathematical relationships between the properties of these components. State machines and activity diagrams can express high level operating modes. Other behaviors can be expressed in other tools, such as Simulink, but those behavioral model interfaces can be captured in SysML.

Design archetypes form a hierarchy. At the top level is the system being developed, such as a ground vehicle. The vehicle is composed of several high level components. These components have alternative design patterns, and so represent design choices.

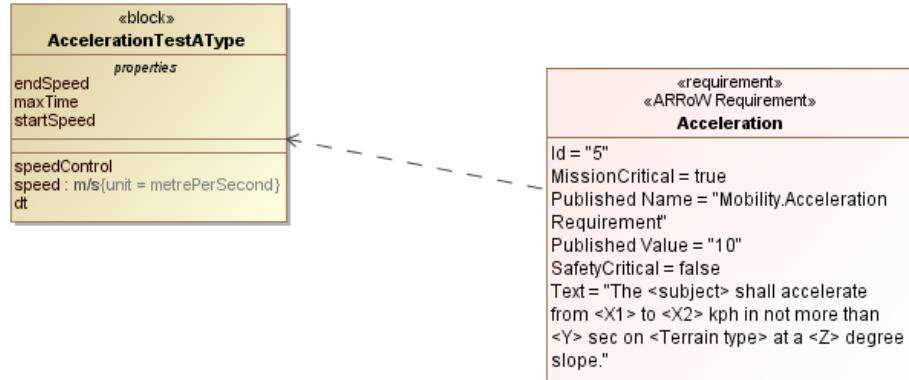
These archetypes are further defined in the remainder of this section.

### 7.3.2.3.1.3 Analysis Archetypes

Analysis archetypes capture the processes and methods used by designers. These are not currently captured as part of the reference architecture as captured in SysML.

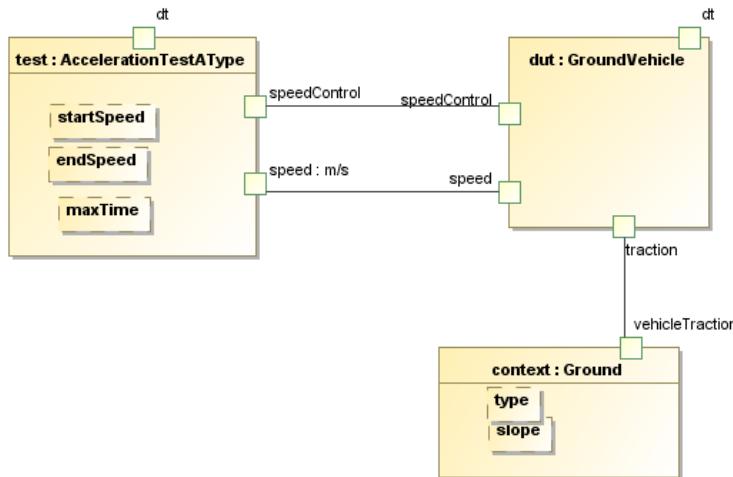
#### 7.3.2.3.1.4 Test Archetypes

Test archetypes define a pattern of tests to verify one or more system properties or requirements. For example, the acceleration requirement archetype defines the pattern of describing the acceleration requirement in various contexts. Figure 7.3-7 shows



**Figure 7.3-7. Acceleration Requirement Archetype**

The corresponding test archetype defines the system interfaces and test driver behavior needed to execute the tests, and to judge whether it passed or failed. The acceleration test (Figure 7.3-8) requires two interfaces of the design under test (DUT), a speed control (throttle) DUT input, and an output indicating the current speed. Properties startSpeed, endSpeed, and maxTime are parameterized and filled in from the requirement being tested, as is the Ground context component, e.g. pavement with 0% slope or sand with a 5% slope.



**Figure 7.3-8. Acceleration Test Archetype**

The behavior of the test driver and its interaction with the system is captured in a SysML sequence diagram (Figure 7.3-9). It basically sets the throttle to maximum, then monitors the current speed until it reaches the target speed, or the time expires, then validates the results.



**Figure 7.3-9. Acceleration Test Sequence Diagram**

These test archetypes are then implemented and populated with the project specific requirement parameters, the context, and the design under test to be executed. The results of these tests feed the PCC.

These tests can then be reused across a variety of design variations, given that the DUT interface is consistent. This test capability would be used to exercise the same set of test cases against all submitted designs as a check or input to PCC calculations.

### 7.3.2.3.1.5 Integrating Requirements, Design, and Test Archetypes

The relationship between requirements, design, and test archetypes is captured in the reference architecture and provides significant speedups in development time. As shown in the previous three sections, requirement archetypes are connected to the design and to the test archetypes that verify them. The test archetypes define required interfaces to the design and to the supporting context models. This pre-defined infrastructure allows designers to test and communicate their designs quickly and early in the process.

### 7.3.2.4 Archetype Development

Archetypes are stored in the component model library (CML). Peer-to-peer component connections link archetypes, as well as supertype-subtype links that represent decisions points, for example wheeled vs. tracked option for vehicle traction.

The set of archetypes is easily expandable. Designers can add new archetype entries to the CML. New components are modeled in SysML; an OWL ontology schema is derived from SysML and installed in CML; the new archetype is then added to the CML. The archetype is connected to an archetype hierarchy representing the system breakdown, either as a new component attached as a peer to an existing component, or a design alternative to an existing archetype.

New design patterns for components that already exist in the CML, like the powertrain, can be added as another design choice. By making the new archetype a subtype (refinement) of the existing powertrain archetype, other designers searching for powertrain options will find the option. By including applicability information (described below), automated reasoners can determine the design conditions under which the archetype should be applied.

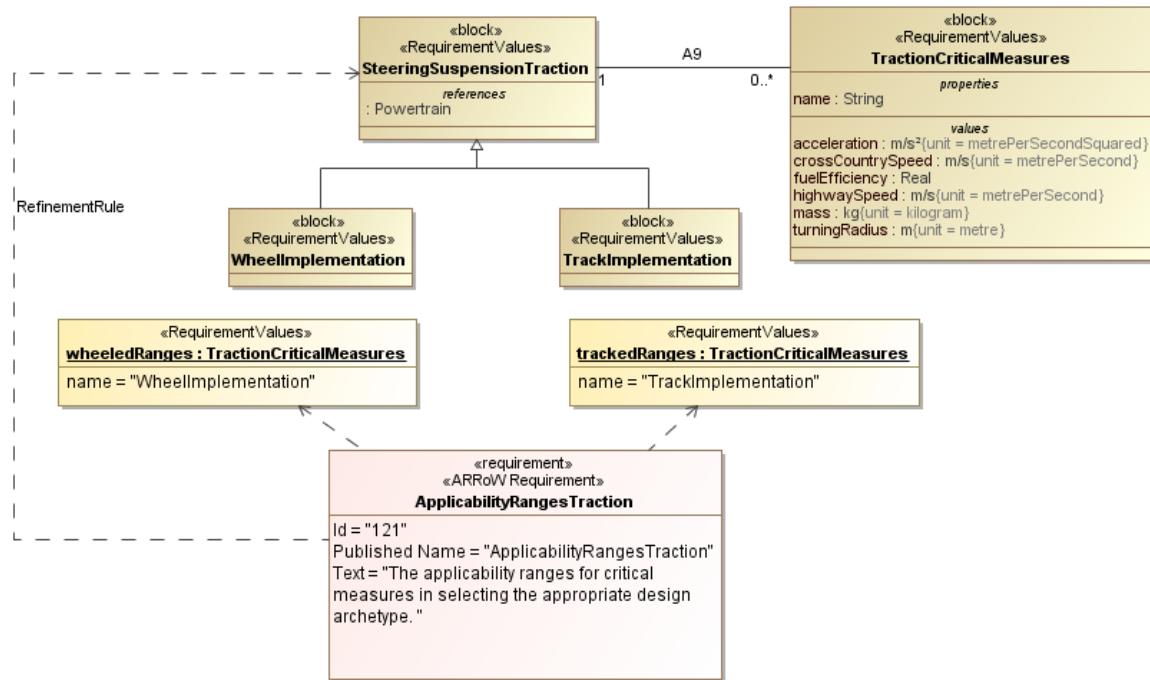
#### **7.3.2.4.1 Heuristics for Choosing Design Patterns**

The reference architecture captures design patterns and organizes them into a structure, all based on the experience of engineers in creating these components and systems. We can augment the archetypes with additional data that describes the tradeoffs between archetypes, and add reasoners to help narrow the design space engineers have to consider, or make recommendations. Thus, SysML provides one option for implementing design exploration automation. Including applicability data describing what contextual properties impact the decision and capture the range of values for which each option best applies allows such reasoners to be independent of any specific system under design or component tradeoffs and is general purpose utility that can operate on any system.

In general, such design exploration reasoners attempt to maximize performance while minimizing cost. Component specific measures determine performance, such as radio range and data rate, vehicle speed and acceleration, or engine power, torque, and efficiency. Cost is typically measured by SWAP (size, weight, and power) and financial cost, as well as some component specific factors, like heat.

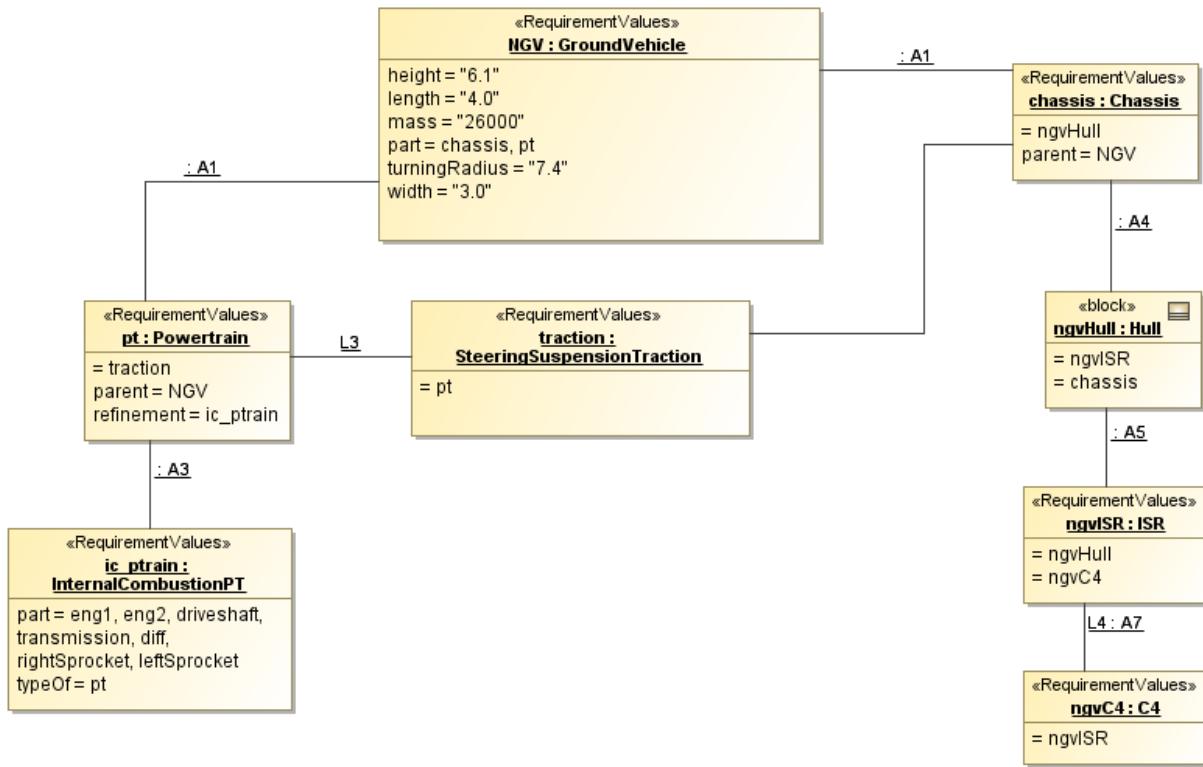
#### **7.3.2.4.2 Traction Example**

This section describes using the reference architecture, applicability data, and a reasoner within the AMIL graph to automatically select the best design pattern for the traction, suspension, and steering – a tracked or wheeled pattern.



**Figure 7.3-10. Design Archetype Applicability Ranges**

Figure 7.3-10 shows a design model with instances of the top level vehicle components. A top level archetype, **SteeringSuspensionTraction**, stands in for a design decision yet to be made – **TrackImplementation** vs **WheelImplementation**, which are shown as subtypes in SysML. The supertype defines a set of critical measures or properties of the vehicle design that affect the choice of subtype. These are captured in the **TractionCriticalMeasures** block. The subtypes define their applicable ranges of values for these measures – i.e. a wheeled vehicle is better for high fuel efficiency and low weight, while tracked is better for high weight and high off-road speed. A pseudo-requirement, **ApplicabilityRangesTraction** is the root node for the applicability data for each sub-archetype. A RefinementRule dependency connects the pseudo-requirement to the top level archetype to which it applies. These applicability ranges are stored in SysML the same way property ranges for requirements are captured, in the ARRoW dialog box on the instances of the applicability ranges, **wheeledRanges** and **trackedRanges** in the diagram. The name property is the return value, indicating the archetype to which the ranges apply.



**Figure 7.3-11. Ground Vehicle Design Instances**

When a supertype design archetype such as SteeringSuspensionTraction is used on an implementation diagram within a vehicle's design, it is a placeholder for a decision, either one of the archetypical subtypes or a new, manual design (see Figure 7.3-11). The user may right-click the block and select ARRoW>RefineArchetype to trigger the reasoner to make a selection, and add it to the diagram. The reasoner collects design values or the range of values for the property as captured by requirements and ranks the design alternatives.

The refinement algorithm pattern is data driven and therefore, plug-and-play. A new refinement archetype option, say half-track, can easily be added and included in the reasoning by adding its own half-track applicability criteria.

The refinement reasoner/heuristic algorithm is driven by meta-data. This means that the algorithm components can be easily applied to refinement algorithms for other design archetypes. The ranges and the references to the vehicle properties to which they should be compared are stored in AMIL and evaluated by the reasoner. The same reasoner can be configured to select engines or other components or patterns expressed in the reference architecture.

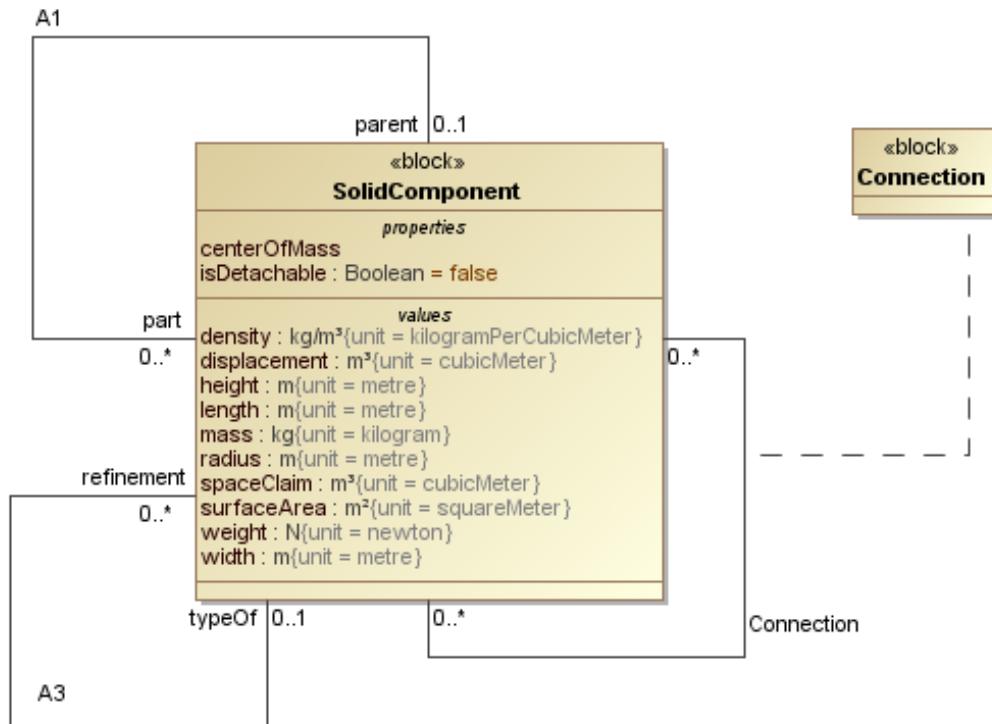
Thus, the reference architecture and the reasoner could become building blocks for a larger scale, dynamic design space exploration across multiple design patterns captured in the reference architecture as other requirements/design decisions are met.

### 7.3.2.4.3 Core Component Relationships

There are three core relationships between components leveraged throughout the reference architecture.

The SolidComponent represents a common supertype to all any non-fluid physical components in the reference architecture. Association A1 in Figure 7.3-12 captures the whole-part relationship between components. A component may be made up of multiple parts. A component made up of no parts is an atomic component, like a screw or bolt. Each part has one assembly as its parent component, except for the top level system being developed, which has no parent component.

The Connection association represents the peer-to-peer connection between components. Each connection usually has more information associated with it. A radio is connected to an antenna, but through a cable and connectors that interface the cable with the radio and antenna.



**Figure 7.3-12. SolidComponent Block and Core Relationships**

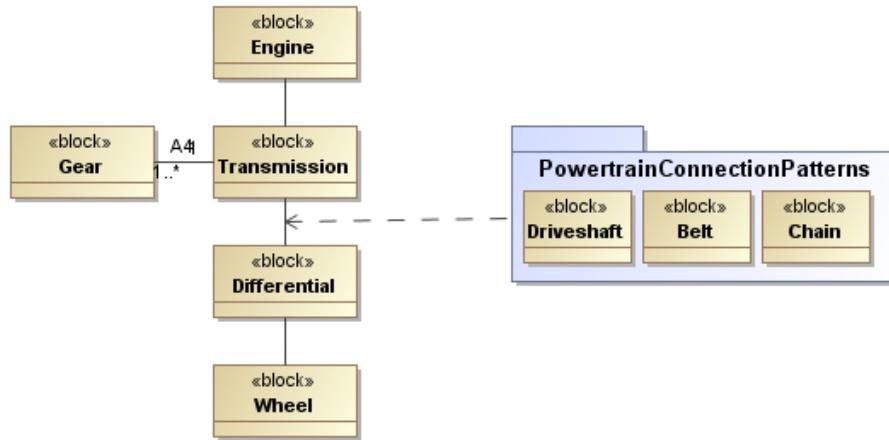
Association A3 describes the refinement relationship between archetypes, abstract components, and physical components. Archetypes like the traction subsystem can be refined into either wheeled or tracked implementations. An abstract engine can be refined to a gas, diesel, or electric engine, and then to a physical engine available in the CML.

### 7.3.2.5 Abstract Components

Abstract component is a placeholder for a real component from the component model library. The abstract component can be modeled, simulated, and marked with desired property values. The property values can then be used to query the CML to find components that match these desired values, assisting in the component selection process.

### 7.3.2.6 Design Refinement

The reference architecture defines the subcomponent types and interaction constraints within the parent component. Designers can make instances of these component blocks and link them together. For example, when refining the PowerTrain component, the designer uses the reference architecture model and creates instances of the engine, transmission, and differential. Figure 7.3-13 illustrates this.



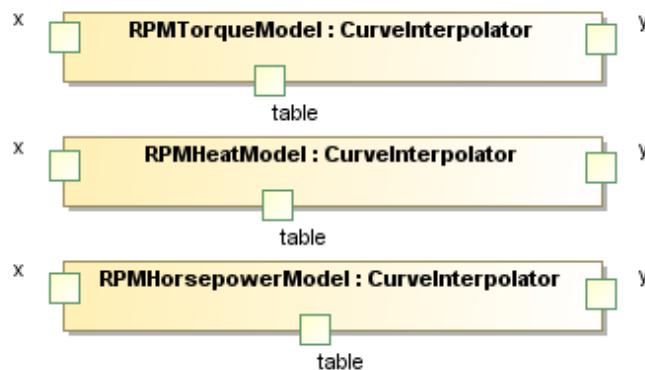
**Figure 7.3-13. Power Train Design Instances**

The reference architecture defines connection patterns for transferring mechanical power. Users can utilize these in making design choices. Different configurations, such as front wheel, rear wheel, and four wheel drive can be created.

At this point, the designer has created a model of the power train components and connections. He can now start assigning values to the block properties, such as engine power, torque, and mass, that will be used to understand the design trade-offs.

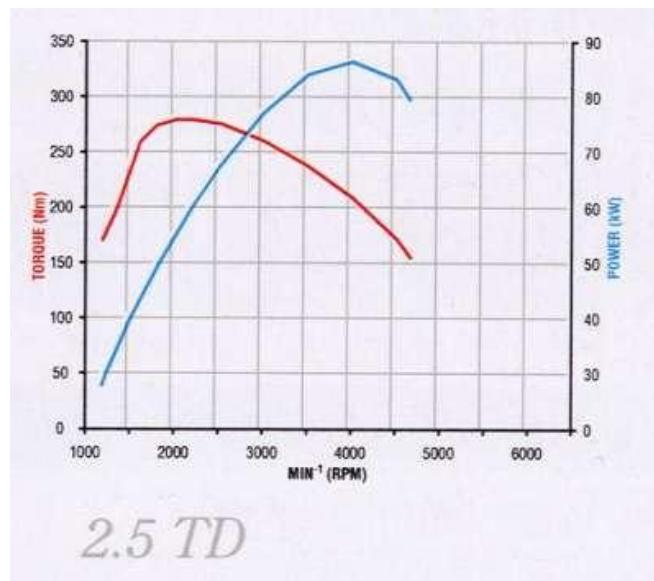
#### 7.3.2.6.1 Default Behavioral Models

The reference architecture includes behavioral models that can be used in early simulation/verification. For example, simple behavioral models representing the power, torque, and heat curves vs RPM for an engine are shown in Figure 7.3-14. These models accept the RPM as input (x) and produce the corresponding output (y), given a lookup table (table). The table can be configured with the curves to describe the engine. These models can then be combined with other behavioral models to build a full model of the vehicle for use in quantitative simulation.

**Figure 7.3-14. Behavioral Models**

As the design is refined, behavioral models associated with specific concrete components can override the default behavioral models. Or, these curve-models can be re-used by supplying the appropriate table data, making it easier to add new components and models to the library.

The CurveInterpolator is a support class that does simple linear/spline interpolation of a set of points. Given curves that correlate the engine RPM to the generated heat, horsepower, and torque (Figure 7.3-15), these can be combined to make a low fidelity model of the engine.

**Figure 7.3-15. Example Engine Power and Torque Curves**

#### 7.3.2.6.2 Concrete Component Selection

Once the key properties of the abstract component are given design values and have been analyzed for system level tradeoffs, we need to find a component that matches those specifications in the CML. The CML is indexed by an OWL ontology schema, searchable with the ARRoW tools. Using the design and/or requirements values for the component, a CML query is generated that returns a set of matching CML entries. Selecting a component loads the remaining properties from the CML entry into the design, making it possible to analyze the side effects of the selection. For example, we can select an engine out of the CML based on

required power and torque, then load up the additional mass and thermodynamic properties and re-evaluate the design.

#### **7.3.2.7 Context Components**

Context components are the components that the system interfaces with in the environment. Context components are modeled with the same approach as system components. Context components are represented by blocks, with properties, interfaces, associations, and behaviors. Context components are also reusable across multiple projects and reference architectures for different systems.

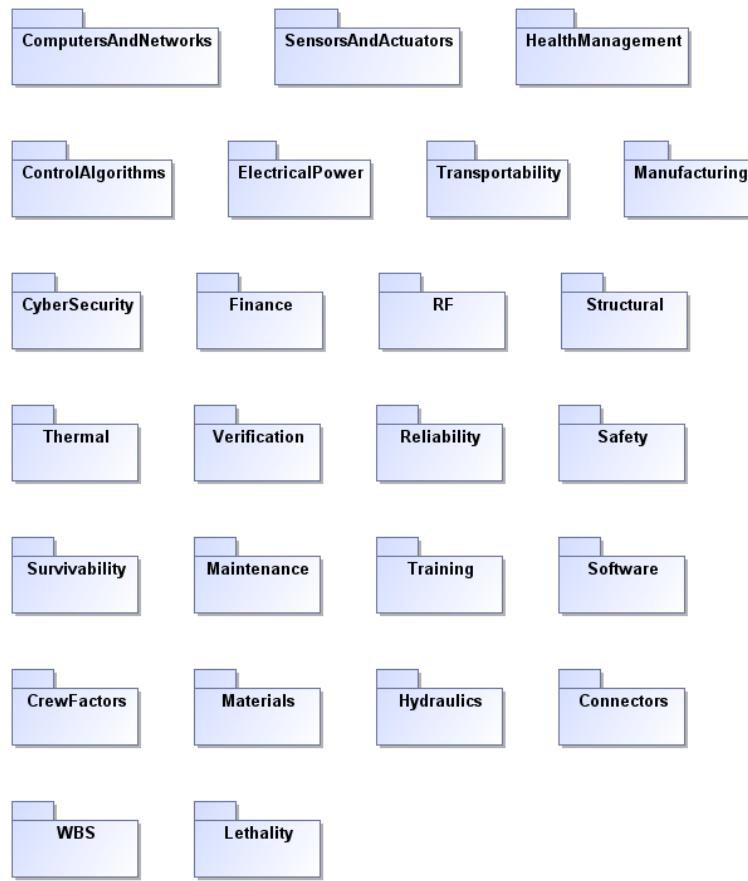
#### **7.3.2.8 Viewpoints**

A Viewpoint is a specification of the conventions and rules for constructing and using a view for the purpose of addressing a set of stakeholder concerns. They specify the elements expected to be represented in the view.

Viewpoints in the reference architecture tend to line up with domain specific engineering areas, such as electrical and radio communication. It is these viewpoints that connect the system architecture in SysML with the domain specific tools that engineers use. This connection, through AMIL, allows the design to be constantly evaluated for project specific tradeoffs to ensure objectives are being met.

Viewpoints are also reusable – their isolation from the details of the system under development and the focus on one subject matter make them reusable in other designs and reference architecture. The electrical viewpoint, addressing power consumption, can be used with ground vehicle or satellite reference architectures, without any changes.

Figure 7.3-16 shows the set of viewpoints related to the ground vehicle reference architecture.



**Figure 7.3-16. Ground Vehicle Viewpoints**

### 7.3.2.9 Parametrics

SysML formalizes the mathematical relationship between block properties using the parametric view. For example, the relationship of vehicle mass, engine power, track sprocket size, and maximum sustained vehicle speed can be captured in a parametric diagram, including equations. The toolset of Magic Draw, Paramagic, and Open Modelica can then be used to solve these equations in any direction – computing either target speed or required engine power, depending upon what values are filled in.

Parametrics is used within the reference architecture to maintain and adjust these mathematical relationships over the span of designs covered by the design archetype. For example, design choices between 2 wheel vs. four wheel drive, electrical vs. gas powered engines, and even two engine configurations should be analyzable using parametrics without having to rewrite large number of equations.

### 7.3.2.10 Relationship with CML

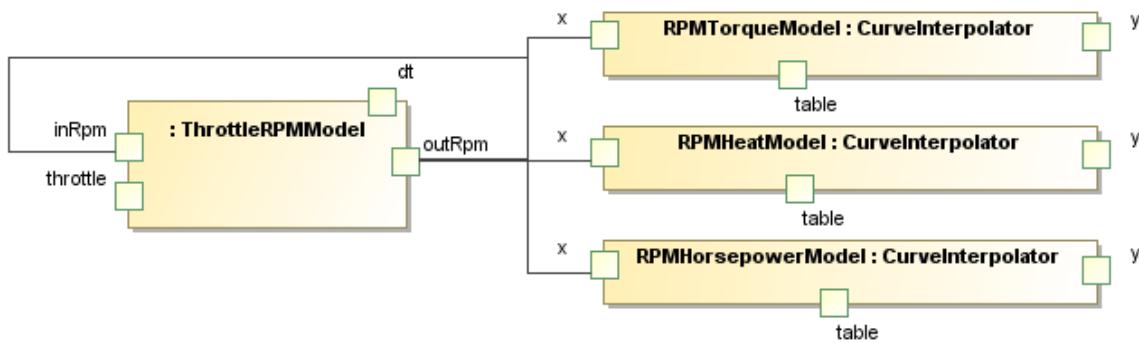
The reference architecture is closely tied to the component model library (CML). Components defined in the reference architecture are converted into an OWL schema and placed in the CML. Components of that type can then be added to the CML. In addition, the reference architecture components are also CML components and need to be published into and pulled from the CML.

### 7.3.2.11 Behavioral Models

A component may have multiple behavioral models, showing different aspects of its behavior and at different levels of fidelity. Behavioral models can be represented in SysML as blocks with input and output ports. The actual implementation of the model may be in another language or tool, but the interfaces can be captured in SysML.

Behavioral models of components may be composed to create a model of the larger system. These models can then be interfaced with the context models and connected to a test driver to execute quantitative tests on the system. Having multiple fine grained component models running in parallel under the same test allows the detection of emergent behaviors.

Connections between the output ports to input ports can support building a co-simulation model. Figure 7.3-17 shows behavioral models of an engine connected together.



**Figure 7.3-17. Combining Behavioral Models into a Co-Simulation**

### 7.3.2.12 SysML Usage

SysML is the Systems Engineering Modeling Language, an Object Management Group standard that extends UML notation for software engineering to cover systems engineering concepts. This section will describe how SysML is used within the reference architecture.

The following section describes the parts of SysML that were used in the reference architecture and in the design of a vehicle.

#### 7.3.2.12.1.1 Block Definition Diagrams

Block definition diagrams (BDDs) capture the components the owning archetype component, their properties and relationships. A BDD contains all the blocks that participate in the design of the containing component. Blocks instances defined here are used in the design of a component.

#### 7.3.2.12.1.2 Internal Block Diagrams

Internal block diagrams are used in two ways. First, they define the operational interfaces of a component. These are the external connections, including mounting, electrical, gears, gas or liquid flows, controls, messages, etc. The ports may also include side effect interfaces, like noise, heat, and vibration.

Second, IBDs document the interface to behavioral models. Each component in the CML may have multiple behavioral models. The models may be at different fidelity levels or consider different viewpoints. For example, one model of an engine describes the power and torque generated at different RPMs, while another thermal model captures the heat generated by the engine, heat carried off by coolant, and heat transmitted to the surrounding components and the air.

The ports on the IBDs serve to define the interfaces of these models. The designer can then compose system level tests by combining these blocks by connecting input and output ports.

#### **7.3.2.12.1.3 Sequence Diagrams**

Sequence diagrams capture the interactions of components over time. Within the reference architecture, they are used to document the contract between components and to define test cases.

#### **7.3.2.12.1.4 Requirements Diagrams**

Requirements diagrams capture, graphically, the relationships among requirements. For ARRoW, we used requirements diagrams to capture requirement archetype sets – sets of related requirements that must often be considered when describing a component’s features. For example, a ground vehicle’s mobility requirements address the acceleration, maximum sustainable speed, turning radius, and obstacle negotiation under various conditions.

Requirements are also shown on some BDDs, to show the mapping of the requirements onto the blocks which are affected by them.

For the ARRoW reference architecture, we extend the SysML requirements with additional data to allow it to be shared via AMIL.

#### **7.3.2.12.1.5 Parametric Diagrams**

SysML formalizes the mathematical relationship between block properties using the parametric view. Within the reference architecture, parametric diagrams express these relationships and allow changes in property values to be propagated, showing their effect on other areas.

#### **7.3.2.12.1.6 Implementation Diagrams**

Implementation diagrams are used to capture the design of the system, in this project, a military ground vehicle. Using the blocks defined in the block definition diagrams of the reference architecture, the blocks in the implementation diagram represent instance of the elements of the reference architecture. For example, while the BDD defines component types such as Engine and its properties, the implementation diagram describes a specific engine, with values for power, mass, and volume.

#### **7.3.2.12.1.7 Use Case Diagrams**

Use case diagrams capture the scenarios that the system or component is involved in. Use cases describe a story where the system/component interacts with its environment. The story usually captures the preconditions, the trigger that starts the interaction, a summary of the interaction, and the post-conditions. Pre- and post- conditions describe the state of the system and environment before and after the use case.

Use case diagrams then capture the scenarios the system is involved in. Usually, these start with a high level, concept of operations. A ground vehicle ConOps use case might to transport a squad through cross country terrain, at night, without refueling, with a limited time. This use case then derives a series of requirements, such as sustained cross country speed, range, and passenger transportation. The use case can also evolve into a system level test case, incorporating all the context components with the system design.

#### 7.3.2.12.1.8 Blocks and Associations

Blocks in the reference architecture represent types of physical components, such as engines, ramps, and wheels. Sometimes the blocks represent software. Within the viewpoints, a block may also represent non-physical concepts like a waveform or channel used in radio communications.

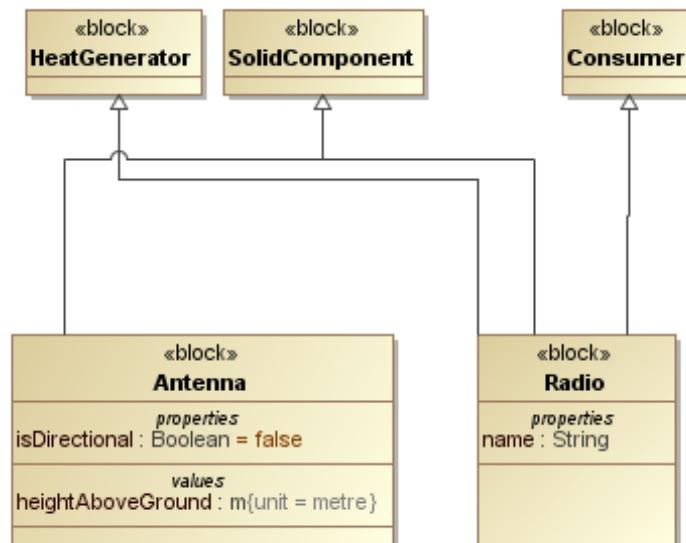
Associations in the reference architecture usually indicate a physical connection. The physical connection often has its own set of patterns to select from. For example, the throttle control, like a gas pedal, may be connected to the engine in a few different ways, by physical wire or by sensor and network communications to the engine computer.

Associations may have additional semantics that are in the description of the model element.

#### 7.3.2.12.1.9 Combining Viewpoints

Viewpoints are domain specific models within the reference architecture that are from a certain point of view. Viewpoints are usually aligned with the concerns from domain specific engineering. A physical component will often participate in multiple views.

For example, a radio draws power, so it has an electrical view. The radio has physical characteristics, size and mass, so it has a structural view. The radio also generates heat, so it has a thermodynamic view. Figure 7.3-18 shows a block diagram of these interacting viewpoints.



**Figure 7.3-18. Block Diagram Integrating Multiple Viewpoints**

Using viewpoints, it is easy to examine the vehicle from the electrical perspective to determine battery capacity required. All electrical components, their connections and characteristics are part of the reference architecture and can be extracted from the model to provide power budgeting for different scenarios. Knowing what equipment is needed in each scenario, with associated power requirements, then drives the battery requirements, or could result in a requirement that the engine be run (if a gas engine) to generate electricity to recharge the batteries and run equipment.

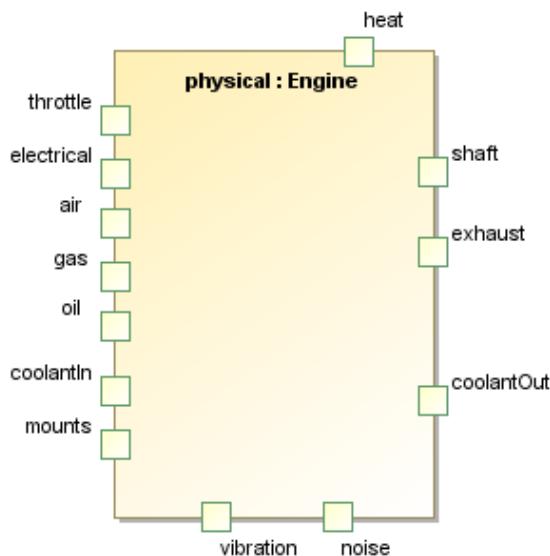
Similarly, the thermodynamic view can address the radio and other electronics within an enclosed area and determine the cooling needed to run them within optimal range.

#### **7.3.2.12.1.10 Interfaces**

Interfaces in the reference architecture are captured in the internal block diagram, using ports for the interfaces. Connectors (lines) show the connections between the components.

#### **7.3.2.12.1.11 Physical Interfaces**

The physical interfaces of a block are captured in an internal block diagram, with each port representing a possible connection to other components, including context components. Figure 7.3-19 shows an example of such physical interfaces.



**Figure 7.3-19. Physical Interfaces of an Engine**

Component interfaces represent contracts that the component and its context must adhere to. There are some interfaces that may be added by components that refine a reference architecture component. But an interface that is ignored is a possible issue with the design and could be a cause of emergent behaviors. For example, if the model fails to address the heat interface of an engine, it ignores the fact that an engine generates heat that radiates to the components around it. This heat may have unintended consequences.

#### **7.3.2.12.1.12 Model Interfaces**

A component's behavioral models also use the internal block diagrams to document its interfaces. The model may be implemented in Matlab, Modelica, or C++ code, but the interfaces are captured in SysML to enable composition of these models.

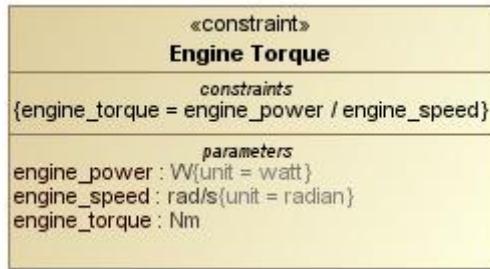
### 7.3.2.12.1.13 Properties

Properties are the characteristics that describe a component. All physical components have associated properties such as a bounding box that represents spatial dimensions, mass, and material composition. These are all shown in the model as properties. The reference architecture captures all the characteristics that can describe the component.

Ontological technology underlies the reference architecture. Unlike other database technologies that require every property to be filled out for every entry, and ontology regards properties as optional. While it is true that all engines have a mass, that mass is not always known, and the ontology allows the property to be omitted. Of course, this limits the usability of the entry, but it is still permitted.

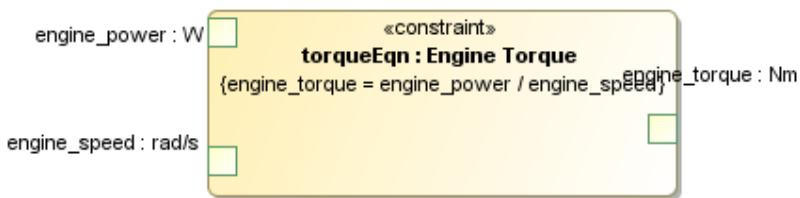
### 7.3.2.12.1.14 Constraints

Constraints are used with the parametric diagrams to capture the mathematical equations that constrain the values of properties within a set of block instances. Figure 7.3-20 shows an example of such a constraint.



**Figure 7.3-20. Engine Torque Constraint Equation**

The constraint is defined once, as a block with a <<constraint>> stereotype. The block defines the parameters and mathematical equation. The constraint handles only one equation per block, and can have only one output value.



**Figure 7.3-21. Usage of Engine Torque Constraint Equation**

The constraint is used in parametric diagrams which use the ports and connectors style of diagram like the internal block diagrams. Figure 7.3-21 shows an example of this.

### 7.3.2.12.1.15 Connector Patterns

Many alternatives exist for connecting components together. For example, a driveshaft, belt, or chain can transmit mechanical power from an engine to wheels on a motorcycle. Each option has applicability and constraints/limitations, and the design context must be evaluated to select the best choice. Cost, maintainability, complexity, reliability, and capability drive the pattern selection. As with any component, connector patterns can be expressed abstractly in SysML, and yet have corresponding models in CAD or other viewpoints.

Connector patterns also have applicability measures, just like components. A belt is not going to be strong enough to move a 20 ton vehicle, while a sufficiently strong chain would have a larger space claim than a driveshaft. However, for smaller vehicles, like a bicycle or motorcycle, the tradeoffs are different. A driveshaft is less efficient than a belt or chain, but easier to maintain.

The connector pattern is one that occurs throughout the reference architecture. Whenever two high level components are connected, there will be a connector pattern and set of options between them. The connector pattern also incorporates a set of components that are used to connect the high level components together. The pattern then operates recursively, until you get down to the nuts-and-bolts level.

For example, an antenna is connected to the hull of a vehicle. But the antenna does not attach directly. A mounting bracket connects the antenna to the hull. The bracket is then connected to the hull with nuts and bolts, while it is connected to the antenna with clamp and screws.

### **7.3.2.12.2 Language of Design**

#### **7.3.2.12.2.1 Block Instance Diagrams**

Called Implementation Diagrams in Magic Draw, these capture the design of a component using instances of reference architecture blocks defined in the component archetype. Links between instances correspond to the associations in the reference architecture. The properties of instances can be assigned values. These values represent design choices, as the requirements values are captured using a custom dialog. Figure 7.3-22 shows an example of this.

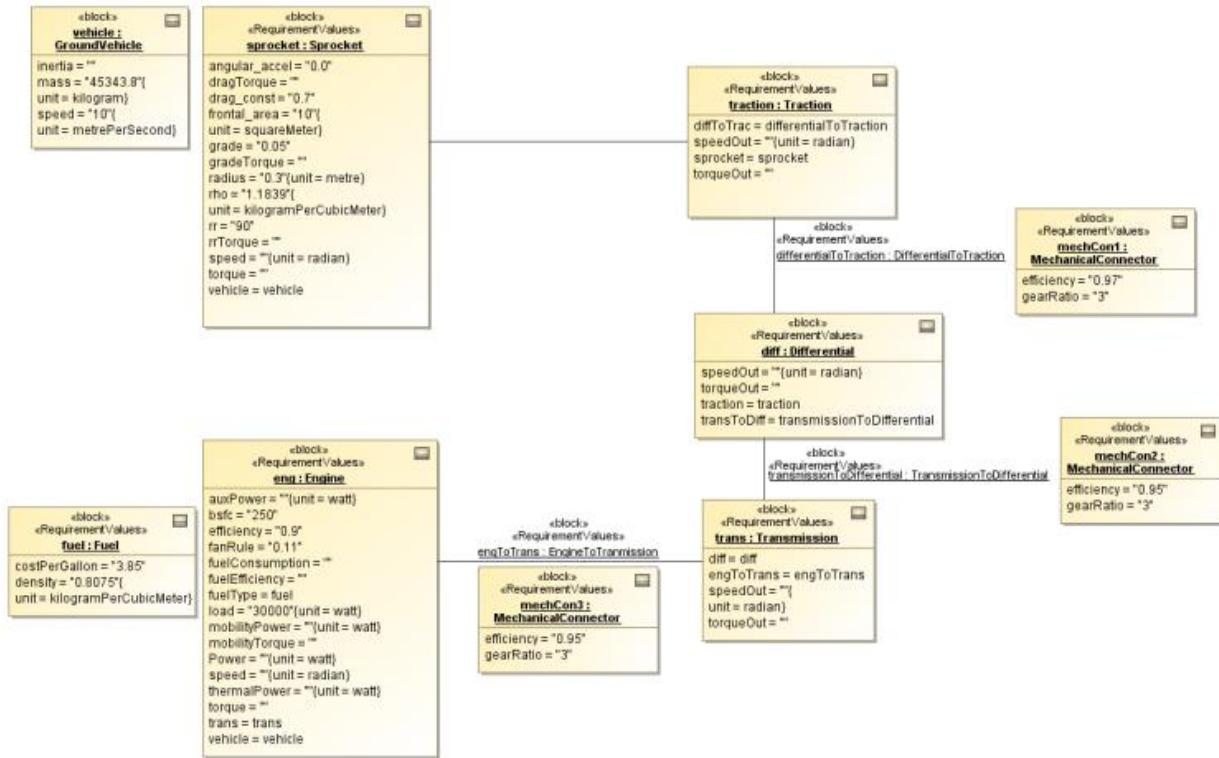
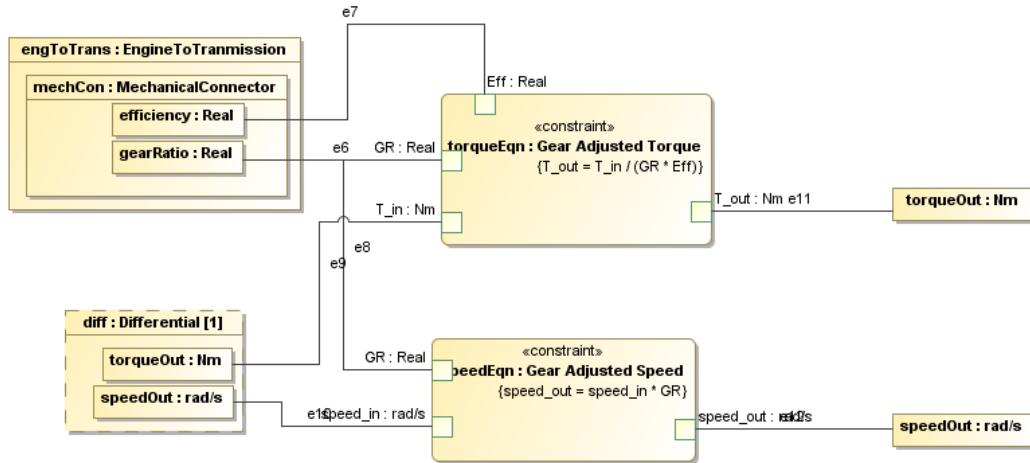


Figure 7.3-22. Power Train Design Instances

### 7.3.2.12.2.2 Parametric Diagrams

Parametric diagrams are used to define the mathematical relationships among the properties of the design block instances. These relationships are built up by chaining constraint equation blocks together and connecting them to input and output properties.

Once these relationships are defined, values can be assigned to the properties. When enough values have been assigned, tools can solve the set of equations for the unassigned properties. Used this way, the parametric diagrams can be useful for design space exploration. With some additional work, two and three dimensional graphs can be made to better visualize the tradeoffs in the design space. Figure 7.3-23 shows a parametric diagram.



**Figure 7.3-23. Engine Torque Parametric Diagram**

#### 7.3.2.12.2.3 Block Instances

Instances are used to capture a specific component and properties within the design. Thus, there is a type of combat vehicle called a Bradley, there are many instances of a Bradley. The Bradley with serial number 12345 specifies a particular instance of the vehicle. So the SysML block represents the type of component, while an instance represents a particular one.

#### 7.3.2.12.2.4 Property Values

Properties of block instances can have values assigned to them. Each phase of development places different values and views on the properties. In order for the components to support the requirements, design, implementation and test phases of development, the language supports multiple values for properties, each with a different viewpoint.

For example, requirements define a desired range of values for a property, possibly under certain conditions. For example, a vehicle may have a max speed of 60 mph on flat pavement, and a max speed of 23 mph on cross country terrain. The ARRoW tools support setting multiple requirements values on a property, one value range per requirement.

Design values can also be assigned to a property. These are single values rather than the ranges captured for requirements. Design values are used with parametric models to determine the effects these design decisions have on other properties and components. Multiple design values are often applied to properties to evaluate different alternatives. The current tools only support one design value at a time.

In addition, properties will have an “as built” value that describes the final measurable value of the property for the built vehicle. This may differ from the design and requirement values. Since the scope of the META program covered only through design time, the “as built” value is not currently supported.

The User’s Manual describes how property values are captured in SysML and mapped to the AMIL graph.

#### 7.3.2.12.3 ARRoW SysML Extensions (stereotypes)

This section describes the extensions made to the SysML language to support ARRoW. UML/SysML provides stereotypes as part of an extension mechanism called a profile. Stereotypes change the semantics of the UML/SysML model element to which they are applied, and can add tagged data in the form of name-value pairs to the model elements.

### **7.3.2.12.3.1 RequirementValues**

The RequirementValues stereotype is applied to both blocks and block instances. It is used to capture the impact that requirements have on block properties. Multiple requirements may apply to each property, placing different, sometimes conflicting, ranges of values on the property.

The stereotype has a single property, attributeValues. This stereotype tag holds a string holding the table of attribute value ranges and the requirements they derive from, in XML format. A custom dialog was added to the SysML tool to support the collection of the data stored in that string. See the RequirementValues section of the User’s Manual for more information. Figure 7.3-24 shows a RequirementValues Stereotype.



**Figure 7.3-24. Requirement Values Stereotype**

### **7.3.2.12.3.2 ARRoWRequirement**

The ARRoWRequirement stereotype captures additional information associated with a requirement.

PublishedName and PublishedValue tags are published to the AMIL graph. The PublishedName becomes the AMIL node name, while the value is an early version of the ranges of values now captured in the RequirementValues stereotype. Figure 7.3-25 shows an ARRoW requirement.



**Figure 7.3-25. ARRoWRequirement Stereotype**

The other attributes capture details about the requirement or requirement archetype that are important to other aspects of systems engineering. AllocationRationale helps capture the provenance of what a requirement exists and how it connects with design elements and other requirements. Maturity indicates if this requirement is something an organization is familiar with or is new and therefore implies higher risk that needs to be managed. Safety and Mission-

Critical tags assert that there is additional engineering work around the requirement to ensure reliability and safety, and indicate that these requirements are likely to significantly affect the PCC of the design.

#### 7.3.2.12.4 Integration With AMIL

AMIL is the means by which modeling tools share information within ARRoW. Model data can either be published into the AMIL graph (pushed) or AMIL can extract (pull) the data from the tool when requested by an AMIL client. Whether the data is pushed or pulled is invisible to the client requesting the data (except that it might take a bit longer in the pull case).

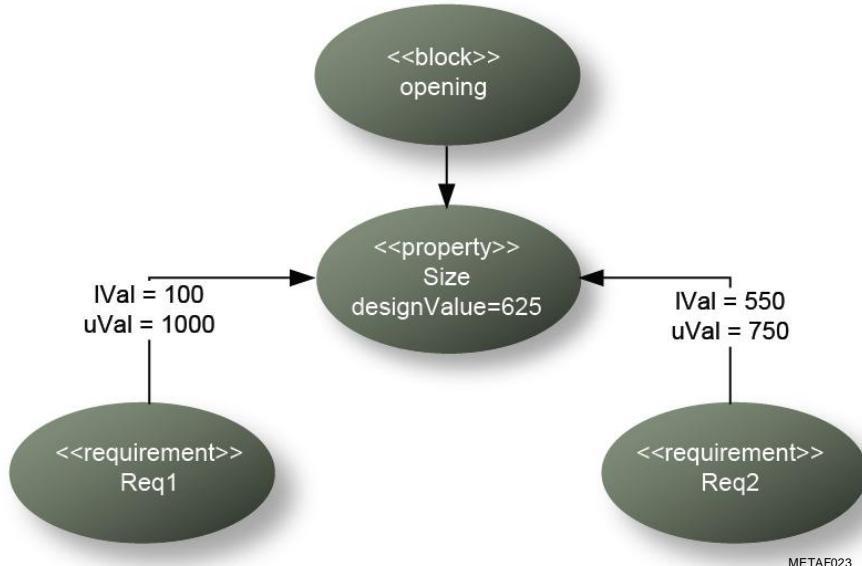
##### 7.3.2.12.4.1 Write

The SysML plug-in to Magic Draw uses the push method to share its data into AMIL.

Blocks, instances, properties, and requirements are each mapped to their own AMIL node. The arc from block to property represents the ownership of the property by a block. Attributes of the property, such as type, description etc., are not shown.

The arcs from requirements to property represent the constraints or desired values of the property, and the range expression is mapped to attributes of the arc. Here, the IVal and uVal ranges are included on the arcs.

Design values are assigned to a property and mapped to the AMIL graph using a *designValue* property on the SysML property's AMIL node. Figure 7.3-26 shows how this would be represented in AMIL.



**Figure 7.3-26. AMIL Representation of Requirement Ranges of Values**

##### 7.3.2.12.4.2 Read

The ARRoW SysML tools support reading back *designProperty* values back from the AMIL graph, allowing tools to exchange design information. Since SysML is the authoritative source for requirements in our application of ARRoW, requirement values are not read from AMIL.

Blocks are not read from the AMIL graph, but this is a possible future enhancement. This would allow tools such as ECTo to add components to the design, and communicate those blocks back to the SysML viewpoint. The reverse flow could also be supported, allowing components to be added in SysML and added to ECTo for 3D visualization and allocation.

### **7.3.3 AMIL**

The top level goal of AMIL is to automate in a rigorous fashion the joint use of tools, solvers, and reasoners that are specialized for different parts of the design challenge and have very different, possibly incompatible, syntax and semantics. To that end, AMIL is less concerned with individual models and more about how the models are used by the tools, solvers, and other specialized reasoners to accomplish specific tasks in the design and analysis phases. However, before one can understand how these models are used one needs to understand what these models are. For this reason, AMIL must incorporate some meta-modeling to characterize the models and must also incorporate relationships between the models that characterize how the models are used.

#### **7.3.3.1 Uses of AMIL Analysis and Verification**

The point of analysis is to verify that the design can achieve the intended effect, but the effect is to be realized in some environment. The relation between the designed component and its environment must be captured in the analysis to verify that the component has the desired behavior. In Contract Based Design (CBD), this is checking that the contract of a component is *compatible* with its environment [AB11]. Analysis may also be used to verify vertical relations such as *refinement*, where in CBD the contracts are used to check that the implementations of a contract also satisfy that of the refined model. One role of AMIL in analysis is to capture these relationships.

#### **7.3.3.2 Design Support**

Design of a complex vehicle will proceed in stages, starting with a high level, abstract, concept that get successively refined into a completely defined vehicle.

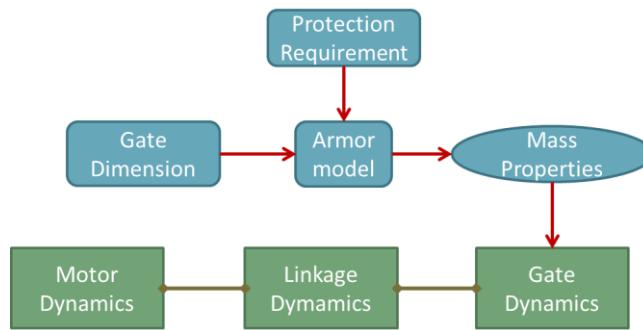
This process is captured in the system design artifacts, in which functions get implemented with systems and subsystems that themselves get implemented by interconnecting individual components. The component model library facilitates this process by establishing a hierarchy of abstraction between components and functions. As a system is refined, not only does the analysis verify the refinement, but corresponding changes in the setup of the analysis may be necessary. AMIL captures this analytical setup through analysis archetypes and provides direction on which links in the setup may require updating.

#### **7.3.3.3 Chaining of Tools**

Consider a mechanical linkage, such as an actuator arm that operates a gate. The system view, together with a 3d design can show all the individual components (bars, fasteners articulations, ball bearings) and how they are interconnected. Given dynamical models of each of those components it is possible to derive a dynamical model of the interconnection by translating the system interconnections into dynamical ones. However the dynamical models of each of the components will themselves be composite. For example, the bending modes of the links may need to be computed individually or jointly, by integrating the corresponding partial differential equations in an appropriate tool, before the dynamical model can be executed. This

relationship between the models allows computation of bending modes, and the model that computes dynamic behavior is not a system relationship, nor can it be easily derived from them. However, bending modes are reusable, and can be formalized.

For example, when computing the dynamic behavior of the vehicle gate, we need to update the dynamical model with the mass properties of the gate. Those can be estimated from the shape of the door, and the type of armor used, and the type of armor used can be estimated from protection requirements.



**Figure 7.3-27. Relationships Among Models and Design Elements**

The existence of such a chaining of tools that can be used to inform the gate dynamics model cannot be directly inferred from the system diagram. If sufficient declarative knowledge about all the entities involved is available, it is conceivable that we could deduce that the gate dynamics need to know the mass properties, that those can be approximated from the gate dimensions and the armor model, and that the armor model depends on the protection requirements. Finding the right string of tools can be complicated, even with a formal definition of each of the models involved.

An alternative approach is to describe the pattern of links between the analysis components in a formal language. AMIL can be used to define the links between the components and indicate how the different analysis tools can be interconnected to answer specific analysis questions. In Figure 7.3-29 the red links are AMIL links and the green links are system links. In order to carry out the analysis (in this case simulate the system behavior) both types of links are needed.

Consider the example in Figure 7.3-27. The first time it is setup, design engineers will select the models necessary to build the analysis construct. Engineers will also create the interconnection between the links. The most common type of link would indicate data transfer: a value generated by one model should parameterize another; or a signal generated in one model is fed to a port in another. However they could indicate simply that one model informs the other.

#### 7.3.3.4 Analysis Reuse

AMIL is designed to facilitate reuse, not only of analysis components like the ones in the example, but most importantly of analysis constructs.

Once the analysis structure has been built and used for one design, it will get persisted in the library of designs. The network of interconnections can then be used as an analysis template. In order to build the analysis graph for another similar problem, we start with the auto-generated part of the diagram (the green boxes). We then analyze which parts of the graph are still

undefined (in this case we find out it is the mass properties). Next we search the library of analysis graphs for similar cases in which the corresponding dynamics block had an external definition of mass properties. One we retrieve feasible options; we can follow the AMIL links to the other elements necessary for the AMIL graph. We expect that in general we will find many possible solutions. Selecting the most appropriate one can be done either manually or automatically.

### 7.3.3.5 Semantics of AMIL

AMIL records links and dependencies between various model types, to assist the creation of analysis packages. The semantics of AMIL links are defined with respect to the particular analysis product or process being carried out. By letting the link semantics be polymorphic, we create a simple mechanism for the language to be naturally extended as new analysis capabilities are introduced.

The semantics of an AMIL link are defined by the set of inferences that can be made based on the link, while executing different analysis or design processes.

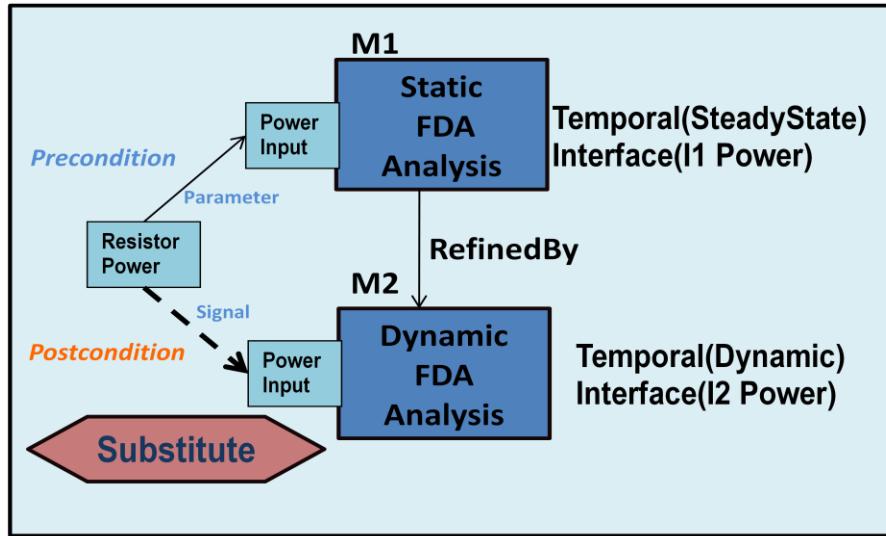
From the Link Semantics we are able to make inferences about the design during the development process. These inferences are based on the existence, type, and data content of the links between nodes. Some of the link types include *informs*, *partOf*, *refines*, *isA*, *instanceOf*, and *uses*.

One useful inference is to conclude the set of all models on which a test is dependent. Defining certain link types to be transitive allows one to derive the transitive closure of AMIL nodes from a node that represents the test. Similarly, one can trace back from a node representing an analysis result to the nodes representing the requirements that are related to this result.

Substitution of models is a common activity in the development process. Being able to infer new or updated relations between a model and its environment facilitates an automation of the overall process. In Figure 7.3-28, a new Signal link is inferred between the Resistor Power and the Power Input of the Dynamic FDA Analysis model. The inference rule used here is the following:

```
Action(Substitute) ^ RefinedBy (M1, M2) ^ Temporal(M1,
    SteadyState) ^ Temporal(M2, Dynamic) ^ Interface(M1, I1) ^
    Interface(M2, I1) ^ Parameter(R, M1.P) ➔ Signal(R, M2.P)
```

The predicate *Parameter(x, y)* represents the existence of an AMIL link between AMIL nodes x and y and the type of the link is *Parameter*. Similarly, the *RefinedBy(x, y)* predicate indicates the existence of an AMIL link between AMIL nodes x and y and the type of the link is *RefinedBy*. The *RefinedBy* link definition constrains the types of the nodes linked to be nodes representing models.



**Figure 7.3-28. Example of Inferring a Link.**

### 7.3.3.6 AMIL Syntax

The underlying structure of AMIL is based on an attributed graph. The syntax of AMIL is designed to edit and manipulate this structure. The AMIL syntax uses a JSON notation for its simplicity and readily available parsers.

An AMIL Statement is executed by the AMIL interpreter, which responds with a return value that is also a JSON formatted string. The following is the grammar for an AMIL Statement and its parts.

```
AMILStatement =>
[Action, SequenceOfNodeOrEdge]

Action =>
{
    action : actionName,
    preConditions : {predicateList},
    postConditions : {predicateList}
}

actionName =>
"createNodes" | "createLinks" | "getNodesRaw" | "getNodes"
| "getLinks" | "deleteNodes" | "deleteLinks" |
"updateNodes" | "clearDatabase"

PredicateList => PredicateCall | PredicateCall,
PredicateList

PradicateCall => predicateName : [parameterList]

ParameterList=> String | String, ParameterList
```

```

SequenceOfNodeOrEdge=>
    NodeOrEdge,
    NodeOrEdge, SequenceOfNodeOrEdge

NodeOrEdge=>
{
    type : nodeOrEdgeType,
    uniqueId : {nameValueList}
}

NameValueList=> name : value | name : value, NameValueList

NodeOrEdgeType=> node | edge

```

### 7.3.4 Qualitative Modeling Language

Our Qualitative Modeling Language, QML, has been designed to describe the topology and behavior of composite models with connected sets of components. Simulation of the behavior of such models is useful in providing guidance to users in early stages of the design process.

Four high-level design tasks have been identified: Description, Synthesis, Simulation, and Analysis. QML supports all four. This section describes the language, how it helps in design, and some results we have obtained in using this modeling framework. Later sections exemplify the features described.

**Description:** Atomic components are the basis for building a systems model. QML has a language to describe the interface, and the qualitative behavior of the component. Component behavior is defined through qualitative algebraic and differential equations relating the qualitative variables.

**Synthesis:** QML has language constructs that describe how to compose these components into larger models using named nodes as connection points and specification of the components connected to those nodes. It also describes initial conditions for the components and initial modes of behavior.

**Simulation:** The test environment of a composite system is defined in the same way as a composite model, with initial conditions that can be imported from a context model. It also specifies how the environment behavior can evolve over time through transitions of modes that represent system state. See the description of the IFV Ramp below.

**Analysis:** Simulation creates a representation (called an envisionment) of all significantly different qualitative states that the system can reach in the test environment. The states in the envisionment can be evaluated with respect to whether they satisfy qualitative versions of system requirements. Paths with failed requirements require attention in more detailed design.

QML has a relatively small library of atomic component models. To grow this library, and to make it align with a current quantitative modeling language, we have built a translator that imports a Modelica library, and converts it to QML. Not all of Modelica constructs can (or should) be mapped into QML. For example, algorithms and functions that define behaviors in terms of programs rather than equations are not included.

### 7.3.4.1 Qualitative Simulation in Early Design

Qualitative simulation may be used in a variety of ways by a designer or automated design tool. The most common would be to simulate a system through a set of use cases with respect to particular requirements. For each use case, the simulation can be used to provide a qualitative answer for each requirement in each use case. A qualitative evaluation of a requirement can be **yes** (all choices for numeric parameters will satisfy the requirement), **no** (no choice of parameters values will satisfy the requirement), and **maybe** (some set of parameters will satisfy the requirement). Using qualitative simulation in this manner can focus quantitative reasoning on the **maybe** cases and throw out designs that cannot possible work.

### 7.3.4.2 Qualitative Representations

Qualitative simulation, or envisionment, is the process of projecting forward from an initial situation, and a model, all possible states that may occur. Central to qualitative simulation are qualitative representations of continuous quantities. Our approach to representing quantities begins with the sign algebra. Each quantity and its derivative is represented as one of four qualitative values  $\{-, 0, +, ?\}$ , where  $?$  represents an ambiguous value. For example, we could say that the voltage across a resistor is  $+$  and its derivative is  $-$ , to mean that the voltage is positive and decreasing. Qualitative simulation would infer in this situation that there is a potential transition from positive to zero voltage. Frequently, additional distinctions must be made for a given quantity. To account for this knowledge, we use *landmarks*, i.e., constant points of comparison, to introduce intervals for quantity values. For example, the substance is between its freezing point and its boiling point. A qualitative state is an assignment of values (intervals) to quantities and their derivatives.

To represent equations, we use qualitative constraints as follows. We represent the mathematical relationship for an ideal resistor  $V=IR$  as a constraint on the qualitative values of  $V$  and  $I$ . In any situation, the sign of the voltage across the resistor must equal the sign of the current through the resistor. We also include algebraic constraints for qualitative addition and multiplication. Finally, we have constraints that enforce continuity from calculus. That is, a value cannot change from  $-$  to  $+$  without being  $0$ , unless there is a discrete change in the system.

Suppose the derivative remains positive ( $+$ ). Then the value will equal  $0$  for an instant. In general, the system's behavior corresponds to an alternating sequence of intervals and instants; a situation is the set of qualitative values that state variables take on for each interval/instant. During an interval each variable remains within a single qualitative region. The end of one interval and the beginning of the next is marked by one or more variables transitioning between qualitative regions [BCW84]. Moving from an interval to an instant, a variable's new value is predicted by old value plus its derivative; the same is true when moving from the instant into a subsequent interval. Discrete changes can happen in the system if a component has a mode change. A mode can reset values and change the equations by which a quantity evolves.

### 7.3.4.3 Specifying Qualitative Models

We develop a qualitative modeling language (QML) to specify qualitative models. We use models of a resistor and a diode shown in Figure 7.3-29 to illustrate the various aspects of QML.

```
(defprototype ideal-resistor
  :external-terminals ((t1 :electrical) (t2 :electrical))
  :fixed-parameters ((R))
  :variables ((v (voltage t1 t2))
              (i (current t1)))
  :equations ((q= v i)
              (q= (deriv v 1 t) (deriv i 1 t))
              (= v (* R i)))
```

**Figure 7.3-29. Definition of an Ideal Resistor including Both Qualitative and Quantitative Equations**

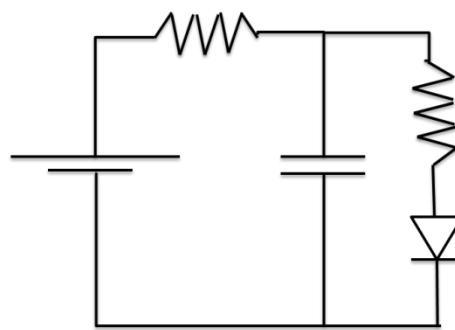
Atomic components are required to have external-terminals, variables, and equations. External terminals define how the component can be connected to others, variables define quantities of importance to the device, and equations relate these quantities to each other. The operator q= defines a qualitative equality constraint. In this case, the voltage and current must have the same sign. Fixed parameters and = are used to define quantitative equations.

```
(defprototype ideal-diode
  :external-terminals (t1 t2)
  :variables
  ((v (voltage t1 t2) :landmarks (Q OnVoltage))
   (i (current t1)))
  :mode (off :entry ((q= i Q0))
           :initial ()
           :equations ((q= i Q0)
                       (q= (deriv i 1) Q0)))
  :mode (on :entry ((q= v OnVoltage0))
           :initial ()
           :equations ((q= v OnVoltage0)
                       (q= (deriv v 1) Q0))))
```

**Figure 7.3-30. Defining an ideal diode requires landmarks to divide the quantity space into additional intervals, and modes to model discrete transitions.**

Figure 7.3-30 contains an ideal diode that illustrates how discrete transitions and intervals are defined. A diode has two landmarks for voltage,  $Q$ , which represents the distinction between + and – voltage as usual, and *OnVoltage*, which denotes the voltage at which the diode turns on. The instant at which the diode turns on represents a discrete, or discontinuous, change and is modeled using modes. Each mode has an entry condition, which is a conjunction of qualitative equations that, when satisfied, cause the component to transition into the mode. On entering a mode, the initial conditions set values of local variables. As long as the component is in the mode, the constraints defined by the mode equations must hold. For example, the entry condition for the *on mode* is satisfied when the voltage is equal to the *OnVoltage* landmark. If one wished to express that the voltage was above the *OnVoltage* landmark, the equation would be ( $q = v \text{ OnVoltage}+$ ). This initial and equation statements for this mode ensure that this value for voltage is maintained as long as the diode is in the *on* mode.

```
(defprototype diode-test
  :internal-nodes ((input :electrical)
                  (n1 :electrical)
                  (n2 :electrical)
                  (ground :electrical))
  :components ((D (ideal-diode n2 ground))
               (C (ideal-capacitor n1 ground))
               (R1 (ideal-resistor input n1))
               (R2 (ideal-resistor n1 n2))
               (B (ideal-battery input ground))))
```



**Figure 7.3-31. Composition of components in QML defining a system involving a diode (shown graphically on the right).**

Figure 7.3-31 illustrates how these components can be synthesized into a design. The topology of the system is defined by creating a set of nodes to which the components are connected. Nodes have a domain type, e.g., :rotational, :linear, :thermal, :electrical. The nodes from each domain are used to calculate constraints resulting from Kirchhoff's current and voltage laws. The list of components names and instantiates models with particular connections to nodes. In this example, the only domain is electrical. The first component in the list is the diode, which is connected to node n2 and n3. The resistor R2 is also connected to n2, and the capacitor and battery are also connected to n3. In the next section, we move away from circuits to show how a model can include the environment's behavior, and illustrate how an envisionment of the model can be used in analysis.

#### 7.3.4.4 Example: Analysis of IFV Door Opening and Closing

Figure 7.3-32 contains the definition of the system for an IFV ramp door opening and closing with a PID controller. The first three components model the output of proportional and derivative control as well as their sum (Note: Integral control is ignored as it is not interesting qualitatively). Given a control output, a piston produces a torque on the door. The last two components are sensors which produce signals concerning the door's position and velocity relative to particular landmarks. Next, the initial conditions of the system are specified. The operators == and qIn are used to specify the mode of a component and the interval of a quantity. In this case, the initial mode of the system is opening, and the initial mode of the velocity sensor is Q, that is, the value of the velocity sensor is determined by the door's velocity measured in respect to the landmark Q or 0 rad/s. The initial position of the door is between the Closed and Open landmarks, and the initial velocity of the piston is in the + direction. The system has two operating modes, which are used to initialize the mode of the position sensor used by the p controller as well as the controller output. In the next section, we describe what the resulting envisionment looks like and how it can be used to guide design.

```
(defprototype sept-demo
  :internal-nodes ((n1 :variable) (n2 :variable) (n3 :variable)
                  (pos-sensor-node :variable) (vel-sensor-node :variable))
```

```

(r1 :rotational) (r2 :rotational))
:components ((p-control (mod-control-pd-6 pos-sensor-node n1))
             (d-control (mod-control-pd-6 vel-sensor-node n2))
             (control (mod-adder n1 n2 n3))
             (piston (controlled-piston-3 n3 r1 r2))
             (door (IFV-door-slab-3 r1 r2))
             (pos-sensor (sensor pos-sensor-node door position (Closed Open)))
             (vel-sensor (sensor vel-sensor-node door velocity (Q)))))

:initial ((== mode opening)
          (== (mode piston) normal)
          (== (mode vel-sensor) Q)
          (qIn (>> theta piston) (Q PistonParallel))
          (qIn (>> position door) (Closed Open))
          (q= (>> w piston) Q+))

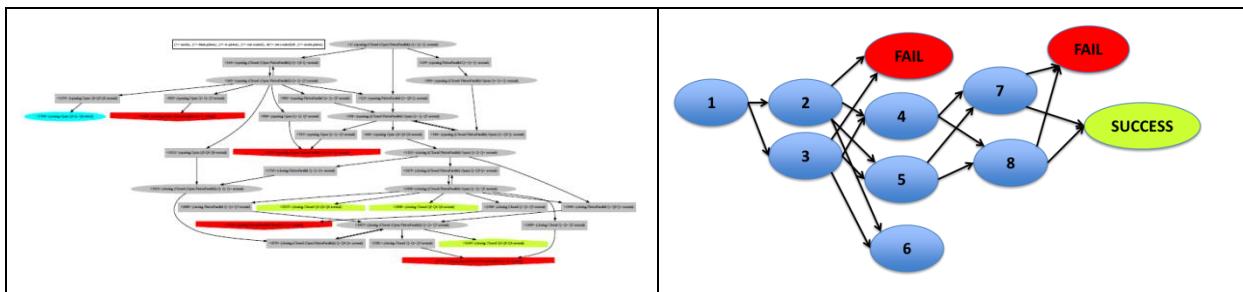
:mode (opening :entry ())
       :initial ((q= (>> out control) Q+)
                  (q= (>> out p-control) Q+)
                  (== (mode pos-sensor) Open))
       :equations ())

:mode (closing :entry ((q= (>> velocity door) Q0)
                       (q= (>> position door) Open0)))
       :initial ((== (mode pos-sensor) Closed)
                  (q= (>> out control) Q-)
                  (q= (>> out p-control) Q-))
       :equations ()))

```

**Figure 7.3-32. Composition of components defining a system in which a door is given a command to open and close.**

The use case for this simulation states that the door should open and close without overshooting and slamming into the ground on open or the vehicle on close. The qualitative simulation is performed by specifying success and failure conditions representing the requirements. A failure situation is one in which the following logical expression holds ( $q=(>> \text{position door}) \text{ Closed-} \vee q=(>> \text{position door}) \text{ Open+}$ ), i.e., the position of the door is either before the closed landmark or after the open landmark. A situation is determined to be a success if the system is in the closing mode and the door is stationary at the closed position, ( $== \text{mode closing} \wedge q=(>> \text{position door}) \text{ Closed0} \wedge q=(>> \text{velocity door}) \text{ Q0}$ ). We are in the process of extending the specification to include other expressions, like those used in linear temporal logic.



**Figure 7.3-33. (left)** Directed graph describing the envisionment of the door system included for concreteness. **(right)** A simplified version of the envisionment graph to describe the example.

The envisionment, shown in Figure 7.3-33, is created by taking the door system and initial situation specified above, and computing each possible successor situation. The envisionment on the left represents all possible trajectories through the qualitative state space. To illustrate the important features of this envisionment, we use the simplified graph with labeled nodes on the right. Starting in the initial situation, 1, the system will transition to one of two intervals, 2 and 3, as the door progresses toward open. The next set of situations, 4, 5, 6, and FAIL, represents the state at which the door reaches the open position. FAIL is colored as red because the position of the door is past the open landmark.

Each of situations 4, 5, and 6 represents an interval after the mode has been changed from opening to closing. Situations 4 and 5 transition into situations 7 and 8, and finally, the simulation ends with two more terminal situations, SUCCESS and FAIL. The success situation satisfies the condition that the door is in the closed position and stationary; and the fail situation is where the door has overshot the closed landmark.

Analysis of this envisionment provides the following feedback to the designer. First, the design may reach a successful situation. Second, we can assess how difficult it will be verify a design quantitatively. In this case, each of the requirements may be violated. Therefore, quantitative analysis will be needed for each requirement. Alternatively, if one is looking for a quantitative estimate for how difficult it will be to verify the design, one could use the ratio of successful states to terminal states, in this case 1/3.

There is a terminal situation, 6, that does not satisfy the success or failure conditions of the system. This dead-end state implies the need for additional requirements to guide the designer. In this case, this situation results from a kinematic singularity in the piston door connection. That is, when the acting angle of the piston is parallel to the angle of the door, the piston produces no torque. While this is part of the piston component model, it is only terminal if the door is stationary at this point. To identify this risk required simulating the system through the use case where the door first opened and then closed. With only individual use cases, one for opening the door, and one for closing it, the envisionment would not show this dead-end state.

#### 7.3.4.5 Envisionment and PCC Computation

In view of how central PCC is to the META program, we did some testing with OSU's Uncertainty Propagation code, to see how a numeric PCC might mesh with a Qualitative Reasoning approach. The two were not well matched.

QR's strength is to rapidly deal with unknowns and approximations early in design space exploration, before part selection and sizing. To run specific stochastic inputs through a compiled Modelica model and do Taylor series uncertainty propagation on the results, the designer must nail down the parts and their tolerances. A great many repeated runs of the model are necessary to compute a PCC. The calculation in Matlab was approximately twenty minutes.

The requirement we evaluated was ability to control ramp operation within the specified ten-second interval, using a progressively lighter and less powerful actuator. In the envisionment graph there is a transition from the "opening" interval to successfully open, and another transition to a failure state in which the ramp is not at a stationary open position within ten seconds. We tried to see if PCC calculations would help a designer to better understand the probabilities of taking those transitions.

Steady state torque needed by the ramp\_A03 model to hold the ramp open was 7342 N m, which was close to the maximum torque needed during soldier egress. Sizing the motor for a nominal 7406 N m leads to output being saturated for 750ms of the opening sequence, so the design is right on the edge and has a PCC slightly above zero. A small increase in nominal motor size to 7440 Nm reduces the saturation interval to one sixth of a second. If actual max torque forms a beta distribution in the range 7350 to 7463 Nm, then PCC for successful ramp operation is 98.8%, with larger motors giving even greater assurance of proper operation.

But in answering the larger question of whether this is an appropriate analytic tool to understand the envisionment graph or to size the motor, the answer turned out to be "no". Simpler analyses of necessary torque margin, or of saturated output duration, are far more productive methods for sizing components. Compute intensive repeated stochastic model runs using specific component parameters should be performed a little later in the design process, to verify that appropriate margins were used and that they perform in combination as expected.

#### **7.3.4.6 Modelica to QML Translation**

Modelica is a fully featured modeling and simulation language designed to do numeric simulations from models of cyber-physical systems. It provides an object-style inheritance for object structure, and specifies behavior using algebraic and ordinary differential equations. It also provides computational capabilities found in standard programming languages such as C. QML is a language that provides facilities for describing the qualitative behavior of models, and a simulation engine that computes an envisionment of possible alternative behaviors of a composite model based on the composition of behaviors of its components. Component behavior has only qualitative values (basically significant ranges of values) and uses qualitative linkages between variables, and qualitative analogs of ordinary differential equations.

For a significant class of constructs, there is a straight-forward mapping from Modelica models to QRL models. We have built a translator that will do this mapping so that we can take advantage of the large standard library available from, and be able to do qualitative simulations of models that were built in Modelica. We have started with the Modelica Analog Basic library. At this point we can successfully translate about three quarters of the models in that library. For the rest, there are constructs that are not appropriate to translate – for example, descriptions that are not model based (support equation based behavior), but instead make use of algorithm and function constructs.

We have developed a methodology for exploration of the translation process that enables us to see where there are issues in our translation. Our strategy for dealing with gaps has four prongs: adding capabilities to the translator; extending the Modelica language to include constructs needed for qualitative modeling (e.g. explicit modes); rewriting Modelica models to in the extended language; and declaring (parts of) some models that need to be hand translated.

The following pair of models indicates how a basic capacitor in Modelica is translated automatically into QML.

Capacitor	
Modelica	QML
<pre>model Capacitor "Ideal electrical capacitor"   extends Interfaces.OnePort;   parameter SI.Capacitance C(start=1); equation   i = C * der(v); end Capacitor;</pre>	<pre>defprototype Capacitor :external-terminals   ((p :electrical)    (n :electrical)) :variables   ((v (voltage p n))    (i (current p))) :equations   ((q= i (deriv v 1 t))))</pre>

**Figure 7.3-34. Modelica and QML Models of a Capacitor**

There are a number of points that should be noted in Figure 7.3-34 above. Modelica provides a general inheritance mechanism from previously defined classes. In this case Capacitor inherits from the class *OnePort*. This inheritance (from *OnePort*) provides two external interface connection points, **p** and **n**; it also specifies that associated with these connection points are electrical parameters of Voltage and Current. In the translation, we recognize this particularly extends construct, and specially create the electrical external terminals, and define the voltage and current parameters locally as they relate to those terminals. Note the translation of the equation, where *q=* specifies a qualitative equality, and *(derive v 1 t)* specifies the first (1) derivative of *v* with respect to time. Note that the numeric parameter *C* (Capacitance) has been dropped since it does not affect the qualitative equation.

The model below illustrates how Modelica synthesizes models from components, and how we translate this to QML. The RLC circuit has 4 components. These components are serially connected, from voltage source to resistor to inductor to capacitor and back to voltage source. In Modelica, a connection is explicitly declared with a “connect” statement using the internal names of the connection points of the connected model. It describes the connection between two terminals of (mostly) different components.

QML’s connection description is different. QML reifies connecting nodes, and creates components with explicit links to these nodes. The Modelica model shows the positive pin of the voltage source connected to the resistor’s positive pin. In QML, the connection point itself is an internal-node, and each component is connected to it. In the QML translation, we see voltage source component *V1* is instantiated connecting *V1p* and *V1n* terminals, while resistor

component  $R1$  is instantiated connecting  $V1p$  and  $R1n$ . These names have been generated based on the Modelica model.

	Modelica
<pre>model RLC   Modelica.Electrical.Analog.Sources.RampVoltage V1 (V=5,duration=10.0e-6);   Modelica.Electrical.Analog.Basic.Resistor R1 (R=3.0);   Modelica.Electrical.Analog.Basic.Capacitor C1 (C=10.0e-6, v(start=-5));   Modelica.Electrical.Analog.Basic.Inductor L1 (L=1.0e-3); equation   connect(V1.n, C1.n);   connect(V1.p, R1.p);   connect(R1.n, L1.p);   connect(L1.n, C1.p); end RLC;</pre>	QML
<pre>(defprototype RLC :internal-nodes   ((V1n :electrical)    (V1p :electrical)    (R1n :electrical)    (L1n :electrical)) :components   ((V1 (voltage-source V1p V1n))    (R1 (ideal-resistor V1p R1n))    (C1 (ideal-capacitor L1n V1n)))    (L1 (ideal-inductor R1n L1n)))))</pre>	

**Figure 7.3-35. Translating RLC Model in Modelica To QML**

**Status:** We focused on the automatic translation of Modelica models in the Modelica.Electrical.Analog library. We correctly translate 75% of those models. Where there are differences in the semantics of the two languages, the translator highlights untranslatable constructs. For example, QML does not support algorithms and functions; this is a principled difference because QML is based on models with “no function in structure” – that is QML

models cannot depend on outside contexts. In addition our translator does not support compile-time construction options. These include declaration clauses with *ifUsed* <Boolean> where the Boolean tells the compiler whether to use the declaration. The translator also does not support iterative constructions and arrays.

As part of our process for bringing quantitative and qualitative modeling together, we have been working with the Modelica group to develop some common constructs; the focus has been on extending Modelica to incorporate QML mode definition.

#### 7.3.4.7 Qualitative Simulation Semantics

A qualitative simulation of a device results in a direct graph of the qualitative states that device may exhibit starting with specified initial conditions – the envisionment. A quantitative simulation of any assignment of parameters to the device results in a single trajectory that can be mapped to one of the qualitative trajectories in the envisionment. Therefore there is a homomorphism from the set of all trajectories of quantitatively specified models and the qualitative trajectories computed for the envisionment, as illustrated in the figure. Thus the semantics of the envisionment is a projection of the semantics of the quantitative modeling system.

#### 7.3.5 Bibliography

[AB11] Benveniste, A. et al. (2011) “Contracts for the Design of Embedded Systems Part II: Theory.” Submitted for publication.

[BCW84] Williams, B. C. (1984), *The Use of Continuity in a Qualitative Physics*, AAAI.

# META Adaptive, Reflective, Robust Workflow (ARRoW)

## Phase 1b Final Report

### TR-2742

## Appendix 7.4 – Library Requirements

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.4     Library Requirements.....</b>	<b>1</b>
7.4.1   CML Library Goals and Capabilities .....	1
7.4.1.1   Existing Libraries.....	1
7.4.1.2   Synthesis of Library Capabilities.....	7
7.4.1.3   Access to Information at Different Levels of Abstraction.....	8
7.4.1.4   Beyond Specification .....	8
7.4.2   CML Goal Enablers.....	9
7.4.2.1   Content is King .....	9
7.4.2.2   Reasoning .....	10
7.4.2.3   Data-Driven Access.....	10
7.4.2.4   Maintenance .....	10
7.4.3   Library Strategies.....	11
7.4.3.1   Developing Ontologies.....	11
7.4.3.2   Creating Content .....	12
7.4.3.3   Organizing for Reuse.....	14
7.4.3.4   CML Administration.....	18

## List of Figures

Figure 7.4-1. Toolkit Library.....	1
Figure 7.4-2. Parts and Datasheet Library .....	2
Figure 7.4-3. Repository library - <a href="http://www.cpan.org/">http://www.cpan.org/</a> .....	3
Figure 7.4-4. Developmental library - <a href="http://sourceforge.net/">http://sourceforge.net/</a> .....	3
Figure 7.4-5. Information library - <a href="http://dbpedia.org">http://dbpedia.org</a> .....	4
Figure 7.4-6. Library of Libraries .....	5
Figure 7.4-7. Product Data Management Library - <a href="http://www.ptc.com/solutions/windchill-10/">http://www.ptc.com/solutions/windchill-10/</a> .....	5
Figure 7.4-8. Library Management Automation .....	6
Figure 7.4-9. Library Mirrors .....	7
Figure 7.4-10. Process of incorporating components from a CML and of sharing candidate components to the library. ....	8
Figure 7.4-11. Component Hierarchy Schema.....	11
Figure 7.4-12. Ramp Component Hierarchy Schema.....	12
Figure 7.4-13. Component Reuse.....	14
Figure 7.4-14. Component Copy and Modify.....	15
Figure 7.4-15. Component Alias.....	16
Figure 7.4-16. Component Inheritance.....	17

## List of Tables

Table 7.4-1. CML technology enablers concentrate on aspects of the library architecture.....	9
Table 7.4-2. Critical Product Development Functions .....	13

## List of Symbols, Abbreviations, and Acronyms for Appendix

Symbol, Abbreviation, Acronym	Definition
BOM	Bill of Materials
CAD	Computer Aided Design
CML	Component Model Library
COTS	Commercial Off The Shelf
ITAR	International Traffic in Arms Regulations
LCM	Life Cycle Management
SHARE	Software Hardware Asset Reuse Enterprise
TRL	Technical Readiness Level
UML	Unified Modeling Language

## 7.4 Library Requirements

Model-based systems provide a means of explicitly storing technological knowledge while sharing and communicating it between adjacent and parallel work threads. This facilitates horizontal integration whereby knowledge becomes accessible independently of time, location, and people. This being the overall goal needed with a Component Model Library (CML) and the virtual product instantiation of the CML, the Master Model.

### 7.4.1 CML Library Goals and Capabilities

The inherent preconceived notions of what a library should do and its ultimate capabilities must be clearly delineated for a successful CML instantiation. Libraries should facilitate the following capabilities.

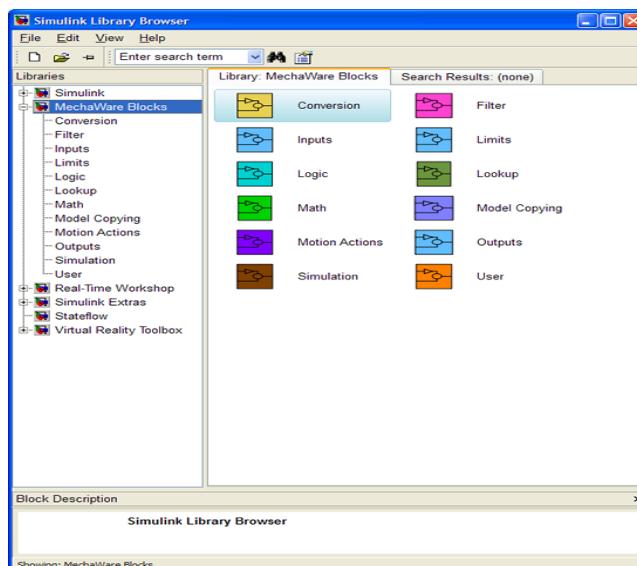
#### 7.4.1.1 Existing Libraries

##### 7.4.1.1.1 Hardware Libraries

Typical libraries should provide access to the large repository of hardware products available today. Industrial supply companies like [McMaster-Carr](#), MSC Industrial Direct Co., Inc., Reid Supply Company, GSA Advantage, Digi-Key, and Glenair provide hundreds of thousands of components for use in everyday products and processes. Linkages and access to this information is critical to aligning the large vehicle design to the smallest of component. The various levels of abstraction of each component will provide the necessary information at all levels of the design process.

##### 7.4.1.1.2 Simulation Convenience

Libraries should make model development and construction convenient. Toolkit libraries come with modeling and simulation applications that aid in the rapid development of models. For example, Matlab/Simulink comes with simulation components that the user can plug into a data-flow diagram via a drag-and-drop mechanism. CAD tools such as Pro/E have similar toolkit library capabilities for commonly used parts. Figure 7.4-1 is an example of a Simulink library interface.



**Figure 7.4-1. Toolkit Library**

#### 7.4.1.1.3 Accurate Behavioral Specification

Libraries should accurately portray the functional and behavioral characteristics of the components. Commercial parts libraries contain datasheets and occasionally working models. For example, electronic component libraries may contain models written in SPICE to specify behavioral operation. Figure 7.4-2 highlights the various types of information available.

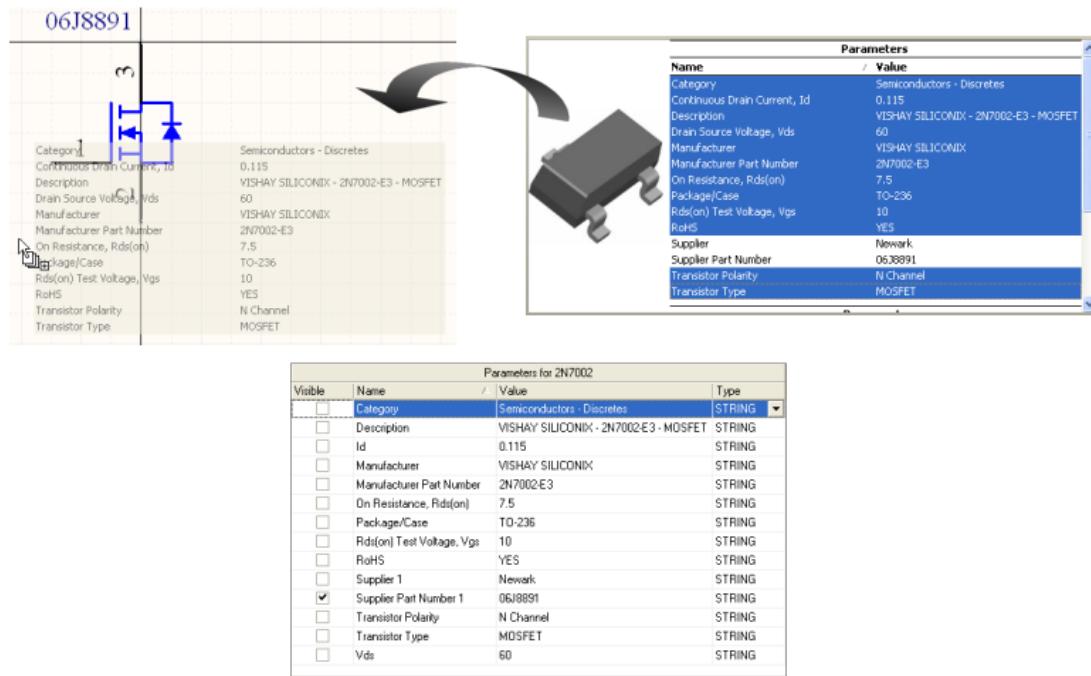
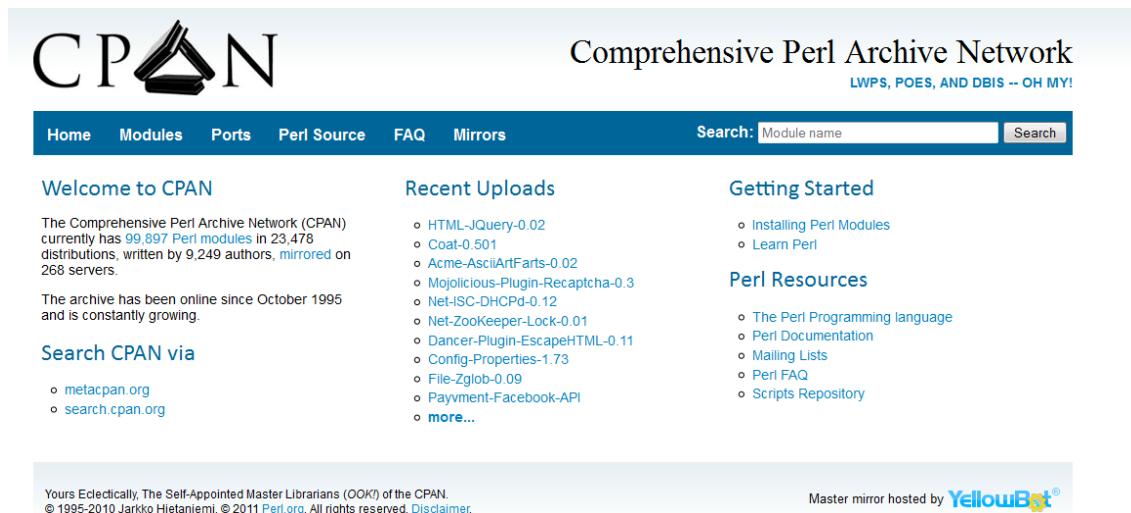


Figure 7.4-2. Parts and Datasheet Library

#### 7.4.1.1.4 Reuse and Sharing

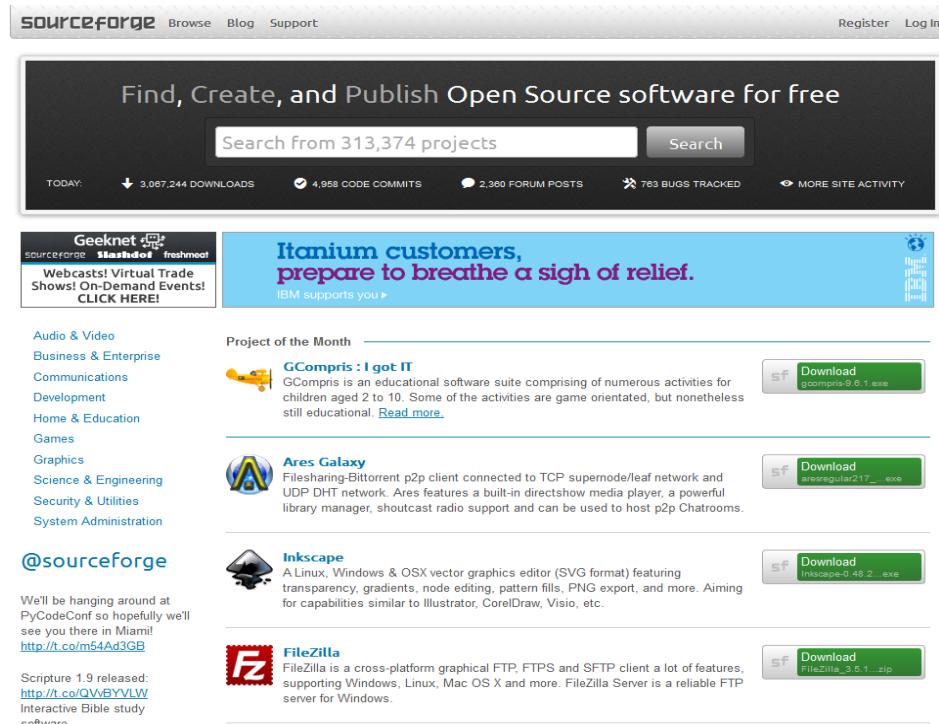
A library should not only establish storage of content, but should also promote utilization of content. A repository library is associated mainly with software. The repositories may be language specific or they could provide translations of algorithms to a number of different languages, such as the Numerical Recipes library (in Fortran, Pascal, C, C++). These can be hosted on a file system or a database. Figure 7.4-3 highlights the importance of an online library such as CPAN.org.



**Figure 7.4-3.** Repository library - <http://www.cpan.org/>

#### 7.4.1.1.5 Flexibility and Integration

A library should not only facilitate the final storage of an article, but should also promote the development of content. A developmental library holds components that are under continuous development, potentially for a Master Model. SourceForge or GitHub are examples of distributed repositories for hosting source code. Figure 7.4-4 highlights the SourceForge example.



**Figure 7.4-4.** Developmental library - <http://sourceforge.net/>

#### 7.4.1.1.6 Knowledge Source

A library should contain data and present knowledge by providing organization such that searches can be guided or directed more efficiently. An information library can contain organized data corresponding to semantic content. This is the basis for the ontologically organized *semantic web*. The information site DBpedia is a semantic version of the more free-form content of *Wikipedia*. Knowledge sources typically contain reasoners which form the basis of a library front-end user interface. Figure 7.4-5 highlights the DBpedia example.

The screenshot shows the DBpedia search interface. At the top, there is a logo for DBpedia with the text "search powered by neofonie". Below the logo is a search bar with the placeholder "enter search terms..." and a "Search" button. To the right of the search bar are links for "About DBpedia", "About neofonie", and "Contact". Below the search bar, there are three filter panels on the left:

- item type**: A dropdown menu with "start typing..." and options "Organisation (107)", "University (107)", and "Educational Institution (107)". A "more" link is at the bottom.
- was established in year**: A dropdown menu with "1966" selected, showing "1966 (107)" as an option. A "more" link is at the bottom.
- country**: A dropdown menu with "start typing..." and options "United States (41)", "Canada (6)", and "India (5)". A "more" link is at the bottom.

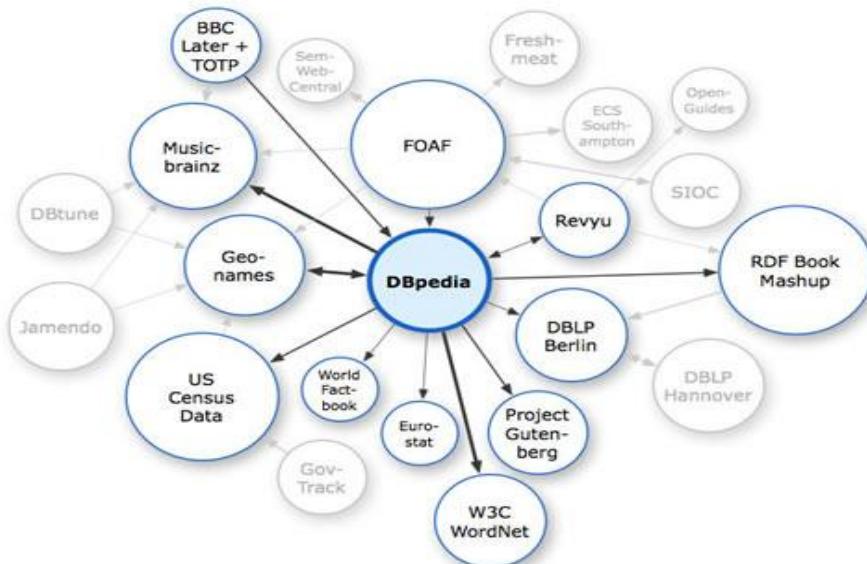
In the center, under "Your Filters", it shows "item type University" and "was established in year 1966". To the right, it says "Results 1 to 6 of 107". Below this, there are two search results:

- University of Calgary**: A brief description of the University of Calgary, mentioning it is a research-intensive public university in Calgary, Alberta, Canada, with 24,000 undergraduate and 5,500 graduate students. It notes its separation from the University of Alberta and its founding in 1966.
- University of Bath**: A brief description of the University of Bath, mentioning it is a campus university located in Bath, England, receiving its Royal Charter in 1966. It notes its ranking in the UK and its placement in the Top Ten of Their Subjects from the Complete University Guide published by the Independent in April 2009, and its placement in the Guardian University Guide 2010.

Figure 7.4-5. Information library - <http://dbpedia.org>

#### 7.4.1.1.7 Uniformity of Reference Method

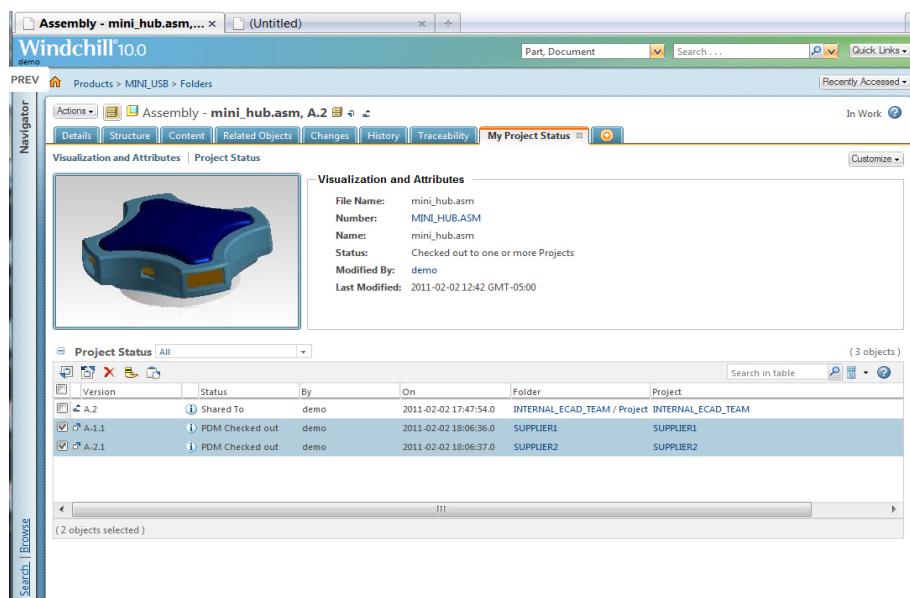
A library of libraries is needed to cross-index between available libraries. Instead of a Component Model Library, we may need to specify a Component Model Warehouse, which has a much bigger scope. The fact is that libraries will need to interoperate, just as traditional brick-and-mortar libraries interoperate with other libraries with shared card catalogs and inter-library loaners. The libraries need to have fundamental uniformity in reference to support inter-operation. Figure 7.4-6 illustrates an example of multiple library interaction.



**Figure 7.4-6. Library of Libraries**

#### 7.4.1.1.8 Life-Cycle Management and Curation.

The library needs to support an LCM process and curation to distinguish the developmental components from the mature and utilized ones. Product Data Management (PDM) libraries offered up by vendors such as Windchill can hold enterprise models alongside other information. The data tracked usually incorporates the technical specifications of the product, specifications for manufacture and development, and the types of materials that will be required. The PDM serves as a central knowledge repository for process and product history, and promotes integration and data exchange among all required functions. Figure 7.4-7 highlights a Windchill example.



**Figure 7.4-7. Product Data Management Library - <http://www.ptc.com/solutions/windchill-10/>**

#### 7.4.1.1.9 Dependency Management and Automation

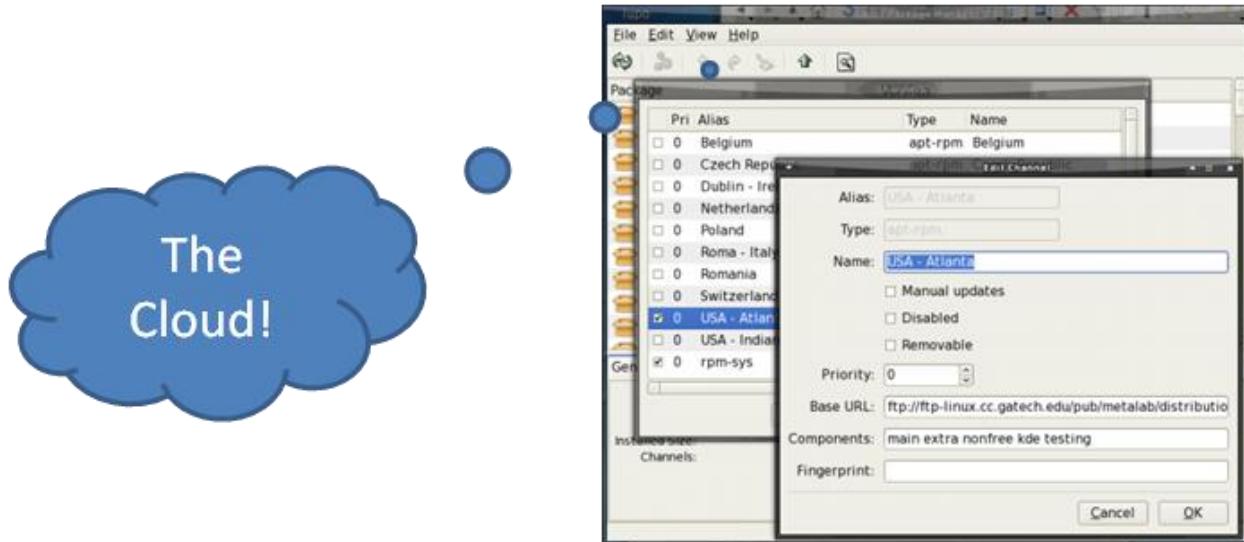
The library needs to support dependency management with the interacting master models to establish out-of-date, up-to-date, and modified components status within the master models. To accommodate the huge number of library runtime components that a software application may consist of, a library manager such as Artifactory which maintains control of versions, etc is necessary. Specific version control, configuration management, build management, and retrieval mechanisms are available for automation, such as GIT, Subversion, RPM, and Maven. Maven is a mechanism for dependency tracking and management. Figure 7.4-8 highlights the Artifactory example.



**Figure 7.4-8. Library Management Automation**

#### 7.4.1.1.10 Dependability and Availability

Libraries are required to be continuously available. As an example, online libraries have an expectation to be dependable mechanisms with permanent availability and fixed reference method. Mirror libraries which contain duplicated archives of libraries, or the recent approach known as cloud computing make availability less of a concern. This also provides a mechanism for protecting Intellectual Property and Classified information by being able to separate the data more effectively. Figure 7.4-9 highlights the Cloud example.



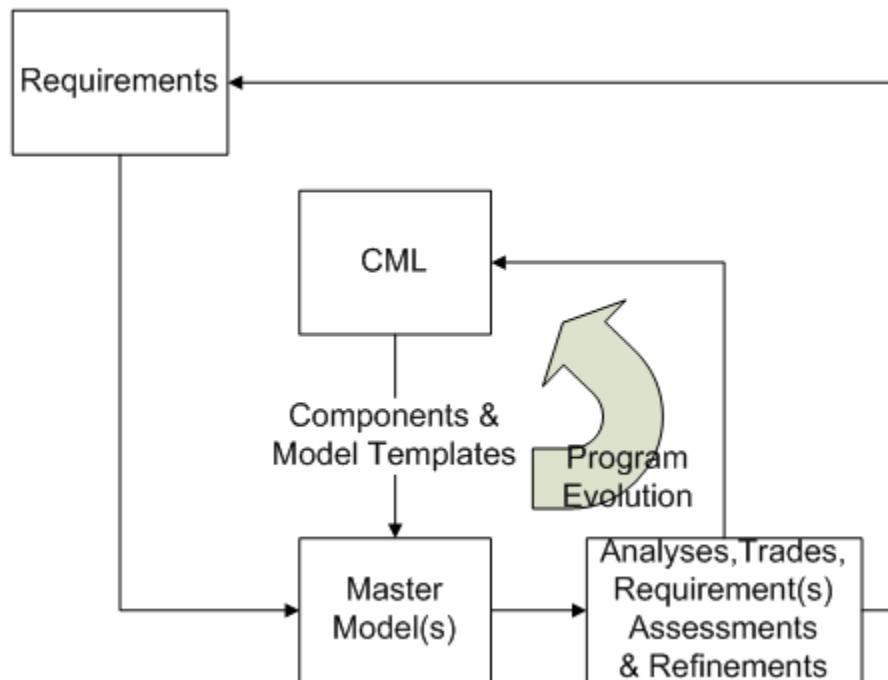
**Figure 7.4-9. Library Mirrors**

#### 7.4.1.2 Synthesis of Library Capabilities

Library interaction would extend from the typical parsing of singular requests to inferring the query trend from the strings of interaction. Ideally, a desirable library capability would capture interaction history and anticipate future needs. This would approach the goal of a tool such as Wolfram Alpha, which essentially responds polymorphically to a user's request for information. Thus it can provide context models if the query is one of asking for "weather data in Chicago from 1950 to 1960" or it will provide a numerical algorithm in symbolic form if the knowledgebase can infer meaning from a math library request.

Barring being able to accomplish that laudable goal, as an interim solution we have an objective to providing as many capabilities as necessary to support the integration of a master model from a component model library.

By building from requirements and an initial population of available components, we can establish a process whereby proven components from the master model can be fed back into the component model library for later reuse. Refer to Figure 7.4-10 for an example of the process flow. Test cases of proven components will also be captured to effectively save time when establishing new assemblies. This will enable the concept of crowd sourcing to utilize archetypes to easily find and develop products off of similar solutions. This will also encourage large organizations to benefit from the crowd-sourced advances.



**Figure 7.4-10. Process of incorporating components from a CML and of sharing candidate components to the library.**

#### 7.4.1.3 Access to Information at Different Levels of Abstraction

The key to effective reuse from the existing CML lies in being able to reason directly from requirements. This is understood when we consider that the highest level of abstraction in a CML element is a component specification. As all components must have some minimal form of specification, this information can be used for unique and unambiguous identification of a component. Thus we can potentially transform requirements into essential information which we can then apply to a component matching process. In other words, we can map requirements onto candidate component specifications.

The authoring of a well-grounded specification will often be extracted from lower level (higher fidelity) abstraction models. For instance, a component's mass may be listed in its specification, and that mass may be derived from a solid model of the component. The specification can also contain information that is not associated with any underlying model. This sort of information can include data describing component maturity, vendor information, government classification, lead time, previous use, and general pedigree.

#### 7.4.1.4 Beyond Specification

The CML will also connect any number of models associated with this component specification or aspects of this specification. For example, a bolt may be specified as 1/4-20 hex head cap screw. This bolt component could then also have a 3D CAD representation, stress analyses and thermal properties associated with it. One essential low level of abstraction for the bolt would be the technical drawing that enables a manufacturer to create a bolt that has all the same characteristics as its models. A software component may be specified via its inputs, outputs, intended usage, and functional definition. Models of the software may include UML or

Simulink representations, and at the lowest level the source code used to generate the actual software library or executable.<sup>1</sup>

#### 7.4.2 CML Goal Enablers

The classical approach to library access and retrieval is to apply the concepts of ***classification***, ***information content***, and ***repository***. This architecture enables a divide-and-conquer strategy which will allow us to organize and access our content efficiently. The idea of ontological classification is to provide uniform, predictable, and consistent long-term organizational structure. The informational content provides an indexing scheme that allows reasoners to operate with specific information while being more variable and flexible than the ontological classification alone. The repository holds all the information of the library including source code, models, and associated artifacts.

Organization of the library in this way allows us to adopt certain tools and algorithms that work well in each of the abstraction level areas as well as across the levels. We broke the enabling mechanisms into these categories (Table 7.4-1): reasoning, access, content, and maintenance. While the preceding enablers represent the life cycle of information in the CML, content is ultimately the primary reason of the CML.

**Table 7.4-1. CML technology enablers concentrate on aspects of the library architecture**

<b><u>Enablers</u></b>	<i>Ontological</i>	<i>Informational</i>	<i>Repository</i>
Reasoning	X	X	
Access	X	X	X
Content		X	X
Maintenance		X	X

##### 7.4.2.1 Content is King

The component model library will contain a collection of existing components and models, archetypes, historical component usage, and a large variety of other information. These repository items are indexed appropriately for fit with the CML flexible structure. This will become the overall knowledge base, with ontologically-classified, efficiently indexed knowledge acting as a powerful goal enabler for component discovery and reuse. In other words, the content is not contained in the ontology but in instances and assertions classified by that ontology.

The content can contain associated meta-data attributes (related to attributes of design elements in the master model). Methods for fast authoring of information, both in defining and populating the content knowledge are needed. For example, adding new artifacts to the

<sup>11</sup> The development of good specifications for software is difficult, the Software Hardware Asset Reuse Enterprise (SHARE) Repository Framework Final Report: Component Specification and Ontology lays out some interesting concepts that could be of assistance in this area.

repository by inheriting aspects of archetypal artifacts is a potentially useful mechanism for populating a library quickly.

#### **7.4.2.2 Reasoning**

At the most fundamental level, access to the CML can be constructed based on some ontologically classified component data types with semantic views that can be filtered by any set of constraints or rules. All access to the CML should have automation hooks available.

The discovery of component models, collections of views of a component, or archetypes is helped by reasoners optimized for searching the CML knowledge base. A search compatible mechanism such as a “map reduce operation” together with a CML structured through attributes and relationships, has tremendous advantages over traditional directories and hierarchical topologies by increasing efficiency and effectiveness of the search capability. This type of action specifically indicates that the reasoners work at the indexing level and not at the repository level to preserve the integrity of the actual component model information.

#### **7.4.2.3 Data-Driven Access**

Applying mechanisms that allow uniform, fast, and unambiguous access to information is necessary across the board. Only the information levels of abstraction should change. The development process must not be hindered by the amount of time it takes to access data from the CML. A good efficient interface with reasoning behind the scenes keeps developers engaged in useful work. However, it was determined that open source tools don’t necessarily eliminate the need for standard file formats. This indicates that it is imperative to use some form of uniform development methodologies while maintaining as much of the original models as possible.

Specifically, for the repository, most models are stored in native format, but other representations will become necessary for efficient reuse of components. The combination of segmentation of the master model and CML implies that it will be possible to make component models available for incorporation into designs and to test designs incorporating these models, without making the models themselves public.

The language section discusses the content implementation language, which obviously derives from the ontological scheme chosen.

#### **7.4.2.4 Maintenance**

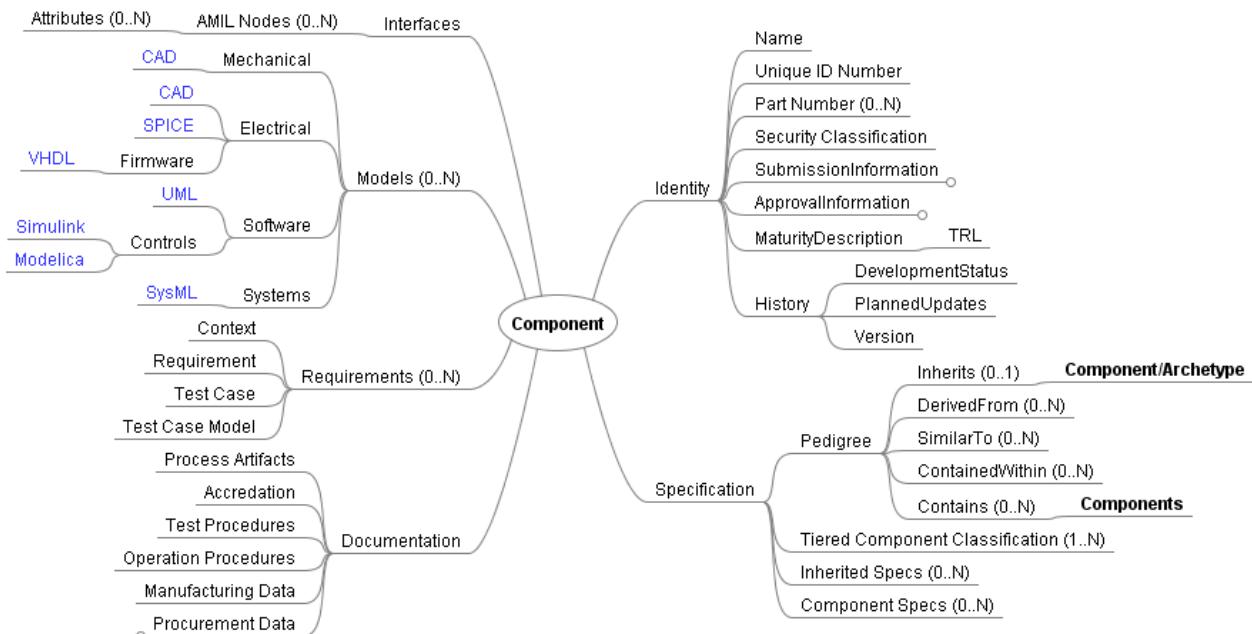
Maintenance will be required over the life cycle of the data within the repository and on the informational structure to the data. The data within the CML will evolve with active development in many instances and go unchanged in some cases. References to the CML from Master Models can become broken links without necessary maintenance. Enabling the lifecycle management and curation within the CML allows the connections between models and the CML to be continued. The invariant characteristics of the ontological classification scheme establish a solid reference for the instance knowledge categorized in the CML.

The mechanism of curation allows systematic design change, i.e replacing or refining a particular design element.

### 7.4.3 Library Strategies

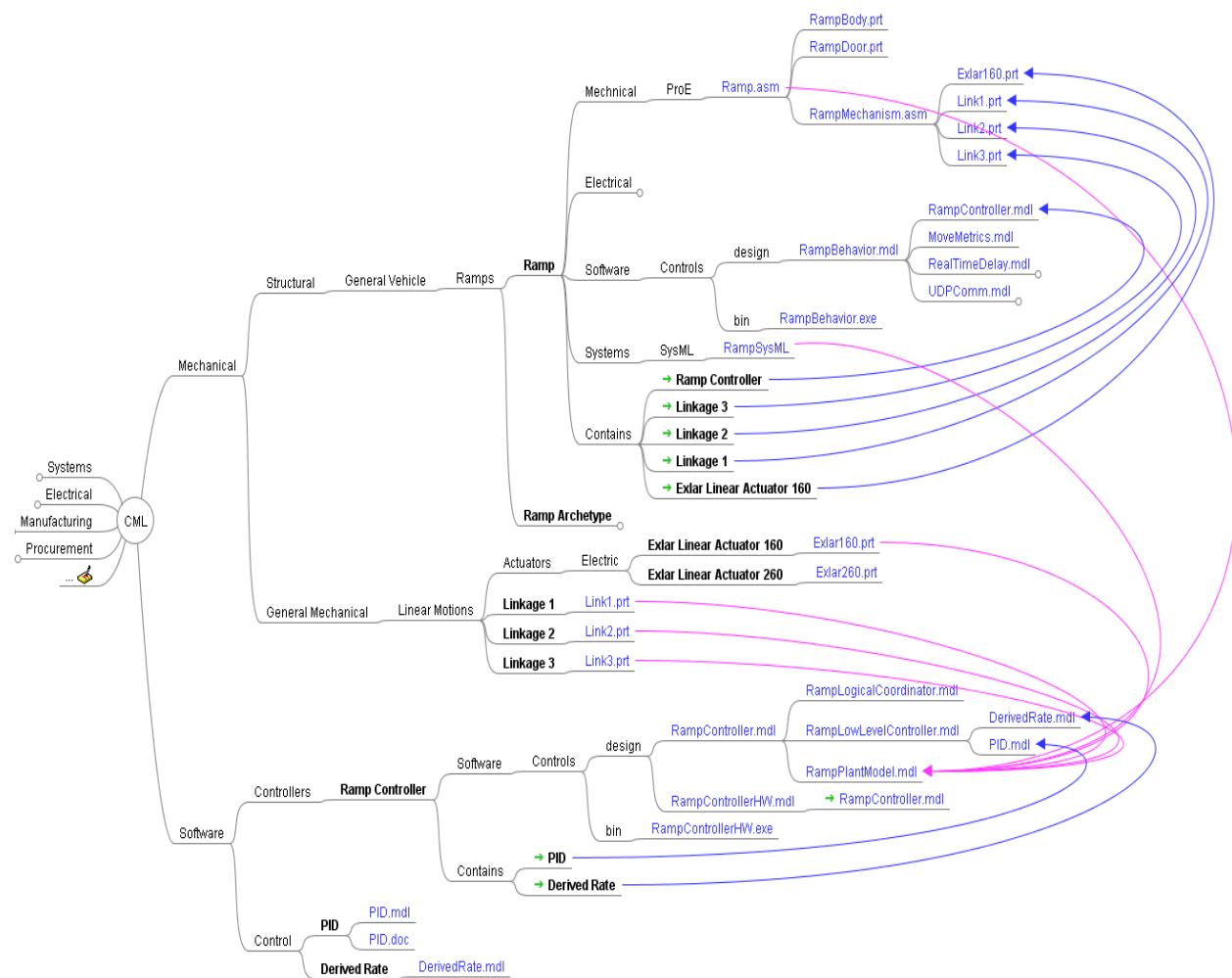
#### 7.4.3.1 Developing Ontologies

Ontology development requires large collection of information to best represent and capture the necessary pieces of a component and its associated models, artifacts, and layers of abstraction. Figure 7.4-11 represents a high level view of required fields of a component. The classification provides the fundamental structure, but the information can change as needed with the CML growth and use.



**Figure 7.4-11. Component Hierarchy Schema**

Figure 7.4-12 represents an instance of a ramp in the CML. The overall ontology structure remains intact, but the appropriate and available information is accessible along with the proper connections that link the data for the various engineering needs.



**Figure 7.4-12. Ramp Component Hierarchy Schema**

### 7.4.3.2 Creating Content

The capture of knowledge is key to establishing the necessary models and artifacts sufficient to represent a component. Historically, many of models used to represent components, subsystems, and systems are productized and built on demand by the appropriate experienced experts. However, the models that are built for a demand can be extended to serve across all product development domains. Table 7.4-2 captures the critical engineering functions that will need to be served with the appropriate models, artifacts, and data stored in the CML. The list following the table is a brief list the types of data that can be necessary in a CML. While not an overly exhaustive list, it captures some of the breadth of information needed.

**Table 7.4-2. Critical Product Development Functions**

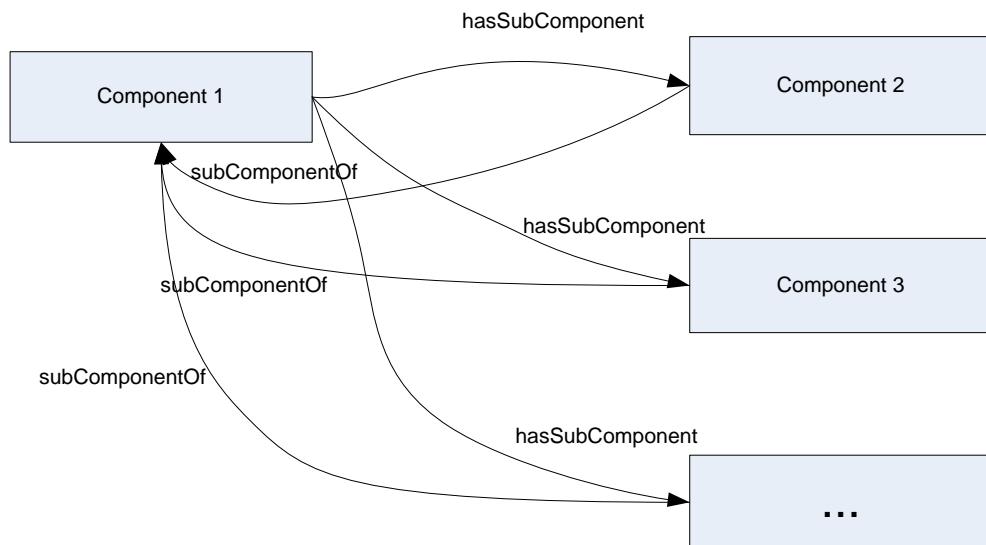
Engineering Functions Provide CML Content		
Controls	Human Factors	Training
Schedule	Operational Effectiveness	Production
Safety	Survivability	Structural
Finance	System Integration	Software
Reliability	Maintainability	Thermal
System Engineering	Design	Test

## Example Types of CML Data

- COTS items
- Fully developed released Pro-E Models
- Space claim representation for Vendor Controlled Drawings/Altered Vendor Controlled Drawings
- Sizing models (excel – spring design, o-ring; weld size, etc)
- Component selection templates (motors/actuators; gearboxes; fasteners; dowels; retainers; inserts, etc)
- Materials database
- Drawing standard notes
- Heat Treat Notes
- Coating Notes
- Problem Records / Lesson learned
- Name of the tools for results that are in the library
- Manufacturing processes, cost factors, producibility limitations
- Supply chain material availability, cost, lead-times
- Access to rough space claim CAD data based on functional inputs
- Ready access to existing components for use in design (fasteners, pins, handles, motors, etc.). Search capabilities across the industry.
- Generic scripts for automation
- Possibly acceptable posture ranges for human analysis
- Common Use Case Model
- As Designed, As Planned and As Supported BOMs

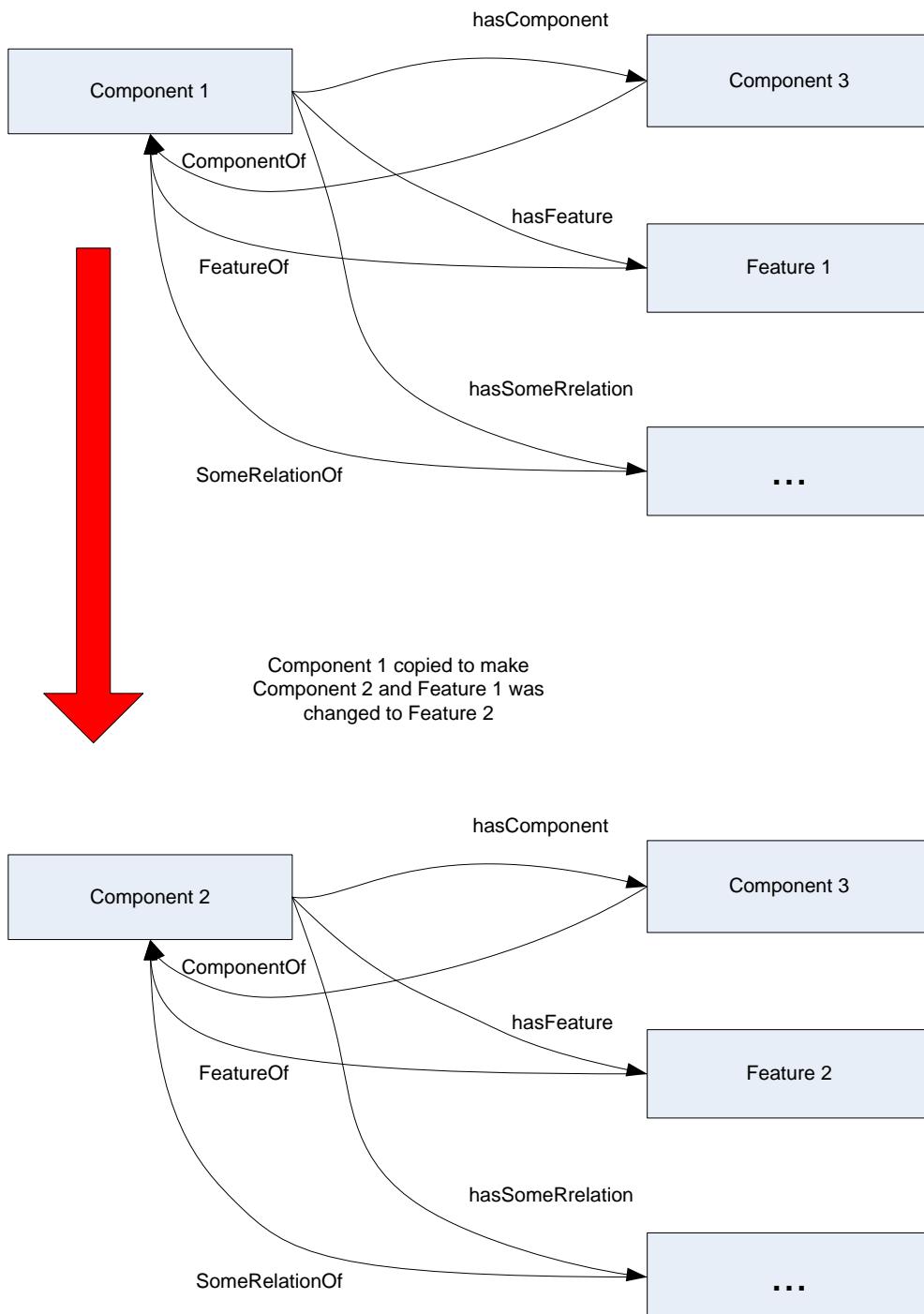
### 7.4.3.3 Organizing for Reuse

The CML will rely heavily on the ability of one component instance to reuse the effort of another component individual in order to realize the  $5\times$  improvement needed for META. There are four typical ways that this reuse can be accomplished: containment, copy, alias, and inheritance. The majority of the reuse will be via containment of other component. In the proposed ontology example, Figure 7.4-13, the object properties of hasComponent and its inverse ComponentOf would be the mechanism for creating a containment relationship from one component to another.



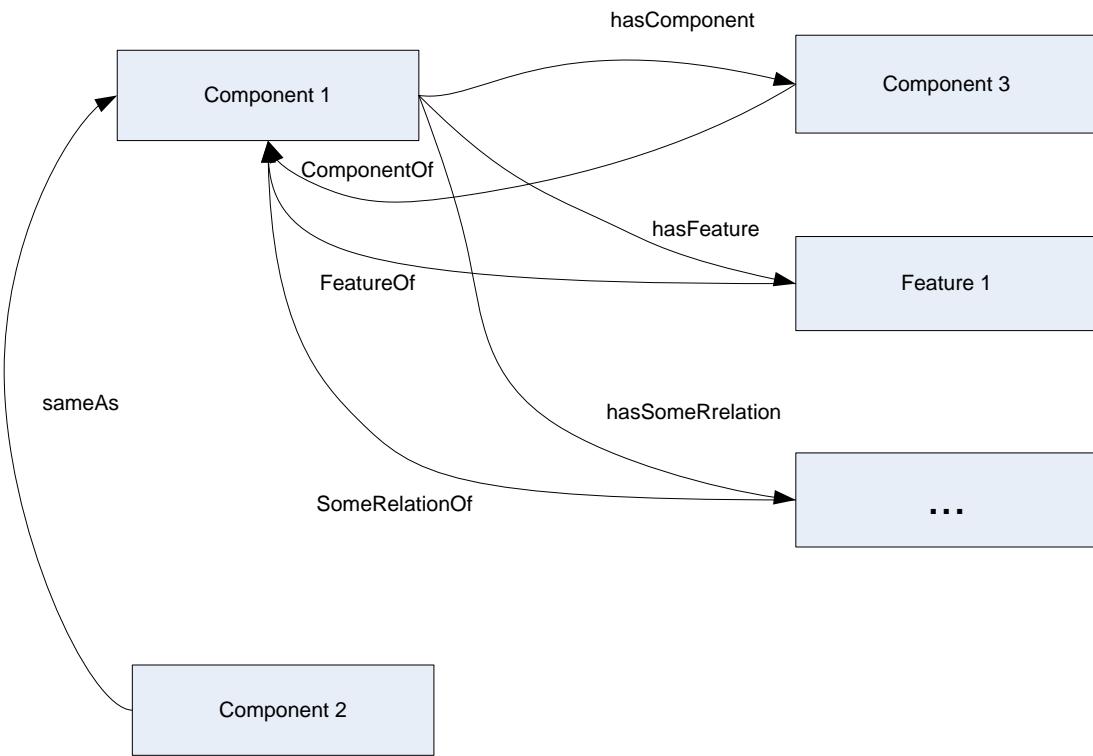
**Figure 7.4-13. Component Reuse**

The second mechanism is a copy of the component. This does not maintain a relation from one component individual to another component individual. This allows a component developer the freedom to be able to change the internal structure of the component but has the disadvantage that the developer will not get changes if the original is modified. Copying and modifying a component is shown in figure 7.4-14.



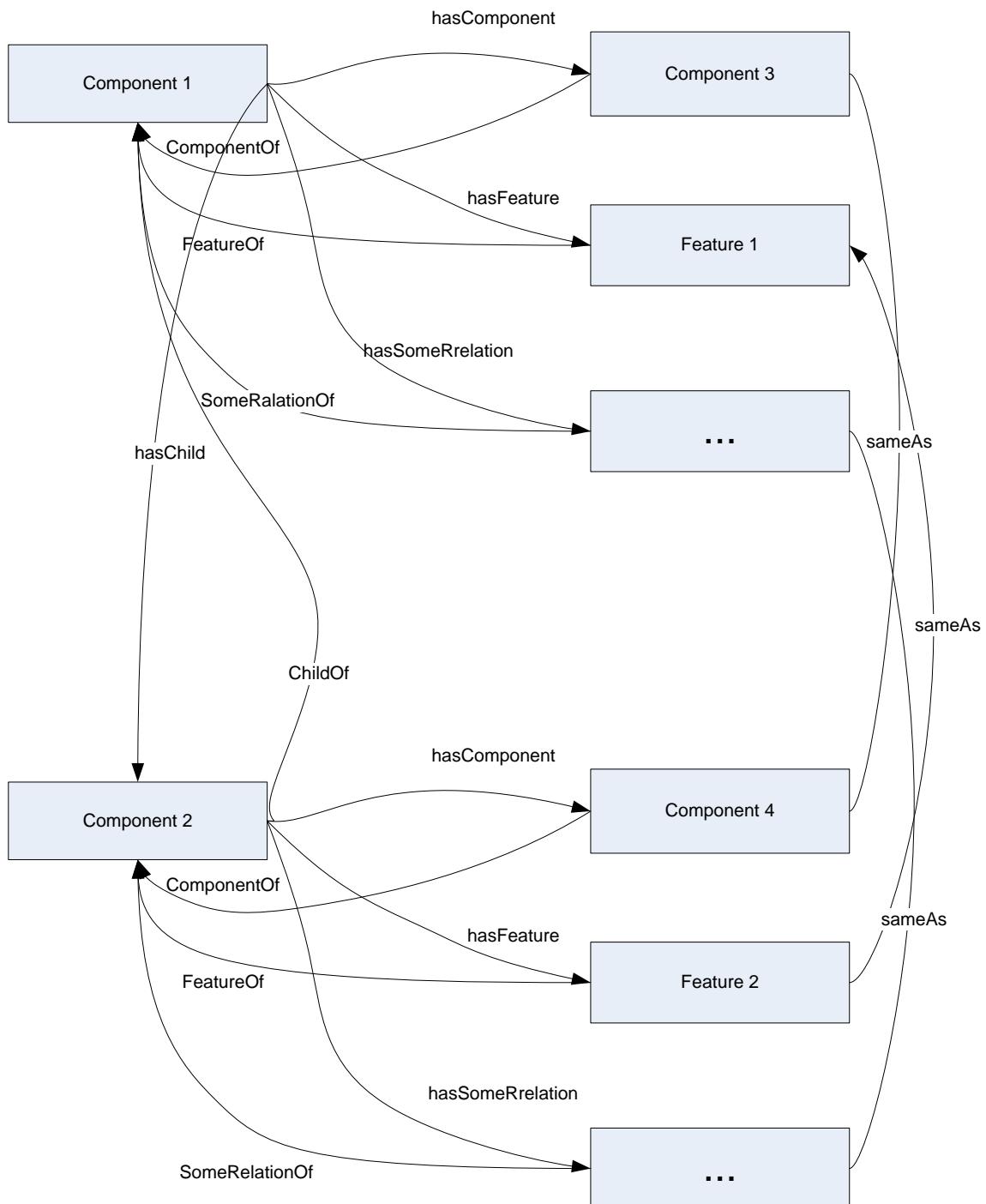
**Figure 7.4-14. Component Copy and Modify**

The third type of reuse is to alias one component individual from another component individual using a syntax such as sameAs. This type of reuse will not allow the component developer to change any of the features or properties from the base component individual. If a change is made the result is an inconsistency in the data and there will be two paths that yield different results. An example of this type of reuse is found in the figure 7.4-15.



**Figure 7.4-15. Component Alias**

The last form of reuse is the inheritance of one component from another. Inheritance of one component individual from another will again need to be done via an object properties defined in the Arrow ontology. All the individuals that the parent has a relation would be aliased so that the child could change these aliased individuals and override the values in the parent. The hasChild and ChildOf properties would maintain the tie between parent and child so that changes in the parent would be reflected in the child. These changes could be done manually but a manual process would be prone to error and lack consistency. The complexity of these relations will necessitate that a tool be used to maintain the component individuals. An example of this relationship is in following figure 7.4-16.

**Figure 7.4-16. Component Inheritance**

#### **7.4.3.4 CML Administration**

##### **7.4.3.4.1 CML Tools**

The difficult task of creating and maintaining the CML will require a robust set of tools. Some of the needs for CML tools are addressed with a combination of tools that are currently available but there is a need to integrate and augment the capabilities to provide a robust solution. These tools will need to enable the curator and users of the CML to create, change and remove components while maintaining the integrity of the library. The curation of the CML will require that a tool set will provide version control, consistency management, dependency tracking, change notification, and maintain the structure of the library. The component creators and users will require client tools that ease the task of component creation, component selection, interface definition, and component maintenance.

Version control is not a new concept so there are many available tools, but the CML does have some unique challenges that will require some augmentation of the existing tools. Because of the high degree of interdependence between specific versions of components there will be additional features needed to manage these dependencies and to manage upgrading components. When there are multiple dependent components, conflicts may arise when any component is upgraded. Warning the user of conflicts and managing these complex dependency trees is something that will need to be addressed by the CML tools.

Another aspect of the high degree of interdependency is maintaining the consistency of the components. If the components were administered without the aid of some tools it would be easy for components to create circular dependencies. Having tools that maintain the dependency tree and alert the user when a circular dependency occurs would prove to be invaluable during component design.

Ease of use for searching for and using components is also a factor that will need to be addressed in order for these tools to be accepted in general use. If this is not the case the 5x goal will likely not be achievable via the use of components. The users will need to be able to find the components and have a high degree of confidence that they are useable and verifiable. Tools do not currently exist that can provide this kind of specific information about components in the CML.

##### **7.4.3.4.2 CML Population**

Component owners will populate the component library in order to encourage the use of their components. Oversight will be required to ensure components are accurately represented, documented, and controlled since component owners may have a conflict of interest in making their components appear as attractive as possible. Component owners will be responsible for entering appropriate information for component models including low fidelity information through high fidelity meta information. The government may also be the owner of some components making ownership more complicated. Ultimately, there needs to be moderation of the library, a mechanism for component owners to populate the model library, and a process for system integrators to solicit the desire for new components.

##### **7.4.3.4.3 CML Integrity**

Integrity of the library will be controlled by the Library Owner/Moderator who will provide the appropriate level of access and control of the library. Some indicators of model usefulness could include an incremental TRL-like metric as the indication of quality and/or a certification of the model providing traceability and quality control. The Library Owner/Moderator would also have the ability to "whitelist" or "blacklist" products, components, technologies, or vendors

as more information is collected. This approach facilitates the open/crowd-sourced nature of the program, while at the same time providing a filtering mechanism to control the breadth of the component space.

#### 7.4.3.4.4 CML Data Control

Part of the metadata associated with each model must include data classification level. The database handles for the models themselves may point to different databases for varying levels of data protection. Access to these databases can be controlled per the established procedures associated with the data within them. There will likely be many models to represent each component and each of these models may have different classification levels associates with them.

Classified and International Traffic in Arms Regulations (ITAR) considerations: Applied models may contain ITAR-restricted or classified information that may require protection. Theoretical models may be well known processes and not be ITAR restricted, or may be innovative models and will fall into the category of ITAR restricted/Company Proprietary.

# META Adaptive, Reflective, Robust Workflow (ARRoW)

## Phase 1b Final Report

### TR-2742

## Appendix 7.5 – Notional Demo System Application

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.5 Notional Demo System Application.....</b>	1
7.5.1 September Demo Walkthrough.....	1
7.5.2 Application of ARROW to the Communications Domain.....	8
7.5.2.1 Introduction .....	8
7.5.2.2 Link Analysis .....	9
7.5.2.3 ARROW Support for Communications.....	9
7.5.3 Electronic Warfare Example.....	11
7.5.3.1 Introduction .....	11
7.5.3.2 Archetype Summary.....	12
7.5.4 META Challenge Problem.....	13
7.5.4.1 Design Space Challenge Problem: Ramp Assembly .....	14
7.5.4.2 Challenge Problem CFV Performance Specification .....	17
7.5.5 Bibliography .....	24

## List of Figures

Figure 7.5-1. The META ARRoW Architecture combines cloud-compatible services with tailored local apps, including open-source, commercial, and META-specific engineering tools..	2
Figure 7.5-2. The ARRoW Model Interconnection Language (AMIL) Viewer application provides a representation of the underlying connections that the META ARRoW system uses to synchronize and automate the design process.....	3
Figure 7.5-3. The Early Concepting Tool (ECTo) interface provides multiple panels for managing the design, reviewing the requirements, and accessing the component model library.	4
Figure 7.5-4. The Metrics Dashboard interface provides reconfigurable panels and a variety of display formats for managing the designer's view of the current system's performance.....	5
Figure 7.5-5. The ECTo tool provides a 3-D graphical representation using CML models to assist designers in making rough space claim decisions during early design conceiving.....	6
Figure 7.5-6. The ECTo tool's 3-D visualization capability can also incorporate space-limiting requirements components, such as tunnels, to provide visual cues to the designer.....	7
Figure 7.5-7. The Early Concepting Tool (ECTo) interface provides tools for querying the CML against computed system requirements.....	8
Figure 7.5-8. Communications Reference Architecture SysML Model .....	9
Figure 7.5-9. Design View of the Communications Link between NGV and Bradley Vehicles.	10
Figure 7.5-10. The Communications Dashboard .....	11
Figure 7.5-11. EW System Abstraction of Archetypes.....	13

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
AAE	Army Acquisition Executive
AMIL	ARRoW Model Interconnection Language
ANSI	American National Standards Institute
AR	Army
BBIT	Background Built-In Test
BIT	Built-In Test
C <sub>2</sub>	Command and Control
CCW	Counterclockwise
CFV	Combat Fighting Vehicle
CML	Component Model Library
CW	Clockwise
DoD	Department of Defense
DoD	Department of Defense
DODISS	Department of Defense Index of Specification and Standards
ECTo	Early Concepting Tool
EIA	Electronic Industries Association
EW	Electronic Warfare
FIT	Fault Isolation Test
GFE	Government Furnished Equipment
IBIT	Interactive Built-In Test
ICD	Interface Control Document
IFV	Infantry Fighting Vehicle
ISO	International Standards Organization
LRU	Line Item Replaceable Unit
MIL-C-	Military Standard C-
MIL-HDBK-	Military Handbook
MIL-STD-	Military Standard
MMBF	Mean Miles Between Failures
MOPP-IV	Mission Oriented Protective Posture (IV)
MTBF	Mean Time Between Failures
NBIT	Non-Interactive Built-In Test
SA	Situational Awareness
SAE	Society of Automotive Engineers
SBIT	Start-Up Built-In Test

Symbol, Abbreviation, Acronym	Definition
SysML	Systems Modeling Language
TBD	To be determined
TBP	To be provided
TECOM	Test and Evaluation Command
TOP	Test Operating Procedure

## 7.5 Notional Demo System Application

### 7.5.1 September Demo Walkthrough

We provide here a brief walk through of the demonstration elements as shown during the September 2011 META Principal Investigator meeting. The overarching layout of the demonstration system includes three primary locally deployed META applications (a SysML editing tool, the Metrics Dashboard, and the Early Concepting Tool (ECTo)) and a variety of META Adaptive, Reflective, Robust Workflow (ARRoW) services hosted in a cloud computing environment (Figure 7.5-1. (Note: The locally deployed applications are depicted on separate screens, but can all be easily managed from a single workstation.)

Step 1. As our demonstration begins, the customer has pre-configured the design space by loading their requirements into SysML using the ARRoW Design Archetypes. *The requirements archetypes are design patterns that encompass the majority of common combat vehicle requirements. These archetypes provide structure for the key parameters that are traditionally embedded in textual requirements, allowing the ARRoW tool set to reason over the requirements. The requirements structures are linked via the ARRoW Model Interconnection Language (AMIL) to provide built-in traceability and synchronization from the requirements to the library of design archetypes for high-level vehicle system categories (e.g. tracked vs. wheeled, ground vs. amphibious, etc.) and to the test and analysis archetypes for system design evaluation.*

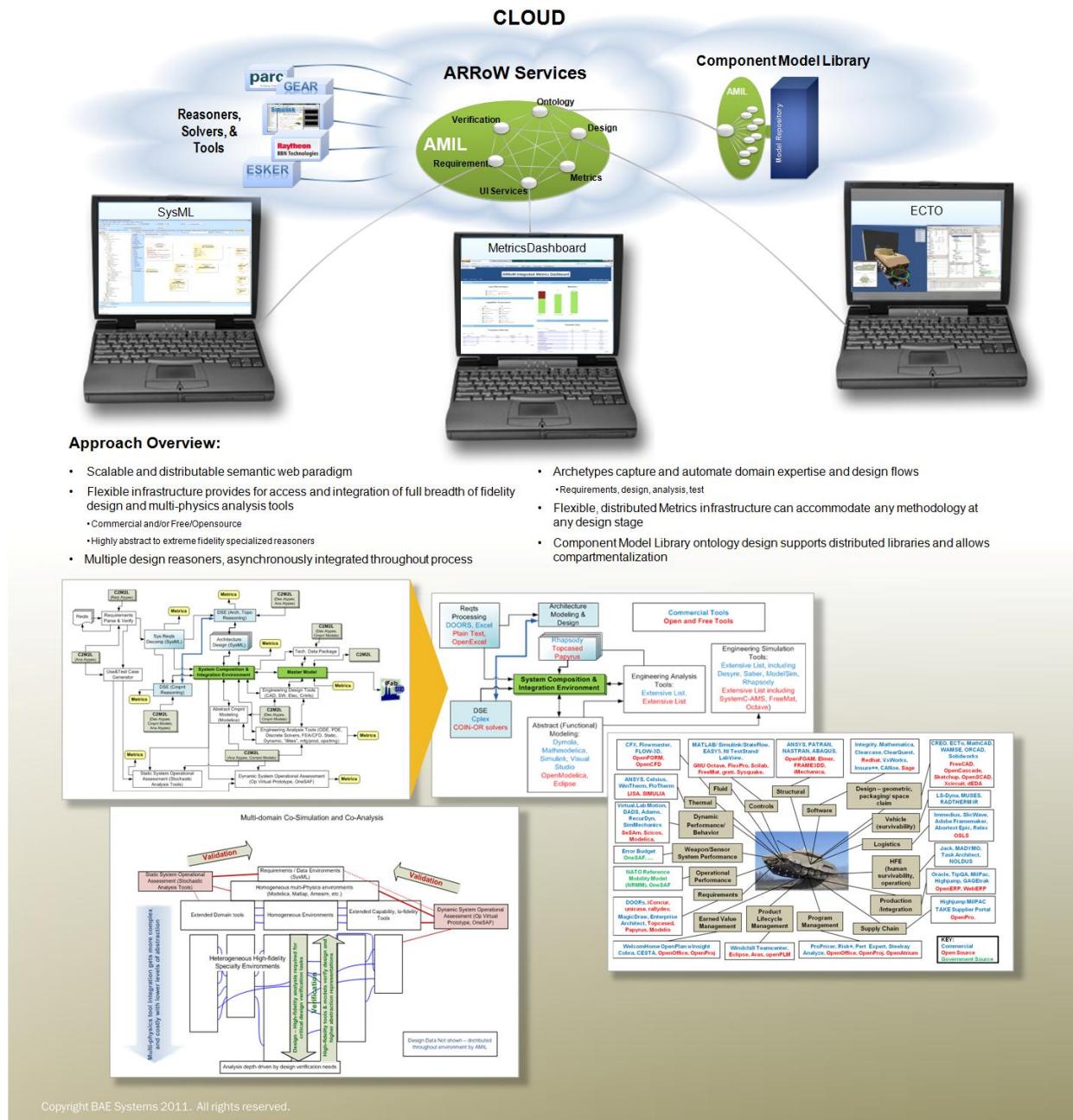
Step 2. The designer reviews a challenge problem statement published on the customer's website:

Design an Infantry Fighting Vehicle (IFV) system such that the total solution is optimized to a set of system-level correctness criteria, and is conformant to the system requirements. The IFV is a versatile medium armored vehicle which provides cross-county mobility dominance, for mounted firepower, communications, and protection to a mounted mechanized infantry squad, overwatch support for a dismounted infantry squad, and deployable anywhere in the world.

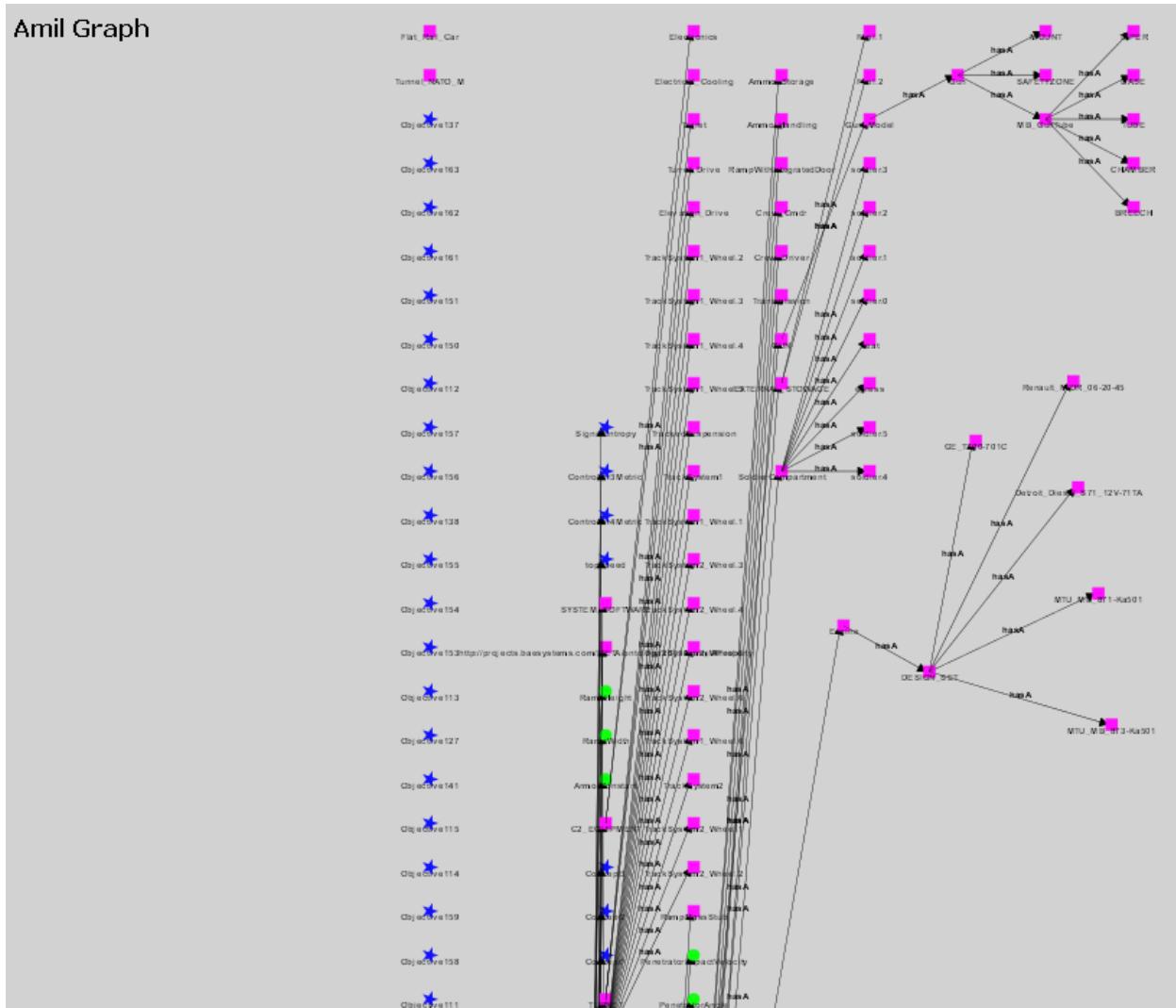
Step 3. The designer manages the requirements in the SysML tool, reviewing and making modifications and refinements to the requirements structures, then publishing the changes. *The designer can use ARRoW functions provided by the SysML tool plug-in (in our case, Magic Draw) to create refinements of existing archetypes and to publish the requirements changes out to the AMIL graph when complete (Figure 7.5-2).*

Step 4. The designer uses reasoners within the SysML tool to assist in the selection of a Infantry Fighting Vehicle (IFV) design archetype. Due to the weight and mobility requirements for this example, the reasoner recommends the Tracked Vehicle archetype. *For designers with more advanced SysML skills, reasoning tools can be integrated at the SysML level to recommend design archetype choices based on automated analysis of the requirements. Our META tool set includes a sample parametric solver implemented in Magic Draw that provides a recommendation for selection of tracked or wheeled archetypes for the IFV.*

**Step 5.** The designer then downloads the META app suite and launches ECTo to begin analyzing the problem through design space exploration (Figure 7.5-3). *ECTo allows the designer to access the customer requirements, review the required system properties, and begin reviewing and customizing the baseline reference designs.*

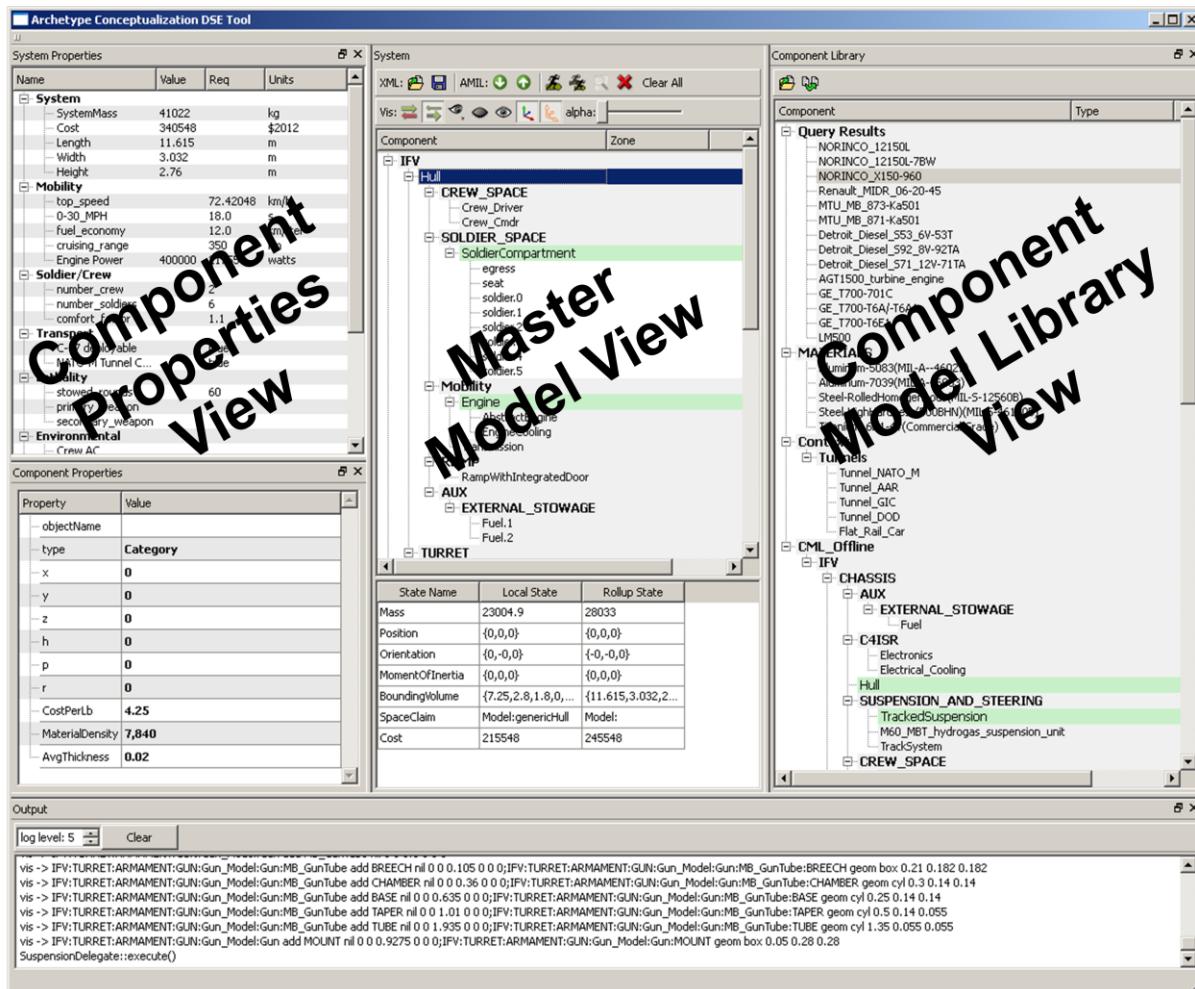


**Figure 7.5-1. The META ARRoW Architecture combines cloud-compatible services with tailored local apps, including open-source, commercial, and META-specific engineering tools.**



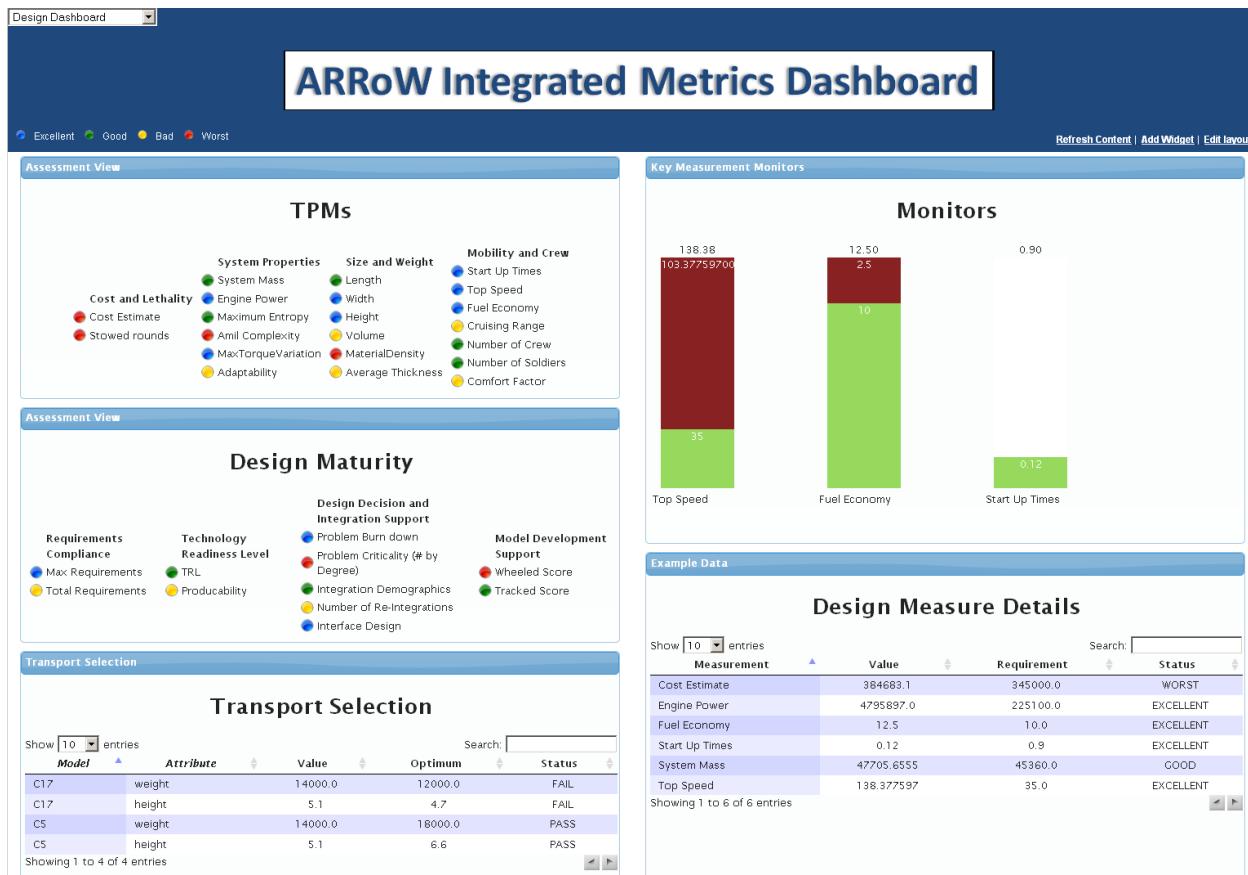
**Figure 7.5-2.** The ARRoW Model Interconnection Language (AMIL) Viewer application provides a representation of the underlying connections that the META ARRoW system uses to synchronize and automate the design process.

- Step 6. The designer searches the Component Model Library (CML) for appropriate system components then drags them into the design. *ECTo provides 3-D visualization for gross sizing and placement analysis, as well as recording the design choices by creating AMIL links between models of design components, including executable models and simulations that can be used to evaluate ranges of design choices. At any point the designer can run all of the models associated with the current design, which updates the design in the cloud.*



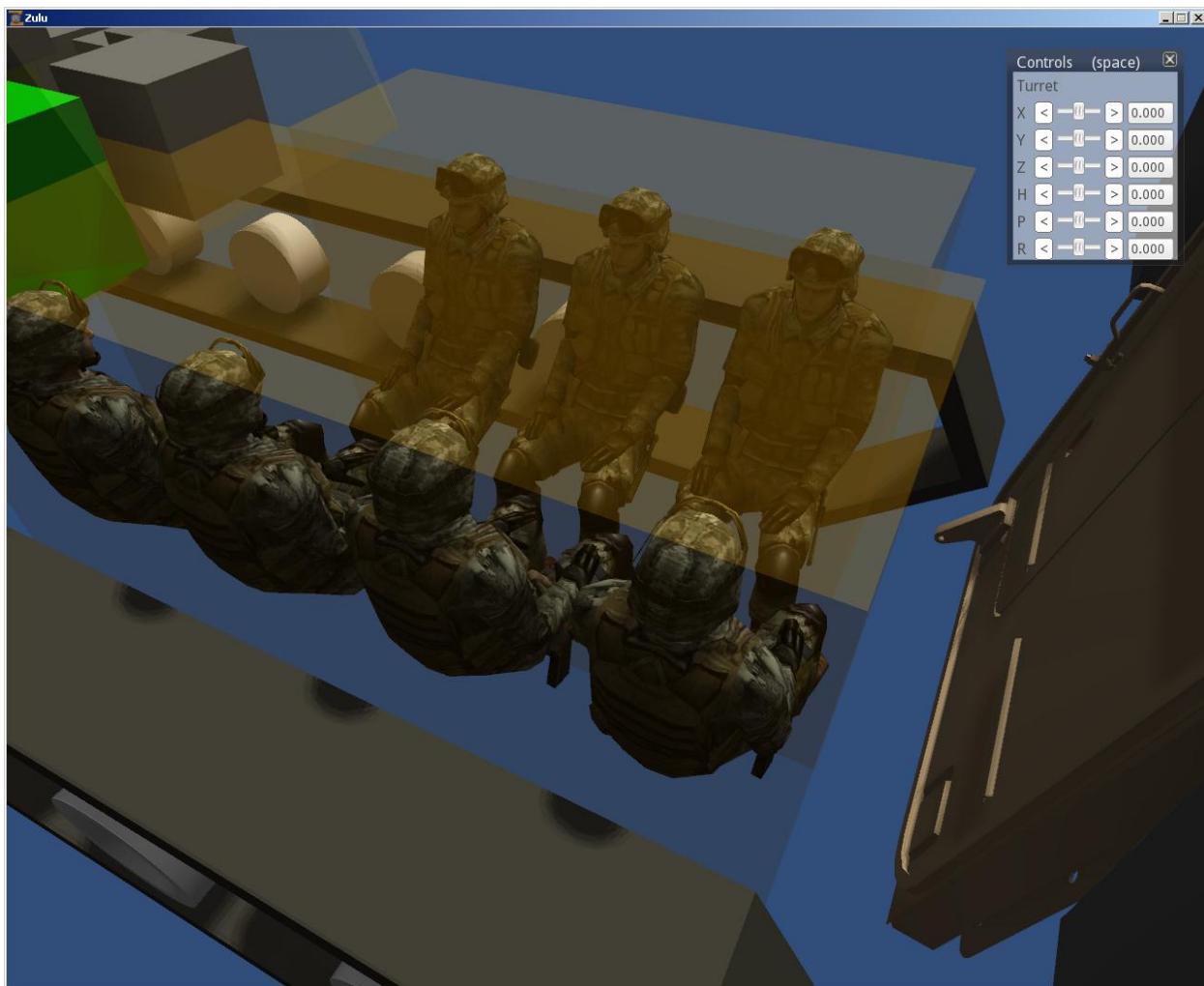
**Figure 7.5-3.** The Early Concepting Tool (ECTo) interface provides multiple panels for managing the design, reviewing the requirements, and accessing the component model library.

Step 7. The designer then launches the metrics dashboard app to review the performance of their current design against the requirements. *The metrics engine is an AMIL service that uses AMIL links between requirements, metrics equations and evaluation tools, and the current system design to evaluate the design and present a dashboard to the designer (Figure 7.5-4).*



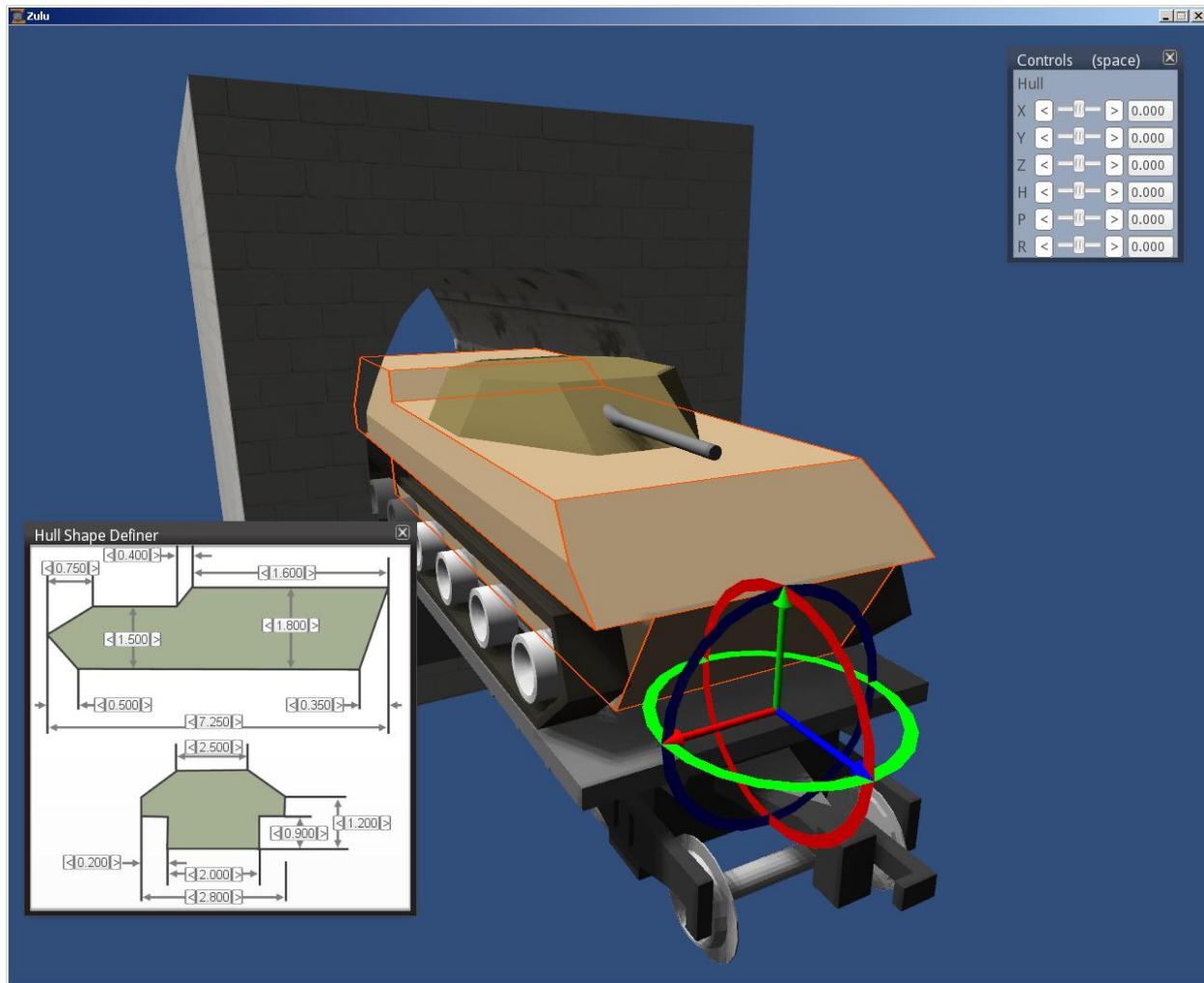
**Figure 7.5-4.** The Metrics Dashboard interface provides reconfigurable panels and a variety of display formats for managing the designer's view of the current system's performance.

- Step 8. As the designer views the metrics, they notice that the crew capacity indicator is amber. A further check reveals that the requirement is for a 9 man squad, but the current design only accommodates 7 men (Figure 7.5-5). *Through the ECTo tool, the designer accesses crew space claim models from the CML, then ECTo performs automated placement and alignment to reflect the number of people the space can accommodate, and displays this in a 3-D simulation view.*



**Figure 7.5-5.** The ECTo tool provides a 3-D graphical representation using CML models to assist designers in making rough space claim decisions during early design concepting.

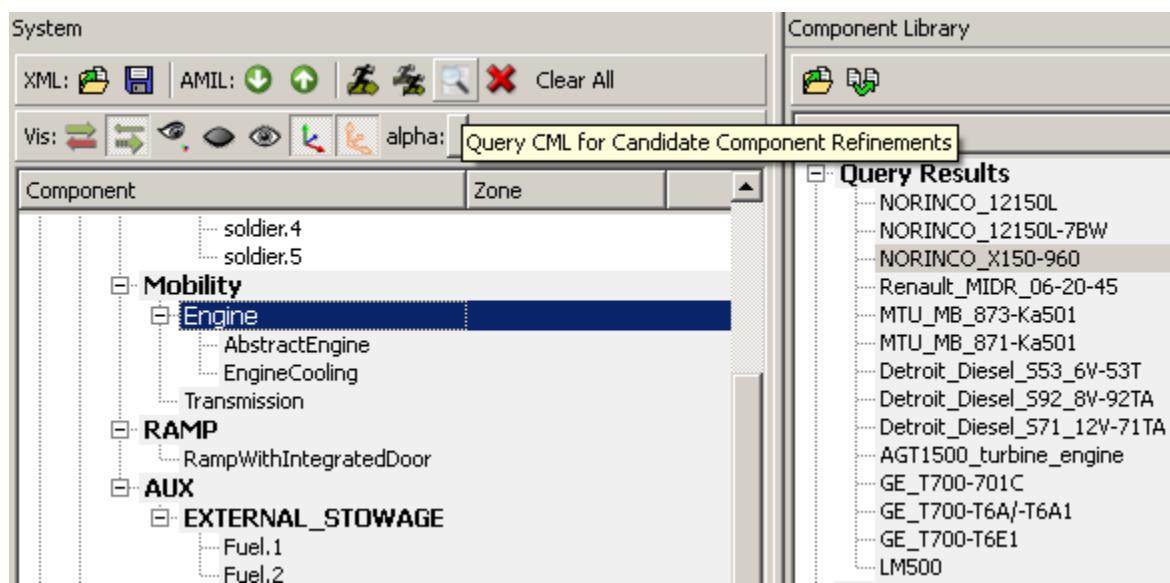
- Step 9. The designer changes the design in ECTo, expanding the hull to allow a crew of 9. Simultaneously, they use NATO tunnel simulation tools in ECTo to keep design within tunnel transportability requirements (Figure 7.5-6).



**Figure 7.5-6.** The ECTo tool’s 3-D visualization capability can also incorporate space-limiting requirements components, such as tunnels, to provide visual cues to the designer.

- Step 10. Another review of the Metrics Dashboard indicates that the new design modifications have caused some of the mobility requirements to fail. A review of the design parameters shows that increasing the hull dimensions has increased the required engine power. *The metrics dashboard automatically computed the required engine power from the estimated weight of the current design and the mobility requirements form SysML such as top speed and acceleration.*
- Step 11. The designer initiates an automated query of the CML database for an engine that meets the required power for the current design. They could then select from the returned list of applicable engines, or engage additional solvers to assist in the downselect process (Figure 7.5-7).

Step 12. A final review of the Metrics Dashboard shows that the current early design concept meets the requirements (at this level of abstraction).



**Figure 7.5-7. The Early Concepting Tool (ECTo) interface provides tools for querying the CML against computed system requirements.**

## 7.5.2 Application of ARROW to the Communications Domain

### 7.5.2.1 Introduction

There are three essential parts of any communications system, the transmitter, RF channel, and receiver. Understanding the characteristics and parameters of each part allows the Systems Engineer to allocate and evaluate requirement compliance against the system design as the analysis and trade studies are continuously refined. The RF link analysis is a key design tool that sets up the initial allocation of the transmitter, RF channel and receiver parameters based on system requirements. The Systems Engineer then has the flexibility of reallocating certain parameters based on trade study and analysis iterations. Refinement of the tradeoffs continues until the system's performance is optimized. Typical challenges that are faced when developing, optimizing, and integrating a communications system onto a platform are communications coverage, interfaces, electromagnetic compatibility, size, weight, power, cooling, and placement of subsystems and components both internally and externally.

Today's communications systems are leveraging Software Defined Radios which are capable of running a variety of waveforms simultaneously. A single vehicle can be required to operate anywhere from 4 – 8 channels of simultaneous communications. The requirement for such a dense electromagnetic environment within a single vehicle envelope drives the overall communication systems design complexity. System requirements such as communications coverage, waveforms, data rate, availability, and size, weight and power (SWaP) initiate tradeoffs and analysis in the area of radio selection, antenna types and sitting, co-channel and co-site mitigation, and subsystem and component placement in order to optimize communication system performance.

### 7.5.2.2 Link Analysis

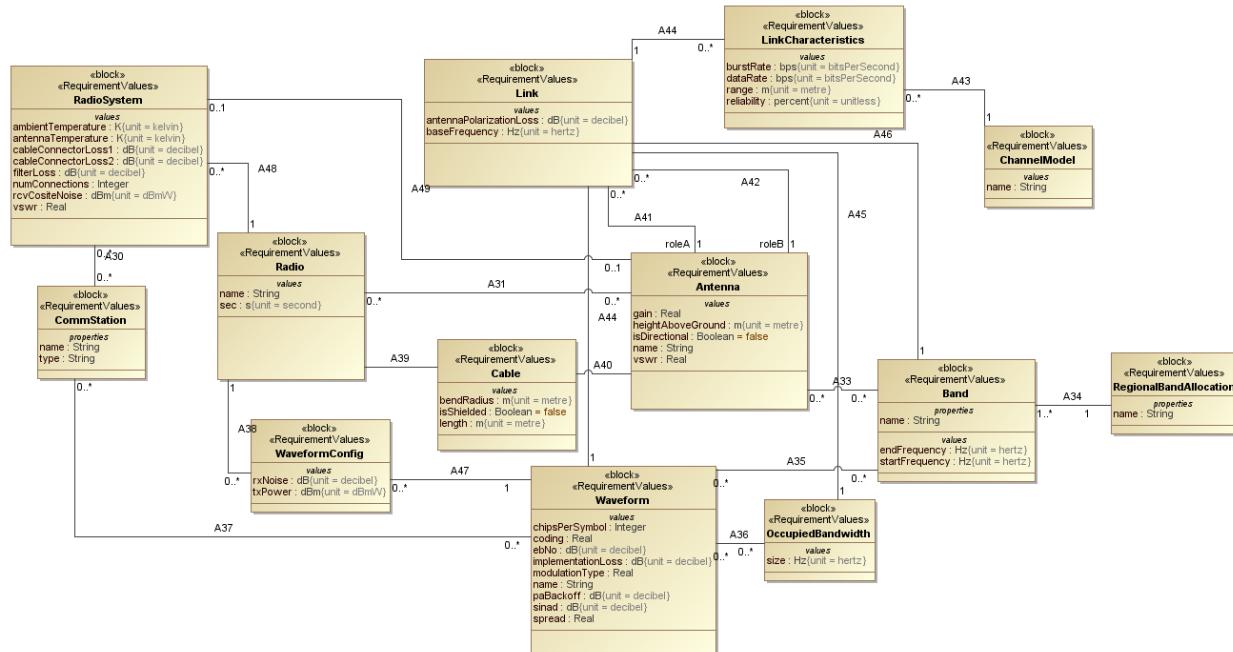
The RF link budget analysis is an accounting of all of the gains and losses from the transmitter through the RF channel to the receiver in a communications system. It accounts for the attenuation of the transmitted signal due to propagation, cables and miscellaneous losses as well as the antenna gains. Randomly varying channel effects such as fading and shadowing are taken into account by calculating some margin depending on the anticipated severity of the channel. The quantity of greatest interest in the RF link analysis is the receiving system's carrier-to-noise ratio (C/N) as modeled at the receiver output. The carrier-to-noise ratio is defined as the ratio of the received modulated carrier signal power C to the received noise power N.

Ultimately the link analysis provides the Systems Engineer with a calculated communications range based on waveform, transmitter, RF channel model and receiver characteristics. The inputs to the link analysis can be varied during the tradeoff and analysis phase in order to optimize the RF link range and communications system performance.

### 7.5.2.3 ARROW Support for Communications

To support the communications domain using ARROW, we first created a reference architecture model in SysML. The reference architecture captures the components, properties, and connections within communications.

The diagram below shows the Communications Reference Architecture. It captures the main components, radios, antennas, and cables, used to construct a radio system. It also captures the characteristics of the Radio System design used to determine the link analysis tradeoffs between range, data rate, power, and availability.



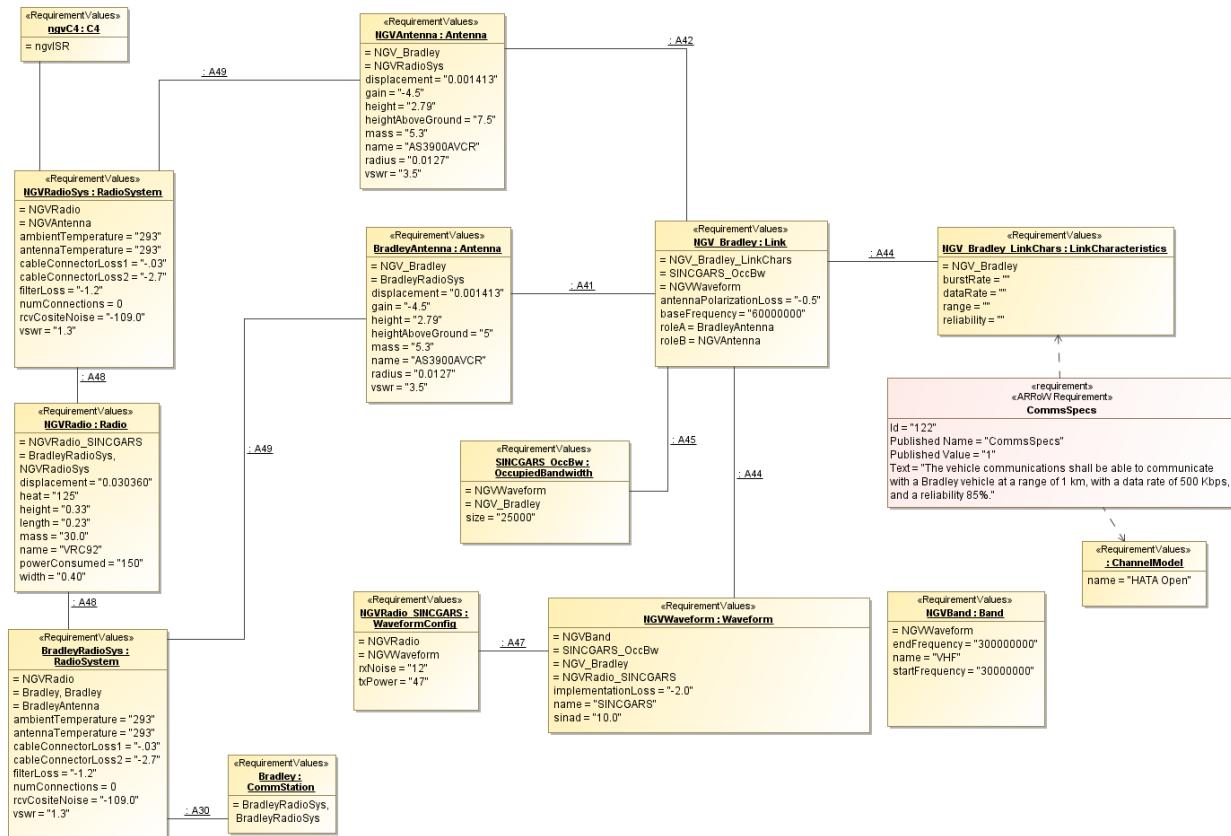
**Figure 7.5-8. Communications Reference Architecture SysML Model**

The Link block represents the RF channel between two antennas, using a base frequency and a waveform to communicate. The link is analyzed using one or more ChannelModels that represent the environment, or context, of the link. Two common link analysis models are

Urban and Open & Rolling, named for the characteristics of the terrain included in their models. The LinkCharacteristics block captures the key properties of the link for each channel model.

The reference architecture is used to create a design of a communications system for a NewGroundVehicle (NGV), shown below. The NGV is required to communicate with a Bradley vehicle, and a specified range, data rate, and availability, captured in the requirement. The design includes the features of the Bradley that affect the comm. link, as well as the NGV communication system design.

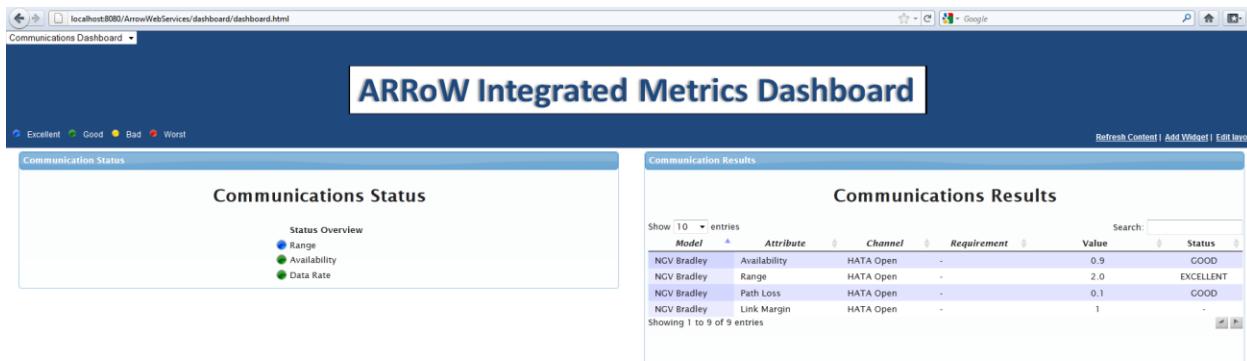
Components and their properties can be imported from the Component Model Library (CML). The CML provides a selection of off-the-shelf components, such as antennas, radios, and even communications waveforms that can be imported into the design. The CML includes constraints, such as which radios use which waveforms. These constraints can be enforced in the design (although this is not yet implemented).



**Figure 7.5-9. Design View of the Communications Link between NGV and Bradley Vehicles**

The communication requirements, design, and property values for gain, attenuation, etc., are passed, through AMIL, to a spreadsheet that runs the Link Analysis models. The results of Link Analysis are fed back into the SysML design model and to a Metrics dashboard that provides summary feedback on the quality of the design and how well it meets requirements.

Size, weight, and power impacts of the selected components are shown on other dashboard panels.



**Figure 7.5-10. The Communications Dashboard**

The Link Analysis spreadsheet is just one example of how specialized tools that support the communications domain can be connected through AMIL. Other tools in this domain that could be connected include FEKO, an electro-magnetic modeling tool that allows mesh models of vehicles, antennas, etc. to be imported into the tool. Once the models are in the tool, antenna to antenna isolation and radiation pattern analysis can be performed to help optimize the vehicle's top deck layout of antennas and structures. The output of this analysis will feed directly into the Cosite and Link analysis. Changes to antenna location made by a mechanical engineer would trigger a re-analysis of the communications, providing immediate feedback on the change.

The reference architecture provides a system level starting point for design, and defines the interfaces and critical properties for domain engineering groups, such as communications. The interconnectedness of the design, at the top level through the SysML reference architecture, and at the tool level with AMIL, allows the impacts of design decisions in other domains and in other components to be understood and assessed.

### 7.5.3 Electronic Warfare Example

#### 7.5.3.1 Introduction

This section provides insights on how the META program approach could be applied to an Electronic Warfare (EW) System. The section provides a preliminary generic description of the major elements of the EW system and how design elements could be assessed in the future.

The classical description of an Electronic Warfare is: Military action involving the use of electromagnetic energy to determine, exploit, reduce or prevent hostile use of the electromagnetic spectrum through damage, destruction and while retaining friendly use of the electromagnetic spectrum. There are three divisions within electronic warfare.

1. **Electronic Attack:** That division of electronic warfare involving the use of electromagnetic or directed energy to attack personnel, facilities or equipment with the intent of degrading, neutralizing or destroying enemy combat capability. This area is also referred to as EA.
2. **Electronic Protection:** That division of electronic warfare involving actions taken to protect personnel, facilities, and equipment from any effects of friendly or enemy employment of

electronic warfare that degrade, neutralize or destroy friendly combat capability. This area is also referred to as EP.

3. Electronic Warfare Support: That division of electronic warfare involving actions taken by or under direct control of an operational commander to search for, intercept, identify and locate sources of intentional and unintentional radiated electromagnetic energy for the purpose of immediate threat recognition. Thus, electronic warfare support provides information required for immediate decisions involving electronic warfare operations, threat avoidance, targeting and other tactical actions. This area is also referred to as ES.

The section provides a high level overview of an EW system modeled in SysML within the context of Model-based Systems Engineering (MBSE). MBSE can be used effectively to manage, to reduce cycle time, and to improve communications among the diverse engineering disciplines necessary for designing complex systems [SF10], [SF09].

The scope of this section is limited to a plausible modeling approach using SysML for an illustrative EW system, abstracted by the Archetype blocks shown in 7.5-11. For simplicity, the EW system implementation has been narrowed down to a pod for attachment to a host platform (Tactical Aircraft or Unmanned Aircraft Vehicle (UAV)). For extensive background and theory of operation of Electronic Warfare (EW) systems, reference [DCS99] is recommended.

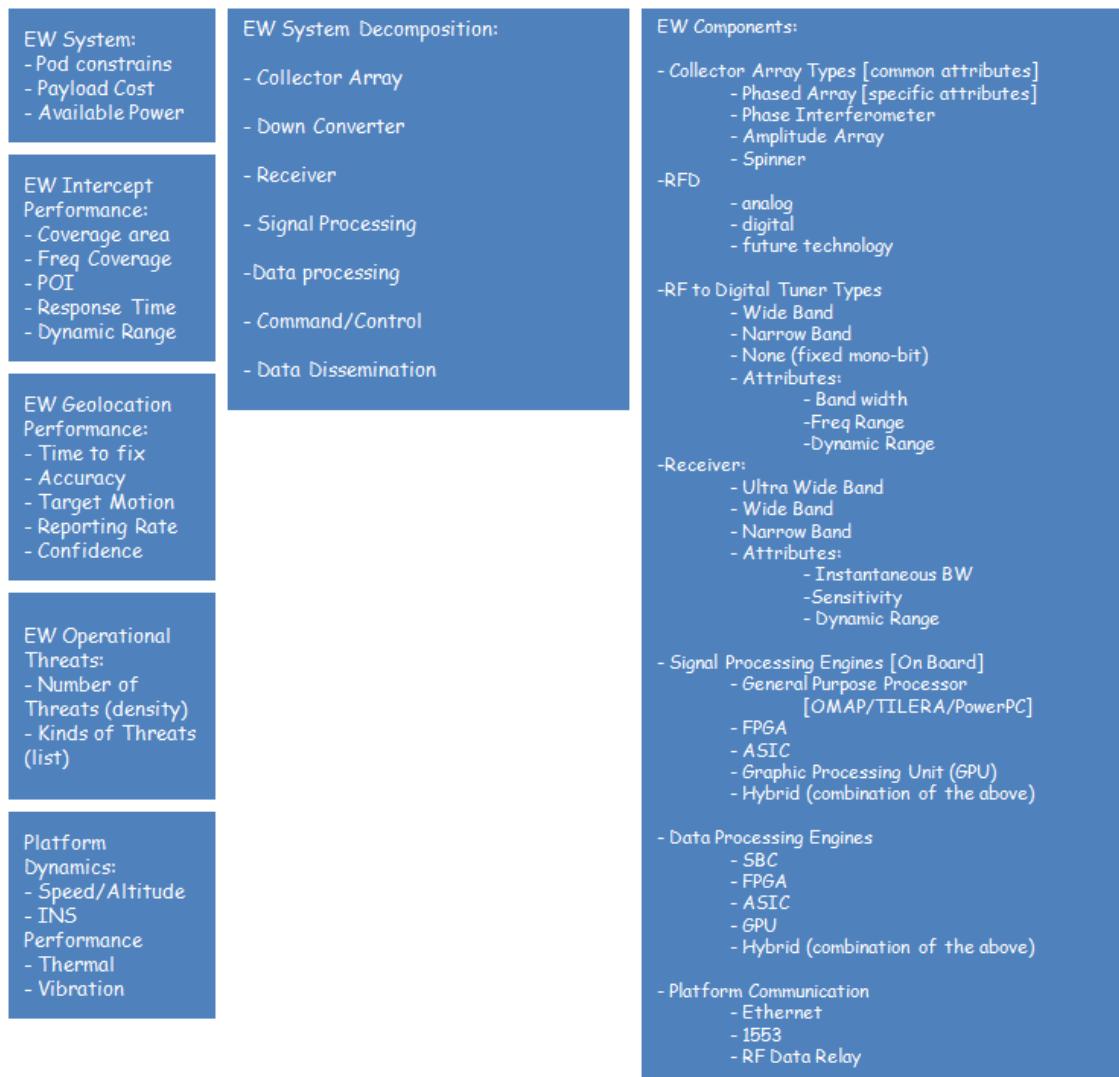
### **7.5.3.2 Archetype Summary**

The EW example has identified a preliminary list of Archetypes (see 7.5-11) that could be used by an engineer to perform and evaluate system level trades.

The following items are some key performance areas that influence operational effectiveness of the EW system.

1. EW System constraints: for this example, a pod represents the carrying structure that would attach to the host platform and would contain all of the EW sensor electronics and associated antennas. The pod size, stowage volume, stowage weight, and available power constrain the EW system. The total cost of the EW system should also be considered in this example.
2. EW Intercept Performance: this represents the operational environment in which the EW System would operate. For this example, it is assumed that the operational environment has an established geographic area of interest (“footprint”) where the EW System will be operating. The EW system has specified response times, scanning rates, probabilities of detection and identification, and other parameters defining how it is expected to gain intelligence on these threats.
3. EW Geolocation Performance: a typical use of an EW System is to provide location information on enemy threats. Performance settings associated with this location are stated in response time, probability of intercept, accuracy, revisit rate, and target motion.
4. EW Operational Parameters: the EW System must support the specific frequencies, bandwidths, modulation types, modes, and number of threats that it will encounter.

The Archetype figure also shows a generic top level system hierarchy and list of candidate components or design features that could be considered by the EW System designer. It is anticipated that the Archetype view would enable the designer to manipulate a set of these items to review changes and impacts to the EW System.



**Figure 7.5-11. EW System Abstraction of Archetypes**

For example, increasing the EW Operational Threat Density, may require additional processor throughput to meet response times. Increasing the number of processors may require additional power, space, and cooling. The Archetype view could highlight the viability of this change and allow the designer to immediately assess if the pod power and volume could support this.

Today's EW System design team uses a loosely coupled collection of special purpose software tools, as well as internally developed MATLAB models and spreadsheet analysis methods to perform trades like the one described above. This requires a high level of expertise as well as access to tools in several development environments, and can be inefficient and time consuming. Integration of these tools, through reference architecture archetypes and through AMIL could speed the design process.

#### 7.5.4 META Challenge Problem

To provide context for the IFV systems engineering studies conducted under the META program, several challenge problems were addressed. This section provides the Ramp

Assembly challenge problem created by the BAE Systems Team. This challenge problem was created to provide examples of complex systems-level vehicle requirements, as well as a multi-domain (electrical, mechanical, and control systems) challenge to illuminate some of the complexities associated with military combat vehicle design.

The problem materials are packaged in two parts: the challenge problem main body, plus a related Combat Fighting Vehicle Performance Specification specifically crafted to feed that challenge problem.

#### **7.5.4.1 Design Space Challenge Problem: Ramp Assembly**

##### **7.5.4.1.1 Challenge Problem Purpose**

Enable the META and META II participants to illustrate key capabilities of the technologies and approach they have and/or are developing in the context of a non trivial problem that exists in the combat vehicle design space today.

##### **7.5.4.1.2 Challenge Problem Goals**

This Challenge problem will:

- Invite alternative solutions for the ramp assembly and its supporting subsystems and/or components to enable subsequent selection of solutions for further design exploration
- Encourage trade-off of multiple inter-related subsystems/component alternatives in order to realize an optimal solution for the given set of criteria
- Enable META/META II participants to showcase their specific toolset capabilities without requiring them to showcase all aspects of the challenge problem
  - Encourage teaming of META/META II performers to stretch toward a large-scale problem
- Be achievable by the September Meeting of the META and META II programs (or earlier).
- Invite definition of selection criteria for defining correctness of solution alternatives
- Invite definition of parametric component properties that relate to selection criteria (e.g., if system weight is a criteria, component weight might scale with power output capability)

##### **7.5.4.1.3 The Design Challenge Problem Statement**

*Design a ramp assembly for a Combat Fighting Vehicle (CFV), together with at least one interfacing subsystem, using your toolset and/or workflow, such that the total system solution (ramp plus subsystem(s) of interest), is optimized to a set of system-level Correctness Criteria, and is additionally conformant to the System Requirements referenced herein.*

*META II performers can either team to accomplish the total design, or address an area of the total problem specific to their initiative. There are aspects of the ramp design problem pertinent to all META/META II Technical Areas: It is highly scalable, contains contributions from every design domain, including cyber-physical subsystems, and is very addressable from an operational context.*

##### **7.5.4.1.4 Ramp Concept of Operations**

###### **Mission:**

The ramp assembly enables mechanized infantry squad mount/dismount operations with a CFV. The ramp assembly is also used to assist in upload/download of supply classes (e.g., ammunition, mission equipment, food and water, and spare and failed parts) and onload/offload of soldiers requiring evacuation and medical attention.

### **General Description:**

The ramp assembly is an automated inclined vehicle egress/ingress pathway that connects the CFV squad compartment with the ground surface.

### **Ramp Operations Battlefield Context:**

Squad mount/dismount operations occur at decisive and tactical locations on the battlefield on a variety of terrain conditions (e.g., at extreme slopes, and on concrete/asphalt, dry, or muddy surfaces). Typically the CFV is oriented in the direction of the mission objective to maximize protection against hostile fire, direct fires for infantry dismounted assaults, and destructive fires against threat vehicles. The CFV will turn-off squad compartment interior lighting when the ramp is opened. The ramp assembly will either provide and/or adapt ballistic protection against ballistic threats.

### **Ramp Operations Activities:**

Ramp operations activities begin when the CFV is at its decisive and/or tactical position on the battlefield. The set activities needed to support ramp operations are the same whether they be to mount/dismount an infantry squad, or to assist in the upload/download supplies or to onload/offload soldiers in need. The set of ramp operations activities includes the following:

- Clear the ramp area for safe operation
- Command and control of ramp operations for the vehicle commander, squad leader, and driver
- Unlatch/Latch Ramp
- Lower/Raise Ramp
- Mount/Dismount an infantry squad
- Assist in upload/download of supplies and/or evacuation of soldiers in need
- Stop ramp motion upon operator initiation

### **Typical Ramp Operations Sequence:**

The driver clears the ramp area for safe use once the squad leader or the vehicle commander orders the squad to prepare to dismount. It should be noted that at any given time the presence of soldiers in the vicinity of the ramp area can occur. Once the driver determines that the ramp area is safe, the driver lowers the ramp. Squad compartment interior lights will be turned off when the ramp initiates opening. Upon the squad leader's order, the infantry squad dismounts up to 2 soldiers abreast with load bearing equipment and assigned individual/crew served weapons and sometimes outfitted with Arctic gear. Command and Control (C2) of ramp operations involves operator controls and indicators (e.g., ramp safety, motion, open/closed positions, and latched/unlatched) for the vehicle commander, driver, and squad leader. Once the infantry squad dismounts and clears the ramp area, either the dismounted squad leader notifies the driver that the ramp can be closed or the driver

determines that the ramp can be safely closed. The driver informs the vehicle commander that the vehicle is ready for maneuvers when the ramp is in the closed position. At any given time ramp movement can be stopped by the driver.

### **Concept Environments:**

The ramp assembly will operate in extreme cold and hot weather conditions (temperature and humidity) and restrict the entrance of rain, hail, snow, and water when closed. The ramp assembly will handle the shock loads and vibration of vehicle movement on hardened surface roads (concrete/asphalt) and cross-country terrain, and the transportation shock loads (including rail hump) and vibration associated with sea, rail, truck, and/or air transportation modes.

### **Maintenance Concept:**

Cleanliness of the ramp assembly will be maintained by either steam or water washing equipment.

#### **7.5.4.1.5 System Requirements**

See the accompanying document “Ramp Assembly Design Space Challenge Problem Combat Fighting Vehicle Performance Specification” (Challenge Problem CFV Perf Spec.docx). Each participant is encouraged to

- Parameterize quantitative constants in these requirements such that tool solutions can be easily reused if such quantitative values change.
- Decompose these System-level (Vehicle-level) requirements to lower level requirements (subsystem, assembly, component) as necessary to support design of the ramp assembly and any supporting chosen subsystems.

#### **7.5.4.1.6 Context Elements**

The following are systems or environmental elements that interface with or can affect the performance of the ramp assembly:

- Ground/Terrain (e.g., asphalt, concrete, dirt/mud, gravel, contour)
- Soldier ( Safety, physical stature, ergonomics)
- Natural Environment (e.g., temperature, humidity, dust, solar radiation, salt fog, rain, hail, snow)
- Hull
- Appliqué Armor
- Electrical/Hydraulic Power Source
- Crew/Operator Indicators & Controls
- Software
- Actuator(s) (e.g., electrical/hydraulic motors)
- Sensor(s) (e.g., position, limit)
- Latches and Locks

#### **7.5.4.1.7 Sample Correctness Criteria**

Use any of the factors below or an alternative approach of your choosing to evaluate the optimal correctness of the total ramp assembly and supporting subsystems/components solution(s):

These factors should be treated as variables rather than constants.

- Criteria
  - Space
  - Weight
  - Power Consumption
  - Heating/Cooling
  - Cost
  - Human Systems Integration (HSI)
  - Force Protection
  - Survivability
- Weighting factors for each criterion
- Utility functions for select or all criteria

#### **7.5.4.1.8 Sample Analysis of Alternatives (AoA) Approach:**

Consider the following alternatives to be traded:

- Alternative discrete components
- Alternative parametric attributes of components
  - Attribute values functions of other traded values
- Alternative number of component instances
- Alternative combinations of and interconnectivity of components
- Alternative location of components within a common constraining envelope
- Combinations of any and all of the above

#### **7.5.4.2 Challenge Problem CFV Performance Specification**

This section provides a sample subset of generic Combat Fighting Vehicle performance specifications that can be used to drive the requirements for a challenge problem. This section was originally delivered as a separate, independent document. This is reflected in the wording and some of the definitions and explanations provided. These have been left in this form so the section can more easily be removed and used to support a challenge problem exercise.

Some of the specifications included here are specific to the ramp assembly itself, while others refer to the vehicle in general, but critically influence the ramp development. These have been color-coded to help the reader differentiate the two.

##### **Specification Highlight Legend:**

**Green text** – Chassis/Mobility requirements

**Yellow Text** – Design constraint and/or ramp-related requirements

#### **7.5.4.2.1 Challenge Problem Scope**

#### **7.5.4.2.1.1 Identification**

This document is a typical set of customer requirements of varying degrees of quality that could be experienced during the development of a complex cyber-physical system, such as a CFV for the DoD. In particular, this document limits its scope to requirements that would likely influence the design of the Ramp Assembly portion of such a CFV. A new innovative technology and model-based approach for DoD material development must contend with customer requirements.

#### **7.5.4.2.1.2 Program Overview**

The goal of the META program is to reduce the DoD development cycle time by a factor of 5x over current cycle time. The META program applies an innovative technology employing a model-based approach to revolutionize the design and verification processes currently used by the DoD industry. META program objectives include: developing new metrics and flows for the innovative model-based material development approach, and defining and developing the new infrastructure (tools, models, component/manufacturing data bases) required for the industry.

#### **7.5.4.2.1.3 System Overview**

The CFV is a tracked, medium armored vehicle which provides cross-country mobility, for mounted firepower, communications, and protection to a mounted mechanized infantry squad, and overwatch support for a dismounted infantry squad.

#### **7.5.4.2.1.4 Document Overview**

This document is a “representative set” of performance, functional, interfaces, and design constraint requirements for a CFV, which upon further decomposition, would influence the design of the Ramp Assembly of the CFV. Both mechanized infantry problem and solution domains in breadth and depth are stated as requirements. The requirement statements vary in maturation and quality due to issues such as: necessity, conciseness, measurability, clarity, implementation/design freedom, attainability/feasibility, completeness and stand-alone, consistency, verifiability, singularity, uniqueness, proper level, and positivity.

### **7.5.4.2.2 Requirements**

#### **7.5.4.2.2.1 Performance Requirements**

Unless otherwise specified, performance requirements in the following paragraphs shall be met with the CFV at maximum combat weight, resting on a flat, hard, level surface, and over the range of environmental conditions specified herein. Requirements relating to personnel shall apply to males in the 5th through 95th percentile in stature wearing Mission Oriented Protective Posture (MOPP-IV) gear and Arctic gear.

##### **7.5.4.2.2.1.1 Operational Profile**

The CFV shall be capable of 24 continuous hours of combat as follows:

- a. Sixteen hours shall consist of:

35% (5.6 hr) at rated engine idle speed.

35% (5.6 hr) over cross-country terrain from 2.0 miles per hour (mph) to maximum safe speed.

20% (3.2 hr) over dirt and gravel roads from 10 mph to maximum safe speed.

10% (1.6 hr) on hard-surfaced roads at 10 mph to maximum operating speed.

b. Eight hours shall be at silent watch with electrical equipment operated as needed for no more than three continuous hours, depending on ambient temperature, without recharging batteries.

#### 7.5.4.2.2.1.2 Acceleration

The CFV at combat weight shall accelerate from a standing start with the engine idling to 50 mph in not more than 25 seconds under nominal conditions. The CFV, at curb weight, shall accelerate from 0 to 50 mph in not more than 20 seconds.

#### 7.5.4.2.2.1.3 Slope Operation

The CFV shall ascend, descend, and emplace on dry slopes up to 60% either forward or backward, and shall maintain at least 15 mph in the forward direction while climbing hard-surfaced slopes up to 15%. The CFV shall maneuver on dry side slopes up to 45% either forward, backward, or emplace. The CFV shall support all ramp operations required herein while emplaced per this requirement.

#### 7.5.4.2.2.1.4 Steering: Pivoting

The CFV shall pivot 360 deg right or left within a 35-ft diameter circle.

#### 7.5.4.2.2.1.5 Water Operation: Fording

Under its own power, the CFV without special preparations shall ford water up to 50-inch deep with up to 35% embankment slopes, while retaining full functionality.

#### 7.5.4.2.2.1.6 Towing

The CFV, operating either forward or in reverse, shall tow comparable CFVs over cross-country terrain. In the forward direction, the CFV shall be capable of towing such a CFV cross-country at up to 5 mph for 10 miles.

#### 7.5.4.2.2.1.7 Survivability: Armor

The CFV armor protection shall be as specified in Appendix XXX. The CFV shall provide mounting provisions and space claim to mount supplemental armor as described in ICDs XXX. The CFV shall provide protection against 14.5 mm machine gun and RPG-7 threats.

#### 7.5.4.2.2.1.8 Auxiliary Systems

##### 7.5.4.2.2.1.8.1 Intercom

The CFV shall accommodate a vehicular intercommunication system with controls at the commander's station and communications ports at each vehicle member station.

##### 7.5.4.2.2.1.8.2 Rear Ramp

The time required for the rear ramp to fully open or close with the engine running shall not exceed 10 seconds. The ramp shall incorporate a lock/unlock mechanism.

##### 7.5.4.2.2.1.8.3 Seals

Static seals shall prevent Class II and Class III leaks. Dynamic seals shall prevent Class III leaks. Class II and Class III leaks are defined in paragraph 6.3.10 herein.

#### **7.5.4.2.2.1.8.4 *Blackout Lighting: Interior Lighting***

All interior lights, except lights for turret control and turret drive power indication, shall extinguish automatically when either the rear ramp or the rear door is opened.

#### **7.5.4.2.2.1.8.5 *Driver's Switches and Indicators***

The CFV shall provide the following operator controls and indicators:

- a. Ramp Up/Down switch and unlocked indicator

#### **7.5.4.2.2.1.9 *Emergency Operation***

The CFV shall provide an emergency operation capability in the case of electronics failures. Vehicle operations requiring backup include:

- a. Fuel Pump operation
- b. Steering operation
- c. Transmission operation
- d. Ramp up/down
- e. Ramp Lock/unlock

### **7.5.4.2.2.2 *Physical Characteristics***

#### **7.5.4.2.2.2.1 *Weight***

The air shipping weight of the CFV shall not exceed 60,000 pounds. The curb weight shall not exceed 100,000 pounds. The maximum combat weight shall not exceed 120,000 pounds.

#### **7.5.4.2.2.2.2 *Dimensions.***

The dimensions when configured for shipping, height shall not exceed 120 inches, width 110 inches, and length 250 inches.

#### **7.5.4.2.2.2.3 *Angle of Approach/Angle of Departure***

The angle of approach for the CFV, defined as the angle between the ground and a line through the forward most part of the hull and track, shall be a minimum of 75 deg. The angle of departure, defined as the angle between the ground and the rear-most part of the hull and track (excluding the pintle) up to at least 40 inches, shall be a minimum of 50 deg.

#### **7.5.4.2.2.2.4 *Ground Clearance***

The CFV shall have a minimum ground clearance to the bottom of the hull of 18 in at the front and 16 in at the rear.

#### **7.5.4.2.2.2.5 *Interior Arrangement: Space Allowance***

Space calculations shall use a 95<sup>th</sup> percentile (in stature) male wearing Arctic clothing and MOPP-IV gear. Space allocation for the squad members, driver, gunner, and commander shall

be in accordance with DOD96. Interior stowage space shall be provided for the fighting equipment of the squad.

#### **7.5.4.2.2.2.6 Ramp**

The CFV shall include a ramp at its rear that permits rapid entry and exit of personnel and supplies. The ramp shall include a door. The ramp shall satisfy the following requirements:

- a. Incorporates a blackout state position sensor or switch.
- b. Restricts the entrance of water into the CFV during fording.
- c. Has a means of being padlocked from the outside.
- d. Permits side-by-side mount/dismount of two 95<sup>th</sup> percentile (in stature) males wearing Arctic clothing and MOPP-IV gear.

#### **7.5.4.2.2.3 Environmental Conditions: Storage and Transport**

The CFV shall be capable of being stored and in transit without sustaining damage under the climate design types hot, basic, cold, and severe cold, including all daily cycle categories as defined in [DOA79] table 2-1; i.e., -60 °F to +160 °F induced air temperature.

##### **7.5.4.2.2.3.1.1 Storage and Transit Humidity**

The CFV shall be capable of being stored and in transit without sustaining damage under the climatic design types hot, basic, cold, and severe cold, including all daily cycle categories as defined in [DOA79] table 2-1; i.e., nil to 100% induced relative humidity.

##### **7.5.4.2.2.3.1.2 Storage**

The CFV shall not require preservation for storage less than 120 days. The CFV shall require preservation prior to storage exceeding 120 days.

##### **7.5.4.2.2.3.1.3 Altitude**

The CFV shall be capable of being stored and in transit up to 40,000 feet above sea level.

#### **7.5.4.2.2.3.2 Operating Conditions: Climate**

The CFV shall be capable of operating under the conditions specified in DOA79, for the climatic categories hot and basic without a cold start aid, and categories cold and severe cold with an aid, with the exceptions in paragraph 3.3.2.2.

#### **7.5.4.2.2.3.3 Steam and Waterjet Cleaning**

The CFV shall demonstrate no performance degradation and show no evidence of damage or deformation following a steam and waterjet cleaning process which uses a cleaner conforming to P-C-437 Type II, P-D220D, or commercial equivalent. Jet pressure shall be  $100 \pm 10$  pounds per square inch gage (psig) for steam and  $40 \pm 10$  psig for water. The jet shall be applied perpendicular to the assembly from a distance of not more than 1 foot for steam and not more than 3 feet for water. The assembly shall be subjected to the jet at the rate of not less than 1 ft<sup>2</sup>/min.

#### **7.5.4.2.2.4 Reliability**

The CFV including Government furnished equipment shall maintain at least 500 Mean Miles Between Failures (MMBF) when operated as described in 3.1.1.1. The CFV Mean Time Between Failures (MTBF) shall be greater than 120 hours (Threshold) and 168 hours (Objective).

#### **7.5.4.2.2.5 Availability**

The CFV including government furnished equipment shall maintain achieved availability of at least 0.80 when operated as described in 3.1.1.1. Achieved availability is defined as the ratio of operating time to the total of operating and maintenance time.

#### **7.5.4.2.2.6 Safety**

The CFV shall ensure the highest degree of safety and health consistent with mission requirements throughout its life cycle.

#### **7.5.4.2.2.7 Logistics/Diagnostics: Self-Test Built-In Test (SBIT)**

SBITs, internal to each subsystem, shall execute automatically upon power up and results shall be displayed within 20 seconds of the application of power to the turret electronics.

#### **7.5.4.2.2.8 Design and Construction: Materials**

All materials, parts, and processes selected for use in the CFV construction shall be compatible with the safety, performance, and environmental requirements as specified herein.

##### **7.5.4.2.2.8.1.1 Fungal Growth**

Materials used in the CFV shall not support fungal growth.

##### **7.5.4.2.2.8.1.2 Corrosion Resistance**

Metals and alloys used in the construction of the CFV that are exposed to corrosive environmental conditions shall be corrosion resistant or shall be coated or metallurgically processed to resist corrosion. Except where impractical, dissimilar metal combinations that promote corrosion through galvanic action shall be insulated to prevent corrosion.

#### **7.5.4.2.3 Definitions**

**Curb Weight.** The CFV is complete with all components and systems, fully serviced with liquids and one-fourth full fuel tank, with track pads, driver, no OVE, no weapons installed, no other crew or squad aboard, no BII, AAL or ICOEI, no ammunition or water, and no supplemental armor tiles. Items may be simulated by ballast weights located at the appropriate center of gravity.

**Combat Weight.** The CFV is complete with all components and systems, fully serviced with liquids and a full fuel tank, with track pads, all OVE BII, AAL, ICOEI, 25 mm and 7.62 mm weapons installed, all ammunition and water, crew and squad, and supplemental armor tiles installed. Items, such as crew, ammunition, supplemental armor tiles, etc., may be simulated by ballast weights located at the appropriate center of gravity.

**Approximately.** As close as reasonable for the intended purpose. In the opinion of the operator the item being tested will not cause failure or malfunction of the system, or cause the system to not function.

**Smooth.** In the opinion of the operator, the item being tested does not exhibit discernable erratic operation, chatter, jump, bind, skip, or does not prevent the operator from properly functioning the item being tested.

**Subjectively.** An intuitive and conscious consideration by the operator, that the item being tested, observed, or checked meets or exceeds the intended function.

**Focus.** Clear, without blurriness, objects at a distance of more than 200 m are sharp and clear.

**Subjective Evaluation.** This verification is a subjective evaluation of the operation or response of the system or component in question. Conclusions of success depend on the interpretations of an experienced operator/tester, rather than on numbers derived from instrumentation, bus data, or other quantitative results.

**Hardware/Software Test.** Specific functions, responses, and other parameters of the system or component in question have been measured or determined during Software/Hardware Final Qualification Tests, component tests, or subsystem tests. Therefore, quantitative or instrumented measurements at the system/CFV level may not be required.

**Previous Tests.** Where appropriate, use the procedures and results of tests of other functions as evidence that the requirements of this paragraph are met.

**Classification of Leaks.** Class I: Fluid seepage is not great enough to form drops, but is shown by wetness or color changes. Class II: Fluid leakage is great enough to form drops. Drops do not drip from the item being checked or inspected. Class III: Fluid leakage is great enough to form drops that fall from the item being checked or inspected.

### 7.5.5 Bibliography

- 〔DCS99〕 D. Curtis Schleher, *Electronic Warfare in the Information Age*, Artech House, 1999.
- 〔DOA79〕 Department of the Army (1979), *AR 70-38: Research, Development, Test and Evaluation of Materiel for Extreme Climatic Conditions*, [http://www.apd.army.mil/pdffiles/r70\\_38.pdf](http://www.apd.army.mil/pdffiles/r70_38.pdf).
- 〔DOD96〕 Department of Defense (1995), *MIL-HDBK-759C: Handbook for Human Engineering Design Guidelines*, available at <http://www.hf.faa.gov/docs/508/docs/milhdbk759C.pdf>.
- 〔SF09〕 Sanford Friedenthal, et al, *A Practical Guide to SysML*, The MK/OMG Press, Elsevier Inc, 2009.
- 〔SF10〕 Sanford Friedenthal and Joseph Wolfrom, “Modeling with SysML”, a tutorial presented at INCOSE 2010 Symposium, Chicago, IL, July 2010. Published by The John Hopkins University Applied Physics Laboratory.

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.6 – Advanced Reasoning and Extended Applications of ARRoW Technology**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.6 Advanced Reasoning and Extended Applications of ARRoW Technology.....</b>	<b>1</b>
7.6.1 Applying ARRoW to Automation.....	1
7.6.1.1 The Goal of Reducing Complexity .....	1
7.6.1.2 Complexity Reducers .....	7
7.6.1.3 Engineering Domains .....	21
7.6.1.4 Application of Reasoning Languages.....	29
7.6.2 Co-Analysis and Exploration.....	43
7.6.2.1 Principles behind GEAR.....	43
7.6.2.2 ESKER .....	61
7.6.2.3 Ontology-based Logic Reasoners .....	84
7.6.2.4 ECTo.....	98
7.6.2.5 Co-Analysis Flow using GEAR .....	98
7.6.3 Co-Simulation and T&V .....	99
7.6.3.1 MoCC and Heterogeneous Simulation .....	101
7.6.3.2 Tagged Signal Model.....	101
7.6.3.3 Multi-Physics and Compartmentalization .....	126
7.6.3.4 AMIL Configuration and Specification.....	129
7.6.3.5 Probabilistic Certificate of Correctness .....	134
7.6.4 Distributed Computing Speed-Up Potential.....	138
7.6.4.1 Spatial Computing .....	138
7.6.4.2 Generic Inferencing.....	138
7.6.5 Bibliography .....	140

## List of Figures

Figure 7.6-1. Galileo Reasoning and Evaluation Tool Space.....	1
Figure 7.6-2. Graph Technology Used with Galileo.....	2
Figure 7.6-3. Search is Used for Solving Constraint Satisfaction Problems.....	3
Figure 7.6-4. Organization and Consolidation of Models, Towards a Component Model Library .....	4
Figure 7.6-5. Synthesis, Co-simulation, and Co-analysis Operate Within an Ensemble Environment .....	6
Figure 7.6-6. Control of Co-analysis, Co-simulation, and Synthesis .....	7
Figure 7.6-7. Component Encapsulation with Cached and Triggered Updates .....	8
Figure 7.6-8. Declarative Approach for Constraint Solvers .....	9
Figure 7.6-9. Influence Diagrams are More Useful for Showing the Directed Interactions than a Design Structure Matrix.....	10
Figure 7.6-10. Partially Decomposable Systems .....	11
Figure 7.6-11. DSM Application Domains (from Pimler and Eppinger, 1994) .....	12
Figure 7.6-12. Design Structure Matrix for Information - Utility Functions Trade-off Criteria for Different Design Choices .....	13
Figure 7.6-13. Heterogeneous Interfaces .....	13
Figure 7.6-14. Set-Based Concurrent Engineering Continuously Tracks Open Alternatives While Culling Out Poor Design Choices .....	14
Figure 7.6-15. A Provenance Strategy Manages Development Processes that Require Multiple Steps .....	15

Figure 7.6-16. Sources of Defects that TDD Aims to Mitigate.....	16
Figure 7.6-17. Parallel Development Process Speeds Development.....	17
Figure 7.6-18. Abstraction Levels in a Multi-Physics Domain Example .....	18
Figure 7.6-19. Test Space Sampling Generates a Histogram for PCC Evaluation.....	19
Figure 7.6-20. Diagnostic Aids Used to Discover a Design Problem.....	20
Figure 7.6-21. Paths to Stochastic Formal Verification - Stochastic Elements Starting from Co-analysis to Co-simulation .....	21
Figure 7.6-22. Graph of Analysis Archetype for Projectile Fly-Out Simulation.....	22
Figure 7.6-23. Archetypes for Analyzing a Derived Fault-Tolerant Reliability Requirement.	23
Figure 7.6-24. Auto-Generated Expansion of Reliability Block Diagram .....	24
Figure 7.6-25. Design Structure Matrix Interactions Needed for Knowledge Engineering....	26
Figure 7.6-26. ARRoW's Template-Based Look-Ahead Concept.....	27
Figure 7.6-27. Test Space Exploration Uses Similar Combinatorial Techniques to Search the Test Space as Used for DSE .....	29
Figure 7.6-28. AMIL as a Knowledge Store.....	31
Figure 7.6-29. Code Sample 1: JSON Parser and Store Rules.....	31
Figure 7.6-30. Code Sample 2: GEAR Reasoner Which Finds the Maximum Density from Elements in a Set.....	32
Figure 7.6-31. Code Sample 3: GEAR Web Server with Handlers Pointing to Rules.....	33
Figure 7.6-32. Difference Between Prolog (left) and a Controlled Natural Language (right) ..	37
Figure 7.6-33. An example of a Typical Assisted Editing Environment for Query Development .....	37
Figure 7.6-34. Loading META/CFV Ontologies and SWEET Ontologies .....	38
Figure 7.6-35. Correct-By-Construction Application Involving a Heterogeneous Multi-physics Interface.....	40
Figure 7.6-36. Analysis Archetype Rules.....	41
Figure 7.6-37. Behavioral Archetype .....	45
Figure 7.6-38. Alternate Rule Layout .....	46
Figure 7.6-39. Gear Suite .....	48
Figure 7.6-40. Abstract Representation of the Operating Environment.....	49
Figure 7.6-41. Dispatch Work Center .....	50
Figure 7.6-42. Material Move Specification in Prolog.....	54
Figure 7.6-43. Testing the Logical Specification.....	55
Figure 7.6-44. Adding Material Availability Constraint .....	56
Figure 7.6-45. Architecture of the Optimization Shell.....	64
Figure 7.6-46. AMIL-like Connections Between the Main Application and Server Applications .....	64
Figure 7.6-47. Part of the Prototyped HTML ESKER Query Interface.....	75
Figure 7.6-48. Web-Served Version of ESKER Used to Populate the Knowledge Base.....	83
Figure 7.6-49. Ontological Reasoners Follow a Similar Pattern of Encapsulating the Deeper Semantic with a Uniform Logical Front-end .....	85
Figure 7.6-50. N3 Ontology for Engine and the Equivalent “invisible” Triples They Represent (To the Right) .....	86
Figure 7.6-51. Weight, Horsepower, Speed Reasoner .....	87
Figure 7.6-52. CML Query Example .....	88
Figure 7.6-53. Parts Repository .....	89
Figure 7.6-54. Call to the SWEET Ontology .....	89

Figure 7.6-55. Component Model Library N3.....	91
Figure 7.6-56. CML library query .....	92
Figure 7.6-57. Book Ordering.....	93
Figure 7.6-58. Terrain Querying Similar to Book Ordering.....	94
Figure 7.6-59. The Web Service Composer Will String Together a Sequence of Service Calls from an Ontology to Allow Flexibility in the Creation of a Composable Workflow.....	94
Figure 7.6-60. ISTAR Query (from [SMV11]).....	95
Figure 7.6-61. Ontology for a Set of Components with Properties .....	96
Figure 7.6-62. Reasoner Which Understands How to Derive Mass from Shape Properties....	97
Figure 7.6-63. Flow of Co-analysis from Initial Requirements Using Automation where Possible.....	98
Figure 7.6-64. AMIL Nodes Serve as Plug-Ins Within a Distributed Co-Analysis .....	99
Figure 7.6-65. Composable Workflow Analysis Which is a Loosely-coupled, Service-Oriented Architecture.....	100
Figure 7.6-66. Typical High-speed Co-simulation Network with Direct Data Paths Between Communication Ports .....	100
Figure 7.6-67. MoCC Applied to Modeling of a Drivetrain .....	101
Figure 7.6-68. The tagged-signal model allows interoperability of different MOCC .....	103
Figure 7.6-69. The Tagged-signal Model Encompasses a Family of Simulation Behaviors (from Lee and Sangiovanni-Vincentelli). .....	104
Figure 7.6-70. Tagged Signal Runtime Polymorphism.....	106
Figure 7.6-71. A Typical Architecture of an Agent-Based Co-Simulation.....	107
Figure 7.6-72. Distributed Command Pattern for Selecting Among Alternative Implementations.....	108
Figure 7.6-73. The Application of Middleware .....	109
Figure 7.6-74. Use of Distributed Pattern in Co-simulated Multi-physics Regime .....	126
Figure 7.6-75. Multi-Physics Data Flow and Integration .....	127
Figure 7.6-76. Multi-Physics Behavior Often Occurs Over Non-overlapping Time Intervals, Allowing a Separation of Concerns.....	128
Figure 7.6-77. DSM for Compartmentalizing.....	129
Figure 7.6-78. Launching co-simulation apps .....	131
Figure 7.6-79. Logical Triples and Tuple Configuration.....	132
Figure 7.6-80. The Distributed Simulation is Collapsed into the Monolithic Model “Node-0” .....	133
Figure 7.6-81. A Multivariate PDF Drawn from 3 Normal Distributions and Applied to Solving a Quality Factor, Q.....	135
Figure 7.6-82. Test Space Evaluation Using ESKER to Map Importance Sampled Test Cases .....	137
Figure 7.6-83. Bayes Update of PCC Applied During Reasoning.....	137
Figure 7.6-84. Generic Reasoning Allows Several Different Approaches to Potentially Be Unified .....	138
Figure 7.6-85. Valuation Algebras Used in Generic Inferencing Run the Gamut of Decision Theory .....	139

## List of Tables

Table 7.6-1. Enduring Engineering Properties .....	2
Table 7.6-2. List of Prototype Optimization Problems .....	66
Table 7.6-3. Elicitation Table .....	77
Table 7.6-4. Categorization of Configuration Parameters for Co-simulation and Co-analysis	130

## List of Symbols, Abbreviations, and Acronyms for Appendix

Symbol, Abbreviation, Acronym	Definition
AIDE	ARRoW Integrated Development Environment
AMIL	ARRoW Model Interconnection Language
API	Application Programmers Interface
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CGI	Common Gateway Interface
CML	Component Model Library
DAML	DARPA Agent Markup Language
DCP	Distributed Command Pattern
DL	Description Logic
DSE	Design Space Exploration
DSM	Design Structure Matrix
DSS	Decision Support System
ECTo	Early Concepting Tool
ESKER	Expert-System Knowledgebase Evaluation Reasoner
FIFO	First In First Out
GEAR	Generative Archetype Reasoning
GUI	Graphical User Interface
HLA	High Level Architecture
IFV	Infantry Fighting Vehicle
IPC	Internet Protocol Communications
JSON	JavaScript Object Notation
MOCC	Models of Computation and Communication
NABK	NATO Armaments Ballistic Kernel

<b>Symbol, Abbreviation, Acronym</b>	<b>Definition</b>
OLP	Onto-Logical Programming
OOP	Object Oriented Programming
OWL	Web Ontology Language
PACE	Prototype Agile Component Environment
PCC	Probabilistic Certificate of Correctness
QSP	Qualitative State Plan
RDF	Resource Description Format
RMPL	Reactive Model-based Programming Language
SAF	Semi Automated Forces
SOAP	Simple Object Access protocol
SPARQL	Simple Protocol and RDF Query Language
SVN	Subversion
SWEET	Semantic Web for Earth and Environmental Terminology
TCP	Transmission Control Protocol
TDD	Test-Driven Development
TSE	Test Space Exploration
TSM	Tagged Signal Model
UML	Unified Modeling Language
URL	Uniform Resource Locator
VHDL	Virtual Hardware Description Language
XML	Extensible Markup Language

## 7.6 Advanced Reasoning and Extended Applications of ARRoW Technology

### 7.6.1 Applying ARRoW to Automation

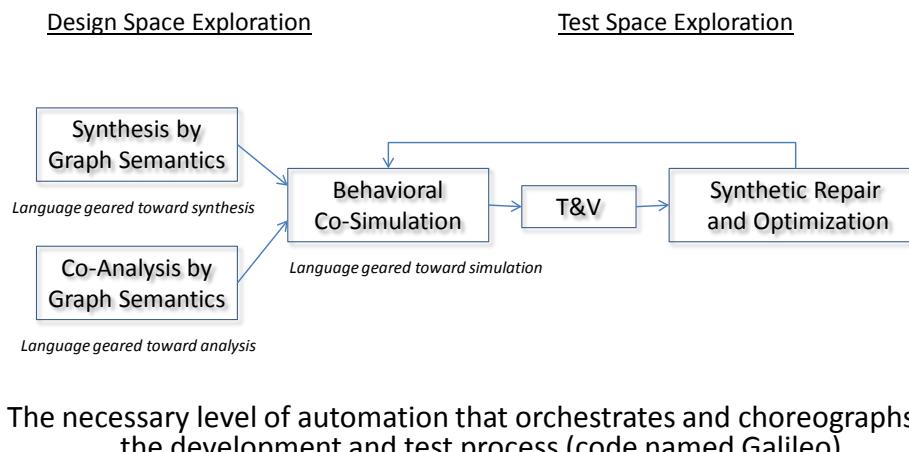
#### 7.6.1.1 The Goal of Reducing Complexity

##### 7.6.1.1.1 ARRoW and the Design Cycle

The aim of ARRoW is to apply an efficient, concurrent evaluation process to the design, test, and diagnose cycle. We broke the reasoning and evaluation capabilities out into general categories that cover the design space and test space, and labeled the automated tools that fit under these categories as Galileo. In practical terms, Galileo covered the planned set of tools that would operate on and reason with ARRoW Model Interconnection Language (AMIL) data and library information.

We further categorized the tools that (1) helped with composition/synthesis, (2) provided co-analysis, and (3) provided co-simulation support. Figure 7.6-1 shows where these tool categories fit within the ARRoW cycle. Since ARRoW defines an adaptive workflow development, we pay special attention to composable workflow reasoners and using a set-based strategy for maintaining flexibility with regard to product requirements.

## The ARRoW automated design cycle



**Figure 7.6-1. Galileo Reasoning and Evaluation Tool Space**

#### 7.6.1.1.2 Graph and Search

We took a data-centric approach to tool development and leaned heavily on the strong organization principles provided by data modeling languages such as AMIL and the standard Resource Description Format (RDF). Two principles that we learn from advances in

information technology are the adages that all information is dependent on a *graph* of some sort and that all engineering is based on *search* principles. In other words, all information is a Graph, and all process is Search.

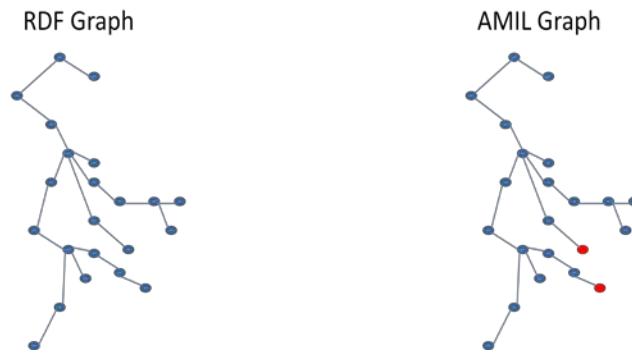
So when we work Design Space Exploration (DSE) and Test Space Exploration (TSE), we rely heavily on deep organization and classification principles, while obviously using computer-automated search techniques to discover solutions and root out problems.

As Table 7.6-1 demonstrates, we apply specific search objectives and craft elements into the graph that allow for the most efficient solution to a problem. So then Galileo becomes a mix of reasoners and solvers that effectively perform a search on a state-space.

**Table 7.6-1. Enduring Engineering Properties**

	Search Objective	Graph Elements
Co-Analysis	Design Space	High-Abstraction
Co-Simulation	Test & Verification	High-Fidelity
Logical Reasoning	Correct-by-Construction	Assume-Guarantee

**Graph.** The ARRoW information organization centers on our use of AMIL and RDF. Figure 7.6-2 demonstrates the significant distinction between AMIL and RDF. A topological view reveals that the two graph languages are structurally identical, but that AMIL provides a dynamic element which can provide additional active content to the data stores.

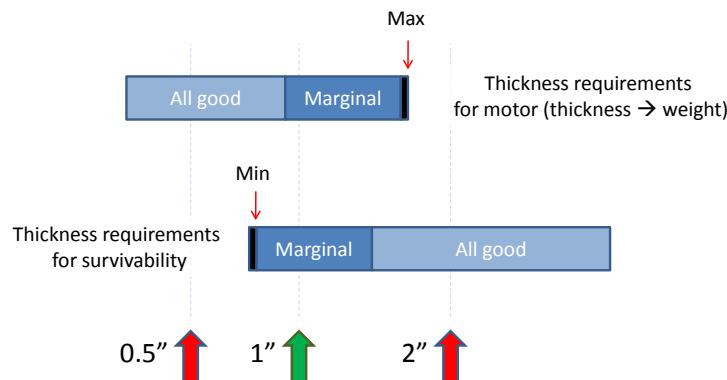


**Figure 7.6-2. Graph Technology Used with Galileo**

As indicated in the figure, the RDF and AMIL graphs are equivalent except for potential behavioral dynamics of the nodes shown in RED. The node evaluation semantics make this an active data store which cannot be duplicated with off-the-shelf triple-store technologies. Only by adding an external reasoner can one implement dynamic solutions; with AMIL this is built-in and used as needed.

**Search.** Figure 7.6-3 shows one example of a search problem applied to a vehicle design exploration demonstration. The search criteria involved finding optimal values subject to possible conflicting constraints.

## Constraint Satisfaction Approach



*Pick from Discrete Choices → optimal not always the best approach,  
as generating a solution with a margin leads to better adaptability and robustness*

**Figure 7.6-3. Search is Used for Solving Constraint Satisfaction Problems**

**Can a graph search itself?** The key element to facilitating efficient search is to use external reasoners to supplant the active content of a language such as AMIL. The set of reasoners and solvers referred to as the codename *Galileo* accounts for this capability.

To demonstrate the needs, we consider one case that could use the dynamics of AMIL and another which we may consider it as overkill.

A practical case is one where a reusable model contains a dynamic calculation which depends on a few other properties of the system. To make the model easy to use, a dynamic calculation is embedded into the AMIL node and then saved as code linked to the CML library. When a user wishes to evaluate the node, no parameters are needed and the user can simply access that value isomorphically as a property. Then the active content of the node gets invoked and returned to the user as a value.

A more complex case can be managed by pairing AMIL with custom reasoners. Such cases, like tracking the meta-information in a large CAD model, can comprise a graph in the form of an extended tree, much like that shown in Figure 7.6-2. Design engineers manage an assembly as a dynamic object with its mass dependent on what the assembly contains. Therefore an aggregated mass calculation on an assembly node is a perfect candidate for an AMIL type of dynamic node. Yet, since current specialty engineering tools like CAD/CAM systems often do not do this well, we instead use back-chaining reasoners because:

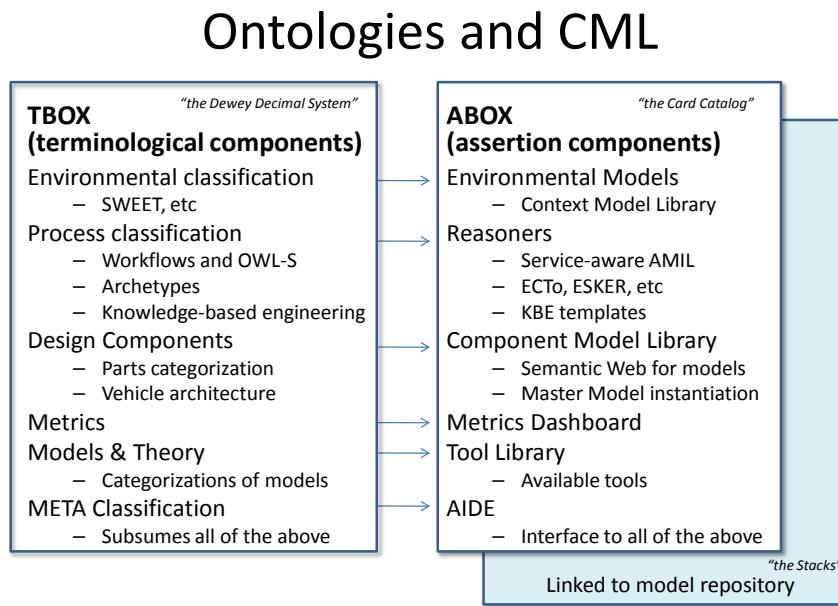
1. **Specificity.** The reasoner can be told to calculate the weight of an assembly main component, the entire assembly, or the weight of the assembly a number of N levels down. An AMIL-like dynamic calculation of assembly weight has no easy way of controlling this.
2. **Efficiencies of computation.** An external reasoner can readily adopt dynamic programming and join tree strategies, which can cut down the search space and the number of computations required.

3. Greater knowledge. For analysis, all states of the system are known. For example, the reasoner does not have to follow a specific path if the valuation of that path is known to not have changed (refer to 7.5.1.4.1).
4. Self-consistency. Side effects caused by a dynamic approach may lead to non-deterministic solutions.
5. Unknown data, underspecified or overspecified data. If data is not available the reasoner can deal with it. If it is overspecified, the reasoner can provide multiple solutions and an explanation for what generated the solutions.

Combining AMIL with reasoners provides a very practical solution without breaking the design tenets of AMIL: the light-weight, adaptive nature that provides the efficiency and broad applicability necessary in the heterogeneous environment. For example, for the problem of over-specified data, we can use a reasoner to provide for a comprehensive set-based approach to development. In this case, the over-specified data (such as alternate design choices) become part of the set of design choices that we can reason against (refer to 7.5.2.2). To leverage AMIL directly, we can embed a query into a node that understands how to call a CAD model like Pro/E and ask for the mass, realizing that the tool manipulating the vehicle structure can do a much better job than in its native environment (refer to AMIL7.5.1.4.1).

### 7.6.1.1.3 Organization of Models and CML

The reasoners we developed interact heavily with the content and classification organization of a comprehensive component model library. The library itself, in keeping with the traditional approach, will provide several interfaces: (1) a classification system based on data ontologies, (2) a meta-information content layer indexed to the ontology [BCW97], and (3) the actual model repository. These interfaces are discussed elsewhere but the overall representation is shown in Figure 7.6-4.



**Figure 7.6-4. Organization and Consolidation of Models, Towards a Component Model Library**

For our reasoners, we will adopt the strategy of referring to classification or *terminological* data as TBOX and instance or *assertion* data as ABOX. This is borrowed from the theory of description logic [LS09] and is at the heart of ontological languages such as Web Ontology Language (OWL) and DARPA Agent Markup Language (DAML).

#### 7.6.1.1.4 Correctness versus Agility

To expedite the fast development of models, the co-analysis and co-simulation approaches lean heavily on providing coarse graining fidelity appropriate for the problem at hand. This approach has long been standard practice on vehicle integration efforts and for large scale simulations such as High Level Architecture (HLA) for OneSAF as well as in the Electronic Design Automation field. Rationale for using coarse graining of models follows.

- Needed for systematic evaluation
  - One high fidelity model surrounded in a sea of low-fidelity representations
  - Iterate through set according to importance of evaluation criteria (down-select, PCC, etc)
  - Use the ideas of templates and archetypes to switch easily between models
- Complexity management
  - Integrating a ground vehicle with a mix of components
    - Real and virtual
    - High and low fidelity
  - SAF-like large scale simulations
    - Three levels in SAF-like simulation environments
      - Standard, autonomous, and focused
    - Interactions between entities of different levels of resolution tested
    - Allows users to “dial up” the level of resolution where needed
- Evaluation performance
  - EDA languages (VHDL) have idea of architecture fidelity *built-in* to the semantics
    - Uses the semantics of *configuration* declarations
    - No hope of simulating or proving designs without similar automated composability
  - One entity high-fidelity and thousands of others behavioral
  - What general purpose model language offers this?

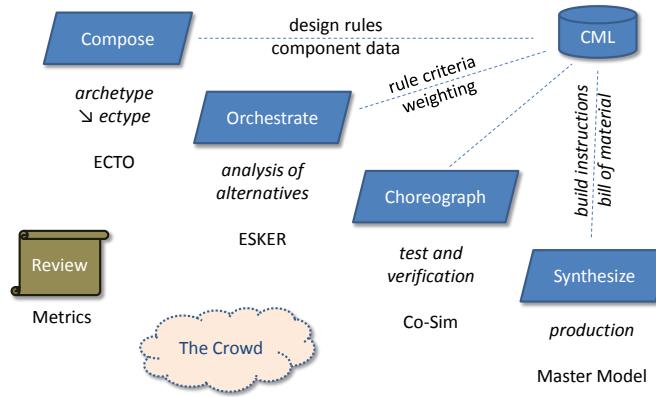
The level of abstraction needs to be carefully evaluated to avoid problems with leaky abstractions (or abstraction leakage).

The significant aspect of the ARRoW toolset is that it readily adopts the set-based approach to extend the plug-and-play philosophy to models of different fidelity representation (i.e. alternative design choices are often polymorphically equivalent to alternate fidelity models as the interface remains invariant). Like the electronics EDA industry has discovered, a large scale simulation effort will fail without this capability.

#### 7.6.1.1.5 The Conductor’s Role

Synthesis, co-analysis, and co-simulation tools can be classified according to the “conductor” guidance that they require, which could be one of Composition, Orchestration, Choreography, and Synthesis. Figure 7.6-5 illustrates the roles that the conductor plays.

## Facilitating Ensemble Engineering



**Figure 7.6-5. Synthesis, Co-simulation, and Co-analysis Operate Within an Ensemble Environment**

The ensemble is both the aggregation of the components that makes up the vehicle and the designer crowd that takes part and conducts the development. In a real-world ensemble, different roles are adopted during different phases of the development. The crowd is able to manipulate the design by using automated tools at each phase and thus assume different “directorial” duties.

1. During composition, we use rule-driven archetypes to create concept *ectypes*<sup>1</sup>, which represent possible implementations of an idealized design.
2. We then orchestrate the down-selection of potential designs (the ectypes) among the conceptual alternatives by systematically reasoning over trade-offs.
3. Once selected, the top ensemble designs are more loosely choreographed to reveal emergent behaviors and requirements failures. The reasoner essentially conducts the exercise over test space.
4. The final step is the production, which captures all the knowledge to synthesize the design suitable for manufacturing.

The tooling interface provides a firewall to the knowledge within the library.

Metrics are used along the path to non-intrusively review and diagnose results<sup>2</sup>. For example, do individual parts cooperate or compete to accomplish a task? Or are they co-operating but competing for scarce resources? We can execute the metrics tools (see Appendix 7.2) to provide measures of effectiveness in which to answer these questions.

This is designing engineering tools for resilience, adaptivity, and emergence as we encourage the crowd to improve the design by using various reasoners that we supply and that they can

<sup>1</sup> An ectype is defined according to Merriam-Webster as an instance or facsimile of an archetype.

<sup>2</sup> Metrics are also useful as criteria for down-selecting, as in step #2, but in this case play a more passive and reactive role, not directly involved with the original design.

customize. Customization of templates has been used effectively in the architectural design and architectural engineering fields [RK06] with building information modeling products such as Gehry Technologies' Digital Project or Autodesk's Revit [GW06].

The essential gambit behind our tooling approach is to keep the style open enough in terms of languages and reasoners that the crowd can easily adopt the tools. And by the same token, they can improve on the tools if they are provided the motivation.

### 7.6.1.2 Complexity Reducers

Besides AMIL and the CML, we are merging several technology approaches to help reduce the complexity of a cyber-physical design.

#### 7.6.1.2.1 Knowledgebase Engineering

The concept of knowledgebase engineering relies on stored tacit design rules and archetypes of idealized representations to automatically generate products or artifacts. The information can be described by patterns, templates, or by concrete and abstract prototypes used in such a way that we can invoke them from a reasoner, inference engine, or expert system.

The key insight that we provide is to combine the traditional and state-of-the-practice KBE approaches with ontological and semantic web technology.

#### 7.6.1.2.2 Reasoners and Archetypes

Following a KBE strategy, we refer to archetypes as tacit information that can be used to establish early concepts, established process workflows, and other design rules. The idea of aligning archetypes with reasoners leads to the term Generative Ensemble Archetype Reasoners (GEAR) to describe these capabilities (Figure 7.6-6). Reasoners, patterns, templates, design rules, and archetypes are essentially synonyms for this generic capability.

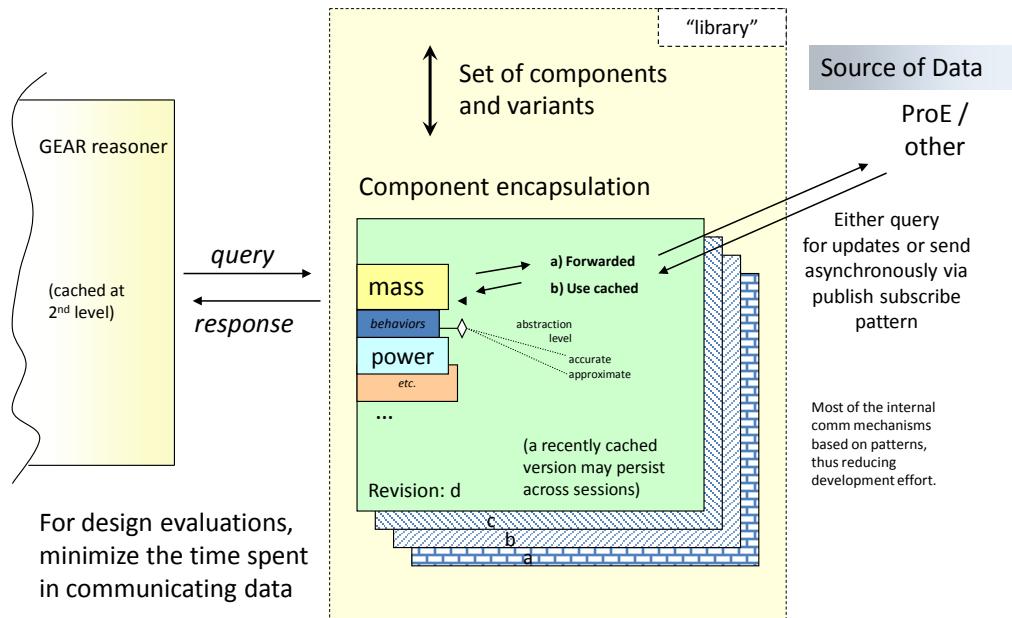
Galileo/GEAR			
<i>compose</i>	<i>orchestrate</i>	<i>choreograph</i>	<i>synthesize</i>
ECTo concept	ESKER DSE	Co-Simulation T&V	Master Model production

**Figure 7.6-6. Control of Co-analysis, Co-simulation, and Synthesis**

Other reasoners such as the KBE Templates [KDG11] used in design automation tools such as Pro/E and Catia and other generative reasoners used in design [AUK10] complement the reasoners that we have developed for META.

#### 7.6.1.2.3 Components

The reuse of components enables faster and more efficient product development [HDJ08]. The general pattern of encapsulating models is shown in Figure 7.6-7.



**Figure 7.6-7. Component Encapsulation with Cached and Triggered Updates**

The strength of encapsulating objects was one of the main driving strategies for providing the AMIL dynamic behavior. If the behavior of the internal dynamics is simple enough, we can simply encapsulate this within a node.

A prevailing issue is how to find the model we need and unambiguously apply it in the correct context and with the correct process workflow. This is essentially a search problem as well.

#### 7.6.1.2.4 Ontologies and Semantic Constraints

The application of ontologies to reasoning brings a tremendous advantage to search strategies. The constraints associated with strong classification schemes can help narrow in on a solution much quicker than a free-form “Google-like” search is capable of. This comes with a caveat that narrowing the search limits creative possibilities, but that is only dependent on the amount of semantic meaning we can apply to the problem. For example, eventually *Controlled Natural Language* interfaces will be able to infer possibilities that extend beyond the strict classifiers.

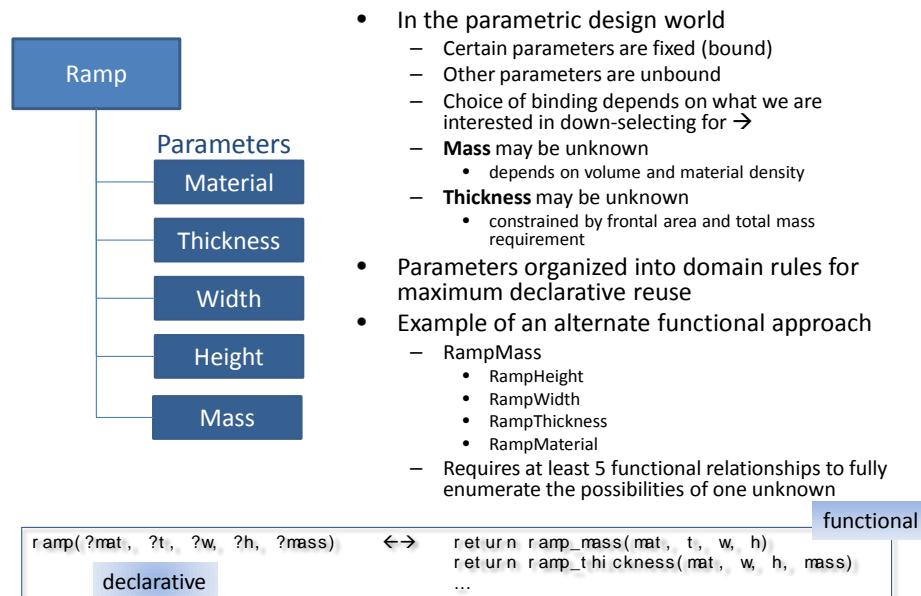
#### 7.6.1.2.5 Declarative Semantics and Logic

The declarative style corresponds to several useful approaches already accepted in engineering development:

- Mathematical equations as in Modelica [BL08], etc.
- Finite domain problems (e.g., *the set of numbers between 1 and 10*)
- Constraint satisfaction problems  
(e.g., *how many ways will these pieces fit into this compartment?*)
- Engineering parametric solvers
  - SysML
  - Inverse kinematics

- Declarative and constraint logic domains which can be expressed as 5<sup>th</sup> generation languages (5GL) such as Prolog [PRP94].
  - Relational database queries via Search and Query Language (SQL) and Simple Protocol and RDF Query Language (SPARQL)

The clearest explanation of the declarative approach is best described by example. Figure 7.6-8 shows how the declarative approach works in a practical application of accessing a parametric knowledgebase with *bound* and *unbound* parameters.



**Figure 7.6-8. Declarative Approach for Constraint Solvers**

The functional and data-flow styles also apply to many conventional applications, and we will apply those as well, but the declarative style finds particular utility in reasoning. For example, the classic SPARQL query for a triple-store data element is declarative and is essentially a subset of the declarative query illustrated above. A declarative approach often leads to problem-oriented abstractions, and not the solution-oriented strategy that ends up employed in a procedural approach.

#### 7.6.1.2.6 DSM and Influence Diagrams

For the Design Space Exploration tools (Expert-System Knowledgebase Evaluation Reasoner [ESKER] in particular, refer to 7.6.2.2) that we constructed, we incorporated the general problem solving strategy of the Design Structure Matrix (DSM). The DSMs are generally treated in matrix or spreadsheet form but we extended this to include the concept of influence diagrams [JP05]. The general idea is illustrated by the following anecdotal application of a DSM search:

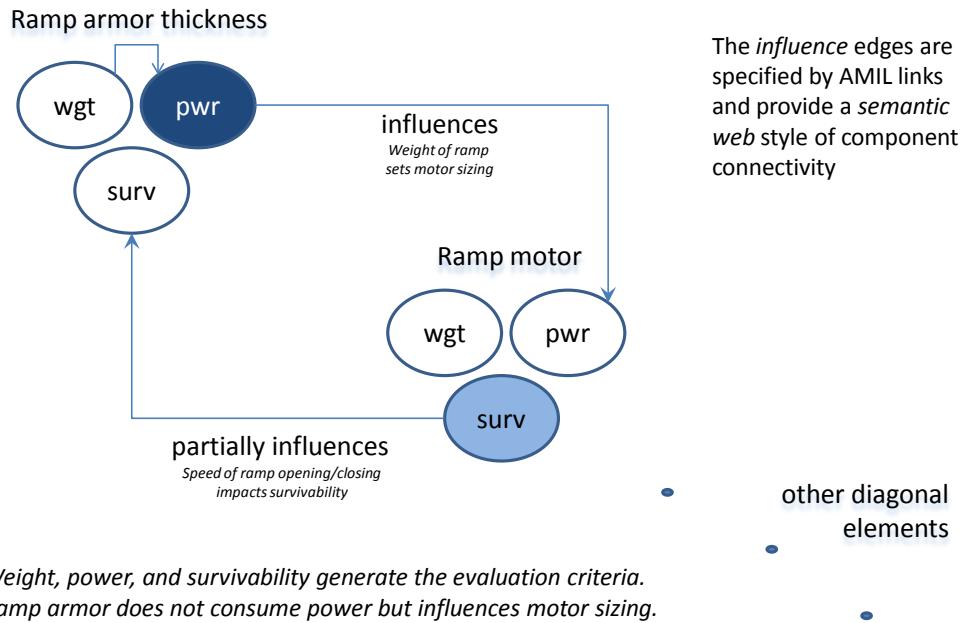
*Consider the situation of the popular “build-your-own-computer” from components. We can conceivably put any components together that we want (power supplies, CPUs, memories, drives) yet some of these will form constraint, incompatibility, or degradation relationships. For one specific criterion, we can consider the selection in terms of power draw, and then we put a*

*premium on low power consumption as one selection criteria. This is a typical influence relationship.*

*As another influence relationship, consider that by bundling a few of the components together one can get a huge discount in price – the parts wouldn't be connected yet they have an influence on each other. That would be an example of no direct data flow between the components.*

The key is that we will need lots of flexibilities in the rules, while a reasoner or expert system conducts the search. A graphical illustration of an influence diagram is shown in Figure 7.6-9. This particular example features the factors that go into designing a vehicle egress ramp.

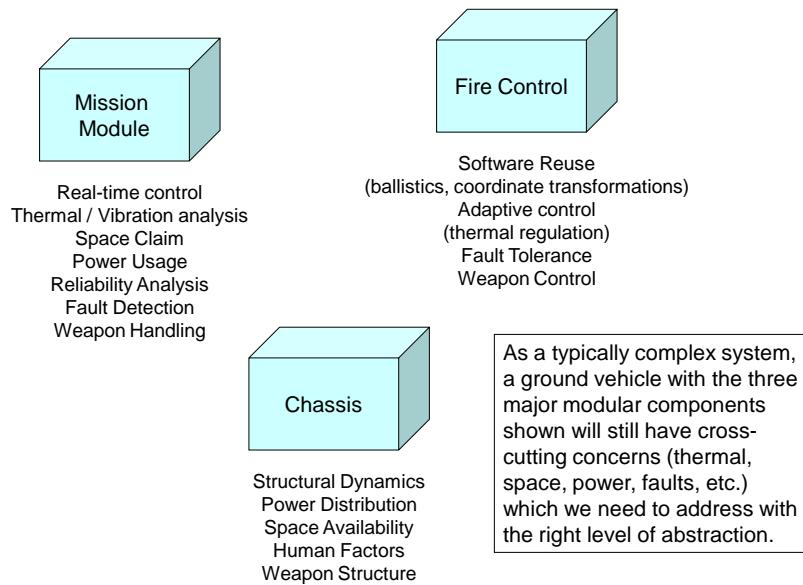
## Influence diagram and Design Structure Matrix



**Figure 7.6-9. Influence Diagrams are More Useful for Showing the Directed Interactions than a Design Structure Matrix**

Periodic application of DSM searches accelerates the development of set-based designs. Whenever possible the analysis of alternatives will weed out poor performers, and if done comprehensively is complete enough to help expose the problems of cross-cutting concerns and possible abstraction leakages. The tool ESKER (refer to 7.6.2.2) applies multi-objective criteria to utility functions for solving Analysis of Alternative (AoA) problems. In other words, we can use ESKER for down-selection and other optimization problems.

The DSM approach also supports the idea of partially decomposable systems, which is a divide and conquer strategy for partitioning a system (Figure 7.6-10).



**Figure 7.6-10. Partially Decomposable Systems**

A module is the definition as put forward by H. Simon to describe a decomposable unit. If a system design contains a number of components that appears nearly decomposable or partially decomposable, then we can try to group the components into modules. Or alternatively, we will partially decompose the system of components into modularized sets of components.

These modules form archetypes as well, in the sense of being close to the ideal and thus provide the design team a pattern or a set of alternative patterns from which to build. The process workflow then becomes one of selecting the components to go into that archetype. To meet a set-based engineering criteria, we can assume we always have alternatives for the components, say for example three alternatives per component, leading to the use of an analysis of alternatives combinatorial estimate:

$\langle A \rangle$  = average archetype influence-set size  
 (the number of interacting components per set)

$N_A$  = total number of archetypes for a vehicle

$\langle M \rangle$  = average number of alternatives per component

SearchSize =  $\langle M \rangle^{\langle A \rangle} \times N_A$

For a reasonable set of sample numbers,

$$\langle A \rangle = 10$$

$$N_A = 500$$

$$\langle M \rangle = 3$$

This draws from a pool of 15,000 components, which is just the three numbers multiplied together. Then SearchSize = 3 million on the first pass with a nominal default selection for each component.<sup>3</sup>

In terms of encoding the interaction knowledge, decomposition becomes a divide and conquer strategy which tries to minimize the number of interfaces and extracts commonly occurring global interfaces as design rules [BC00]. The design rules then provide constraints that reduce the complexity of search, trading off the potential for finding interesting combinations with the experiential knowledge of previous engineering archetypes. From the formulas above for combinatorial complexity, we can eventually determine that:

1. Increasing the number of choices per component increases DSE search time
2. Increasing the number of constituents per module increases DSE search time
3. Increase in efficiency from specialized reasoners is ideal (to reduce by exclusion factors 1 and 2)

In other words, the rules prevent the DSE from going down paths that yield little improvement. A knowledgebase format fits in well with this approach as the tacit knowledge is directly coded as *logical rules* of design.

A component-based DSM also specifies the physical interactions between elements in a cyber-physical system architecture. Different types of interactions can be displayed in the DSM and the types of interactions will vary from project to project. Some representative interaction types are shown in Figure 7.6-11.

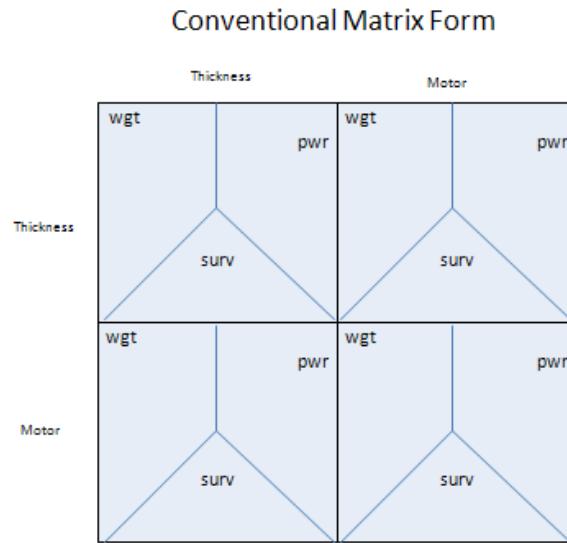
<b>Spatial</b>	needs for adjacency or orientation between two elements
<b>Energy</b>	needs for energy transfer/exchange between two elements
<b>Information</b>	needs for data or signal exchange between two elements
<b>Material</b>	needs for material exchange between two elements

**Figure 7.6-11. DSM Application Domains (from Pimler and Eppinger, 1994)**

In this figure, the spatial variant covers finite element analysis techniques for dynamic structures, and the matrix could become three-dimensional to match the actual physical structure. It also covers the computational arena known as Spatial Computing (covered in the section 7.6.4.1). The energy variant of the DSM covers models such as Hybrid Bond Graphs, where the transitions obey the laws of energy conservation. The information variant is Design Space Exploration (Figure 7.6-12).

---

<sup>3</sup> The Voyager spacecraft designed in the 1960's consisted of 166 different potential assemblies grouped into 51 functional families. When considering the different redundancy implementations, this resulted in  $10^{21}$  potential system configurations and its optimization obviously required some computer support. A good example is [LEH67] because the Voyager was probably the most durable system ever deployed.



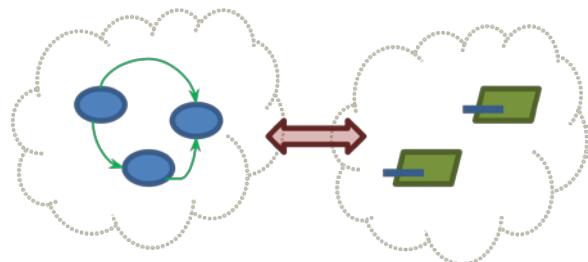
**Figure 7.6-12. Design Structure Matrix for Information - Utility Functions Trade-off Criteria for Different Design Choices**

The material domain covers compartment models (for fluids, etc) and also Markov chains (frequently used for reliability analysis). Again these are all graphs and the universal computational goal in each of these seeks to reduce the time searching for a solution given the physical constraints and laws.

The idea that these nodes often interact with near-neighbors and therefore may not need extensive interconnection implies that *local computation* schemes might prove useful. A synergy exists between all these flavors of local computation, and this is the idea behind the *generic inferencing* architecture described by Pouly and Kohlas [PK11]. To speed up the computations, parallel and distributed processes schemes can be applied, while algorithmically, the properties of valuation algebras and the application of join trees can further reduce the computational load. This is further explored in Spatial Computing (refer to section 7.6.4.2).

#### 7.6.1.2.7 Models of Computation and Communication

When we move from analysis to simulation, different considerations come into existence. In general, the co-simulation models require that we pay much more attention to interface details since they will reside in a heterogeneous environment with potentially cross-cutting multi-physics domains.



**Figure 7.6-13. Heterogeneous Interfaces**

The computations can be heterogeneous in terms of behavior content as shown in Figure 7.6-13 or the communication channels may require different operational time-synchronization strategies (refer to section 7.6.3.1). In META parlance, Models of Computation and Communication (MOCC) describe the general strategy that one needs to adopt.

#### 7.6.1.2.8 Contracts and Assume/Guarantee

Contracts (and the associate assume/guarantee strategy) are generally defined as rules in the form of pre-conditions and post-conditions applied to some design functionality, either through intended use or upon execution [QGP10].

Contracts = controls the use in terms of fitness for purpose

Assertions = strong controls for compile time or run-time guarantee (software in particular)

Guards = adaptive controls for run-time usage

In the contract world, we use co-analysis models and apply co-simulation to show that a particular design works correctly. This assumes in the best case that we can reach a top-level guarantee of 100% correctness. Allowing a probability spread in the model's parameters (i.e. manufacturing tolerances, etc) or environmental conditions (i.e. temperature, vibration, etc) will drop the Probabilistic Certificate of Correctness (PCC) to something below 100%.

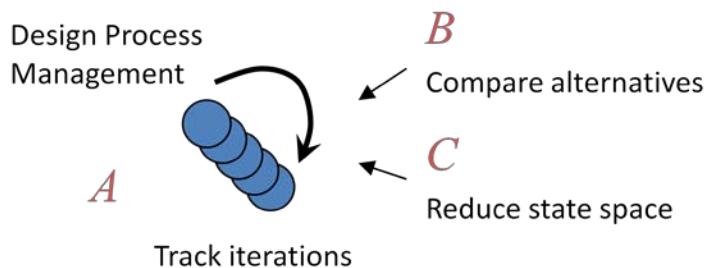
The bottom-line is that if we want to say that we have a PCC of 100%, we require a strong condition of an assume-guarantee on that particular model. The assume part is that all inputs are within range and the environment is within operational bounds. The guarantee is that it will work correctly over that set of ranges.

We thus incorporate an assume-guarantee strategy when we evaluate PCC on our co-simulations (refer to 7.6.3.5).

#### 7.6.1.2.9 Set-Based Concurrent Engineering

Also known as the “Toyota Paradox”, the concept of set-based concurrent engineering involves a strategy of keeping open design alternatives for as long as feasible [MAP09] through the development process. This deferral of commitment to a final design allows the possibility of future design adaptability, which becomes a powerful form of design look-ahead.

We apply the set-based approach via tools such as ESKER and ECTo, which reason on open alternatives and then reduce the complexity of a large state by applying design rules and constraints as needed (refer to Figure 7.6-14).



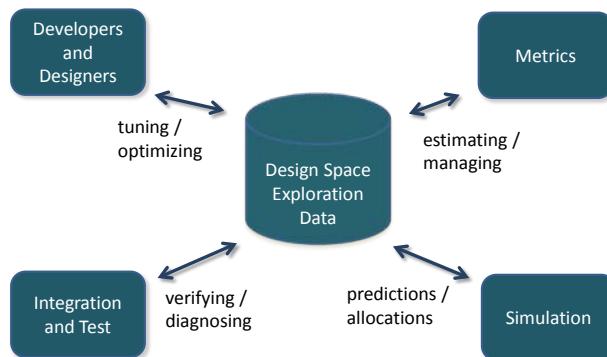
**Figure 7.6-14. Set-Based Concurrent Engineering Continuously Tracks Open Alternatives While Culling Out Poor Design Choices**

Set-based concurrent engineering is only a paradox in that it works against intuition that not committing to a design early is the best possible strategy. Maintaining a robust and reflective process flow through the DSE tools, DSM analysis, and Test & Verification PCC evaluations though-out the ARRoW cycle is crucial to making the set-based approach work.

#### 7.6.1.2.10 Workflow Provenance

Repeatability and regression of simulations is required to allow for reuse and adaptability. It can often become a full-time task to maintain reasoners and their application as the design and data environment matures. The process and composable workflow reasoners we have evaluated will provide a provenance capability. In the traditional definition, “provenance” means to possess knowledge of the origin or history of some object, and the current technology definition of provenance is to use automation to guide the process and thus make the history repeatable. This process could then become repeatable or form a regression test for the usage of tools and data sources as shown in Figure 7.6-15.

### Design Space Exploration data



**Figure 7.6-15. A Provenance Strategy Manages Development Processes that Require Multiple Steps**

One specific approach that we highlight uses the process-based OWL-S semantics to attach automated services to particular workflow tasks [YGD10].

#### 7.6.1.2.11 Test-Driven Development

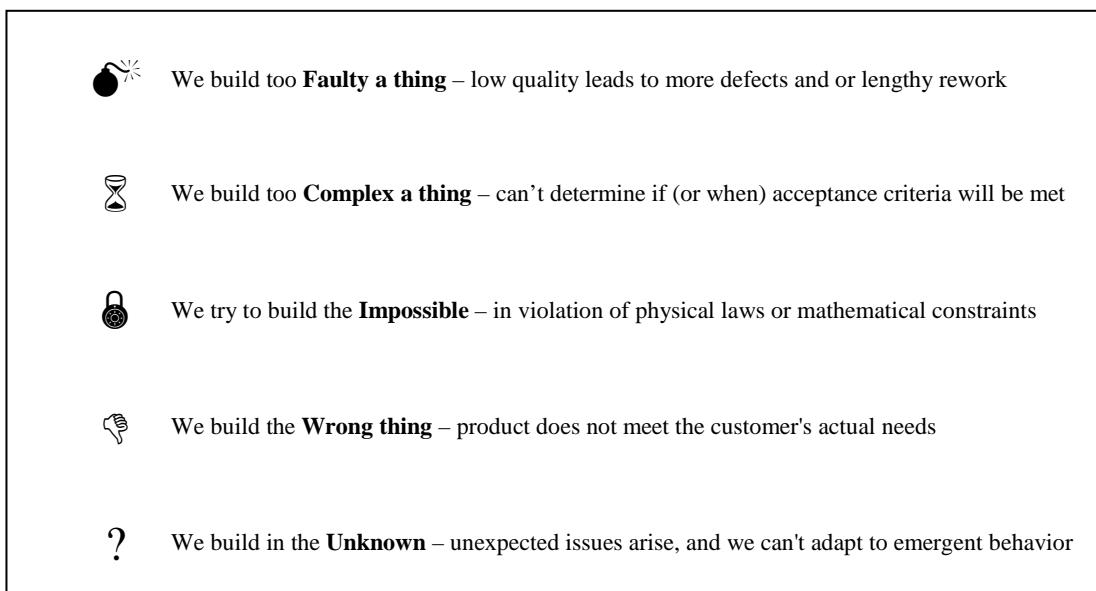
Test-Driven Development (TDD) is an agile process geared for software development, but when placed in a model-based engineering environment it can provide value during the entire cyber-physical system life-cycle.

The key to applying TDD successfully is to start integrating it into the development process at the earliest possible opportunity. This typically corresponds to verifying the first available instance of an operating simulation of the system (or on a concrete or abstract representation of the system). Instituting qualitative (smoke) or quantitative tests here enables the development team to quickly respond to regression failures caused by early design decisions that do not emerge until higher fidelity representations become available. The regression failures often come about due to leaky abstractions not accounted for by the design or model. It essentially

mitigates the risk of not having a “correct by design” system, as the later regression errors have a much reduced chance of occurring.

We assert that this extra level of testing does not impact the development timeline since the test plans and test cases get executed in parallel with the conventional modeling development. In other words, these do not exist along the critical path but ride alongside it. The benefit accrues with time as the model becomes more robust instead of more brittle as the model representation starts to grow in scale. The tests only act as the “driver” of the development because they provide warning signs if and when process veers off course. Experience has shown that identifying defects as early as possible saves time and money in the long run.

The comprehensive TDD approach also aids in the continuous verification of requirements. The key in this regard is to match use cases (i.e. derived software and system —requirements) against test cases. A simple but effective approach is to write use cases with the attitude that anyone can also read them as test cases. The test engineer then does not have to spend time translating use cases into tests; thus, we can further mitigate the risk of testing from bottlenecking the critical process path.



**Figure 7.6-16. Sources of Defects that TDD Aims to Mitigate**

This rigorous attitude to system verification needs to be coupled to a state-of-the-practice automated software testing infrastructure. Automation becomes necessary because we typically leverage virtual simulations of full vehicle behavior and the scope of the tests quickly scales to make any manual testing out of the question. A comprehensive test infrastructure should accommodate the automated launching, monitoring, and test data collection for dozens of applications running concurrently within a distributed environment.<sup>4</sup>

For component-based model development, the critical aspect is to include acceptance tests for each of the individual components and configuration manage these items as testware. The

---

<sup>4</sup> Historically, the team has used the same declarative mechanisms to specify our distributed launch and control as we do in specifying our simulation data and runtime configuration.

component test cases developed early in the life cycle get reused in future stages of the life-cycle, such as during product-line integration. Thus, a finished project delivers two products, the software and the testware. In META demonstrations, we illustrated how early test promotes quality in both verification and isolating leaky abstractions. If we have appropriate acceptance tests, the likelihood of encountering leaky abstractions decreases because the client understands the context for the use of the components from reading the acceptance tests.

The criteria needed for objectively (verification) and subjectively (validation) evaluating the model's representation include: non-ambiguity, verifiability, consistency, modifiability/adaptability, traceability, presentability, and completeness. Testing becomes a full life cycle process that initiates when the project begins to achieve maximal effectiveness as the customer can visualize results immediately. The test-driven development strategy applies to each of the phases in the ARRoW process from conceiving to final test.

Although each of these phases accomplishes different objectives and may use different development teams, the underlying test and verification environment remains a constant. In practice, for each phase of the development branch, there is a concurrent corresponding activity on the testing branch. Putting in place automated tests in the beginning of the product development life-cycle allows us to accelerate the process via concurrent activities, helping to achieve the  $5\times$  speedup desired. Any volatility in requirements is handled by adapting and refactoring of software in the virtual environment, where change is timely, cost-effective and robust with the help of regression and other tests.

#### **7.6.1.2.12 Parallel and Distributed Development**

The ARRoW development strategy strongly recommends concurrent development to achieve a  $5\times$  speedup, especially when considered in concert with and augmentation to robust and proactive improvements in requirements and system conceptualization.



**Figure 7.6-17. Parallel Development Process Speeds Development**

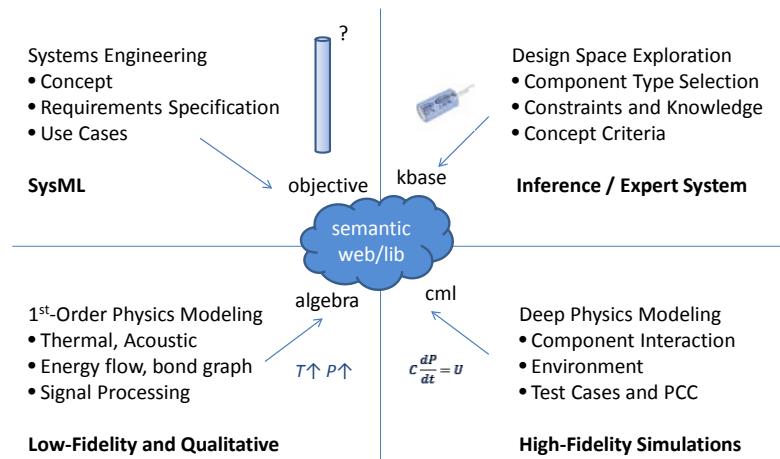
Contention issues can slow development for as simple a reason as bottlenecked database access (refer to Final Report 1A and BBN metrics section). The reasoners employed allow for concurrent development as the knowledge is contained on the developers local memory which precludes contention problems. Other bottleneck issues are described in the META 1a report.

#### **7.6.1.2.13 Fidelity and Abstraction Levels**

Simulations used for verification necessarily require details that reflect the properties of the as-built product. In general, high-levels of abstraction go with lower fidelity. So depth of fidelity becomes a consideration for the abstraction level chosen. As earlier we showed how

compartmentalization can be used in analysis so to can it be useful for simulations. Co-simulation will require clever compartmentalization to make the high-fidelity computation solutions tractable for the deepest multi-physics problems.

## RLC Circuit applied to ARROW



**Figure 7.6-18. Abstraction Levels in a Multi-Physics Domain Example**

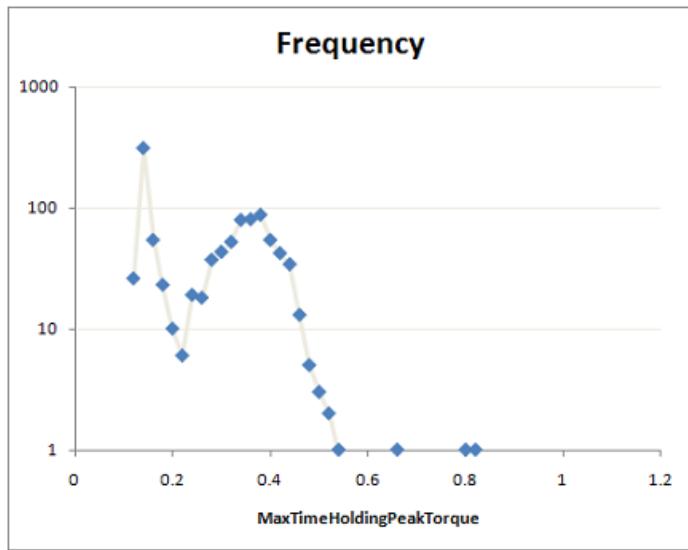
Abstractions applied to the compartments allow potential for reuse for a semantic CML classification scheme (refer to Figure 7.6-18).

### 7.6.1.2.14 Sampling and Verification

For PCC calculations, the time it takes to sufficiently verify test state-spaces can often be prohibitive. We incorporated several importance sampling [MD01] techniques and applied the ESKER tool to explore the test-case state space and record potential failure points.

This is best illustrated by an example, whereby a PCC calculation is populated with failure scenarios via test case generation and then importance sampled for a probability measure.

Figure 7.6-19 demonstrates the sampling approach we used on a ground vehicle ramp design challenge problem. The PCC calculation on the results histogram detects possible elevated drive burnout incidence from frequent sustained high torque requirements. This was caused by soldiers exiting vehicle while the ramp is in motion. Identifying the worst cases provided an improved basis for driving redesign.



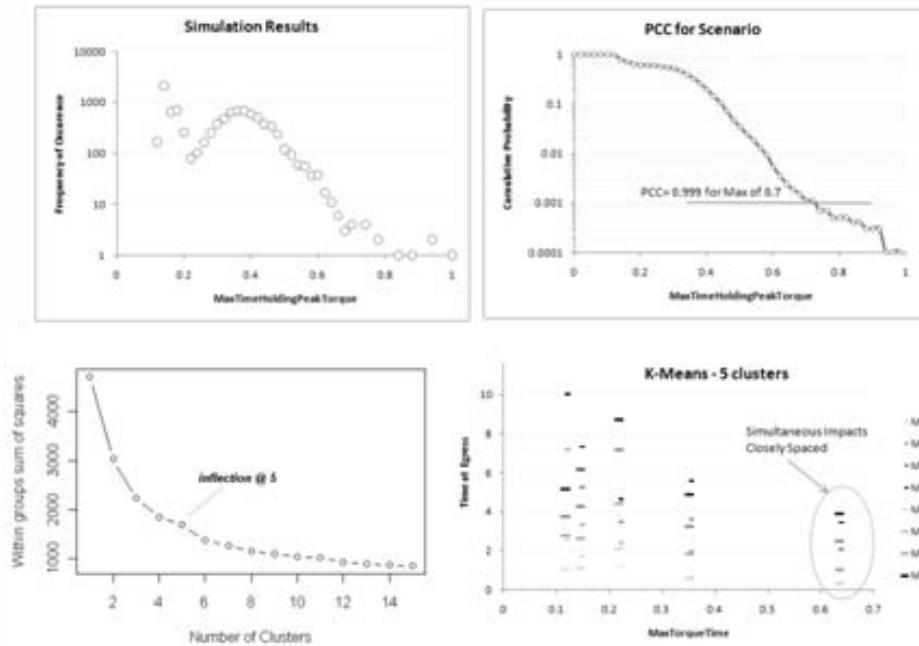
**Figure 7.6-19. Test Space Sampling Generates a Histogram for PCC Evaluation**

#### 7.6.1.2.15 Diagnostics

As the preceding section pointed out, the most wide-open part of the reasoning strategy involves the incorporation of diagnostic aids. Since many of these rely on a variety of statistical tools which are fairly mature, we will defer to the Metrics Dashboard section for how these can be plugged into the AIDE environment.

To work the vehicle ramp problem diagnostics, feedback on the PCC results were fed into a clustering classification method (K-Means) which uncovered a qualitative failure mode. The PCC calculation thus detected assumptions leading to failure and it provides an input to redesign. Figure 7.6-20 suggests a storyboard of the diagnostic aid used in a demonstration.

## PCC and Diagnostics



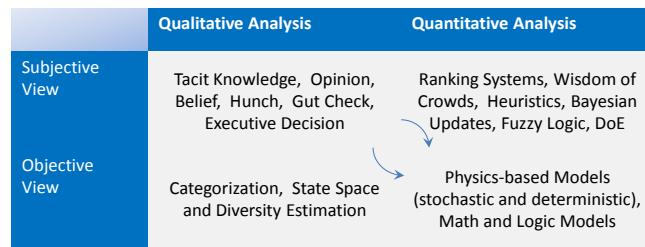
**Figure 7.6-20. Diagnostic Aids Used to Discover a Design Problem**

An iterative design is the preferred approach for complex systems and an agile diagnosis and learning/redesign is crucial. Diagnosis provides a means to get value from inadequate PCC scores, for example by climbing the PCC slope in design space. Or in the case of acceptable PCCs, by progressively relaxing assumptions to detect most plausible remaining failure scenarios.

### 7.6.1.2.16 Probabilistic Context

The original intent for the ARRoW process was to include qualitative simulation into the development tool space. We extended this to not only include qualitative and quantitative evaluation through the DSE reasoners, but include stochastic elements as well. These stochastic elements could include subjective belief information as well as objective probabilities derived from empirical observations and system models.

Figure 7.6-21 shows the elements of subjectivity and objectivity as applied to a migration from co-analysis to co-simulation. The co-simulation will eventually require stochastic models to evaluate a PCC, which is the basic premise that a PCC will need a probability as a pre-condition in order to propagate uncertainty.



**Figure 7.6-21. Paths to Stochastic Formal Verification – Stochastic Elements Starting from Co-analysis to Co-simulation**

#### 7.6.1.2.17 Crowd Sourcing

The eventual strategy is to encourage crowd-sourcing at the level that significant progress can be made. In the metrics report, we developed methods for monitoring development progress based on empirical observations of algorithm development (refer to the MathWorks contest described on page 64 of the Phase 1a final report [META11]). Progress was measured by tracking performance and accuracy improvements in algorithm development, accomplished by automated evaluation and recording of submissions. When used for parallel directed problem solving, this will have benefits, but the key is to measure the asymptotic trends as the law of diminishing returns will set in.

Crowd-sourcing improvements based on spontaneous creativity will always exist and the use of archetypes as templates, placeholders, and alternative architectures will help spur creativity. Individual improvements will come from the use of ontological search mechanisms and performance enhancers such as search caching and parallel computing.

#### 7.6.1.3 Engineering Domains

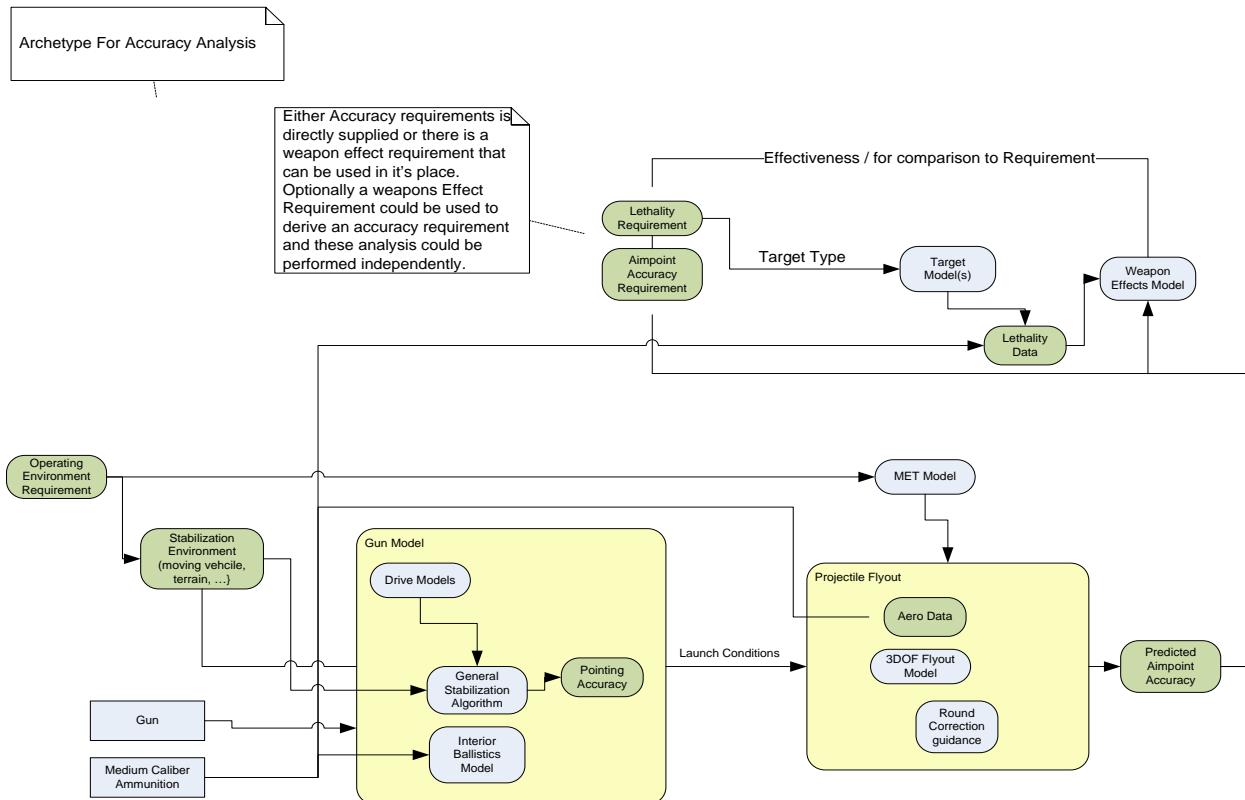
The specific areas that we chose to prototype reasoners include workflows, DSE and conceiving, synthesis, and testing. As an example, some of the domains we can consider include:

- Component selection (use alternatives to evaluate different combinations)
- Test evaluation (use ranges to sweep through tests like we sweep through alternatives)
- Diagnostics (use declarative semantics to isolate problems, working either backwards or forewords)
- Configuring simulations (mix and match alternative fidelity models and essentially stitch together a virtual simulation that is executable)

#### 7.6.1.3.1 Analysis Process and Workflows

The goal is to speed up the development process by providing data-driven automation, reusable components and behaviors, and templates for typical designs. The specific objective is to readily create reference architectures and archetypes that could serve as templates for design and development activities. The templates provide hooks for various possible implementations and components according to the underlying AMIL graph of relations and rules.

Figure 7.6-22 illustrates an analytical workflow required for calculating a fly-out trajectory of a projectile. The analysis steps are standard enough that an archetypal workflow can be converted into a template and a reasoner can suggest certain compositions and ultimately guide the execution itself.



**Figure 7.6-22. Graph of Analysis Archetype for Projectile Fly-Out Simulation**

As another example, the steps in verifying a fault-tolerant design show how to unify PCC, contracts (assume/guarantee), AMIL representation, design synthesis/ composition, and reach set or envisionment (qualitative in the sense that we have either a working element or a failed element). We want to demonstrate these steps:

1. Show that a design has a certain PCC given the context of the real-world
2. Demonstrate how we can apply assume/guarantee
3. Express the problem in terms of an AMIL graph
4. Demonstrate a level of synthesis in either an analysis model or design model
5. Execute look-ahead or exploring failure possibilities implied by reach-sets or an envisionment

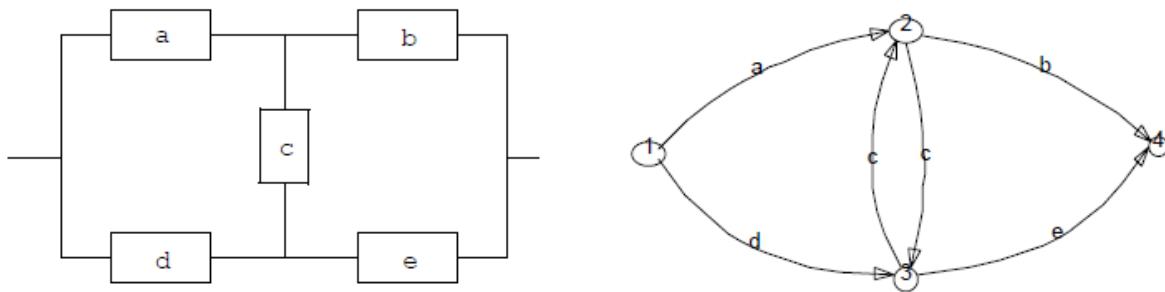
To do #1 and #2 in one pass we only need to convince ourselves that computing a PCC is dependent on having some level of guarantee on every component of the system. The *guarantee* is that the PCC will be 100% *assuming* that each component meets each of its pre-conditions. The fact that all pre-conditions are not met due to contextual externalities will drop the PCC below 100%. This salient point provides the connection between PCC and the

assume/guarantee model. We just need to have components that can be modeled as having less than 100% guarantee given the right circumstances.

We also consider points #3 and #4 in combination. As we can reason on AMIL data at the most essential “triple-store” graph level and then generate a design model (given some extra rules outside of AMIL), means that we can extend what we are doing with design space exploration. But the next step is to show how we can do a PCC from that generated design. This leads to step #5, which is a way of generating test-cases, which can also come out of a synthesis domain.

The canonical workflow example that we can test this unifying theory against is that of a complex fault-tolerant configuration.

Say that we have a requirement that a subsystem has two distinct operations which have to run in conjunction and provisions must be made to handle the case of either of the two operational components failing. This can be specified as a graph archetype shown in Figure 7.6-23.



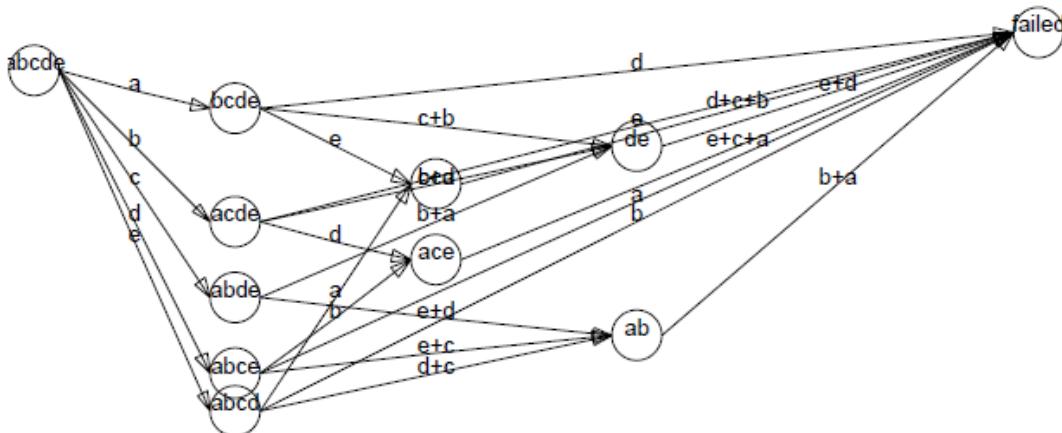
**Figure 7.6-23. Archetypes for Analyzing a Derived Fault-Tolerant Reliability Requirement**

Here the top and bottom paths are essentially redundant paths and the path labeled “c” is for switching over on failure. Each one of the blocks has some assume/guarantee in terms of a reliability value.

Based on this information, we can easily express this example as an AMIL graph (the diagram to the right) and then add some rules to enable generation of an analysis model suitable for calculating a PCC.

We can synthesize the success paths first, such as  $a \rightarrow b$ ,  $d \rightarrow e$ ,  $a \rightarrow c \rightarrow e$ , and  $d \rightarrow c \rightarrow b$  and the failure path as a break in the chain from  $1 \rightarrow 4$ .

From this point we can apply further synthesis rules, and below is the solution graph, which is an “envisionment” of all possible failures.



**Figure 7.6-24. Auto-Generated Expansion of Reliability Block Diagram**

The left-most state is the initial configuration of all components operational and the state at the end is the situation of a critical failure. We can submit this graph to a Markov solver, executing it automatically via AMIL, and get a PCC assuming nominal and independent failure rates of the components. The nominal failure rates come from AMIL relations of those components retrieved from the CML, and we can compute metrics on the complexity of the representation<sup>5</sup>.

This becomes a complete and archetypal example of soup-to-nuts verification of a design. Through automated workflow as driven by a reasoner engine this provides an excellent example for the generic ARRoW process<sup>6</sup>.

For reasoning about a detailed cyber-physical design which incorporates temporal and probabilistic reach-set analysis refer to the Appendix on the Reactive Model-based Programming Language (RMPL section 7.9). This allows solving analysis problems at an abstraction level closer to that of plant and controller, yet is complementary to the logical planning reasoners and state space expansion approaches outlined here.<sup>7</sup>

### 7.6.1.3.2 Design Space Exploration and Optimization

To understand how reasoning within the context of a knowledgebase works in general, let's take an example from the classic concept of automated Decision Support Systems (DSS). Although DSS will never be completely automated due to the human element required, tools can help intuition and to predict results, and then use the real results to confirm educated guesses. As a result, decision makers use results from the models to develop the next step in the thought process so as to gain a deeper insight into the problem.

*Decision Support System: A model-based set of procedures for processing data and judgments to assist a manager in his decision-making.*

Construction of models thus becomes an extension of the decision maker's ability to think about and analyze problems, and not as a replacement of these analytical skills.

<sup>5</sup> This approach can quickly scale into a large graph

<sup>6</sup> This demonstrates the qualitative aspects exemplified via "working" versus "failed" states.

<sup>7</sup> For example see the REST Modeling language described in [PRP94].

Generally a DSS is defined in terms of three interacting components: a language system, a knowledge system, and a problem-processing system. The language system, in this case AMIL (which can contain a GUI such as those implemented for AIDE and a visualization graph) allows the user of a DSS to interact with the other two components of the support system. The knowledge component contains the declarative knowledge contributed by the decision-maker or domain expert. This element has become identified as the *Knowledge-Based Engineering System* (KBS) and it really becomes a larger semantic web of information, which will include an interface to CML meta-information. The connective problem-processing system represents the communication channel between the language system and the knowledge system, referred to as an *inference engine*. This becomes the set of reasoners referred to as GEAR. One can take the operational DSS definition another step by considering the data separately from the rules and analysis models to help structure the knowledge. This merges into an ontological scheme for data classification and storage.<sup>8</sup>

A guide to the process of decision-making is outlined in the following steps:

1. **Elicitation.** Analysis of the decision area to discover applicable elements
2. **Analysis.** Location or creation of criteria for evaluation
3. **Knowledge Engineering.** Appraisal of the known information pertinent to the applicable elements and correction for bias
4. **Scoping.** External connectivity and isolation of the unknown factors.
5. **Tuning.** Weighting of the pertinent elements, known and unknown, as to relative importance
6. **Effectiveness Optimization.** Projection of the relative impacts on the objective
7. **Evaluation.** Synthesize into a course of action.

The key is to get to Step 3 as quickly as possible. Unlike a typical software development effort, which this process looks like on the surface, we should start formulating rules and constraints quickly rather than waiting for a complete specification.

Elicitation (**Step 1**) needs the support of all the stakeholders, so it is best to work this out in a group or crowd-sourced setting. This should take a single session to decide what to cover among power, weight, reliability, cost, etc.

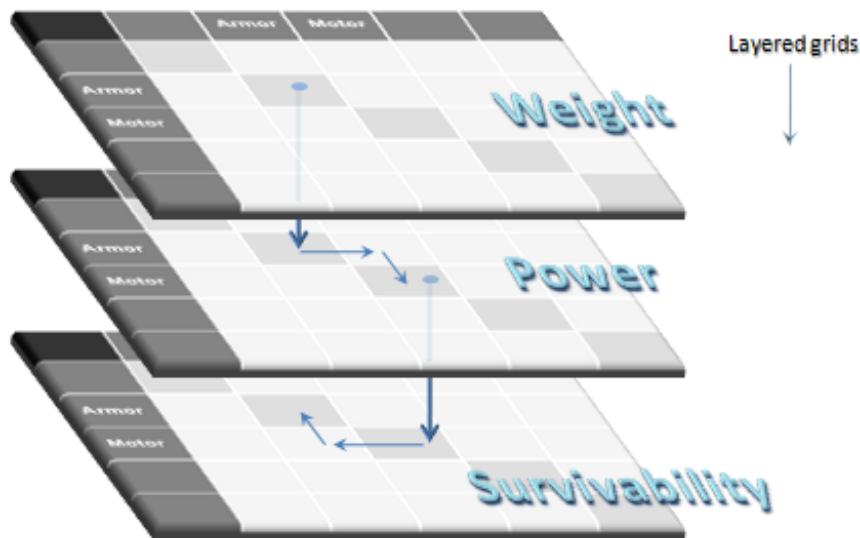
Analysis (**Step 2**) is dependent on the availability of sources of information. This can become a bottleneck if complete information is not available, so we can use heuristics and placeholders for the elements in the elicitation step. For example, external model development can be handled here, and placeholders used for these until the external nodes are scoped into Step 4.

Knowledge engineering (**Step 3**) is where the subject matter expert and coder get together and declare their knowledge in terms of a set of facts and rules and incorporate these as a DSM (refer to Figure 7.6-25). This is routinely a massive undertaking for developing a product the

<sup>8</sup> DSSs tend to be defined in terms of the system needs and the way that they will be used. The definition may not explicitly incorporate the different components of a DSS (KBS, graph database, and GUI). Instead, the definition used is more general to include flexibility during the creation of the candidate system. A spreadsheet solution could have everything rolled into one file, for example. Furthermore, a DSS does not have to generate the complete decision, just enable the support for the final decision-making process. This is in keeping with the practice of semi-structured decision making.

size of a vehicle but progress can be measured via the usual metric of rule count. When the rule count starts leveling off, the rule of diminishing returns starts to set in.

## Influences between DSM Layers



**Figure 7.6-25. Design Structure Matrix Interactions Needed for Knowledge Engineering**

Scoping (**Step 4**) admits to the fact that we will never complete a comprehensive knowledgebase<sup>9</sup>, and that remaining subjective and objective criteria are either coupled into the knowledgebase through external models or left blank as subjective placeholders and simple trade-off heuristics. Constraining the set of allowable alternatives is crucial at this point to allow optimization to take place in the next step.

Tuning (**Step 5**) is where we set up optimization criteria. This can be accomplished concurrently with Step 3 so that we always have quantitative results to show as the knowledgebase matures. This is the latest point at which we can add a GUI before handing the expert system to the user for evaluation.

Effectiveness Optimization (**Step 6**) is where the user can get involved with the evaluation. At this point we should have enough sensitivity analysis and optimization algorithms in place so that the user can execute queries without having to do any programming.

Evaluation (**Step 7**) is where we formally include the results of the optimization into the bigger decision support picture. For example, the results together with a narrative description of the knowledgebase can be included to justify a decision.

### 7.6.1.3.3 KBE Template Design Synthesis

Complete synthesis of systems is a challenging proposition. The process toward that goal is one of reusing domain knowledge and design rules so we can chip away at the feasible parts. As a working definition:

“KBE templates are intelligent documents or features that aim at storing know-how and facilitate its reuse.”

The knowledge is built up in terms of the usual semantic web triplet of *(subject, predicate, object)*. If a predicate is “*isPartOf*” or “*partOfIs*”, depending on the relationship between the subject and object, then it becomes just a natural part of the organization of a component model library. For an interface to a component that has empty slots on it, we need to describe the placeholders for other abstract interfaces. A *ramp* will need a slot for a *hinge*, for example.

*(ramp, partOfIs, hinge)*

The same goes for “*parameter*”, which is a predicate for some subject that will either go to a named object or some property.

*(ramp, parameter, material\_stee)*

Drawing from CML, the approach is to fill in and instantiate the pattern matches.

- In the CML the *partOfIs* and *parameter* are left open or define types as part of an archetype. It is a pure reusable utility function arising from the ontological classification.
- In the Master Model the *partOfIs* and *parameter* are filled in. In many models, such as a Pro/E tree structure you won’t even see this but it does exist, at least implicitly.

In a Design Space Exploration, the connectivity is defined by the rules and any declarative semantics that we can pull out of the knowledgebase.

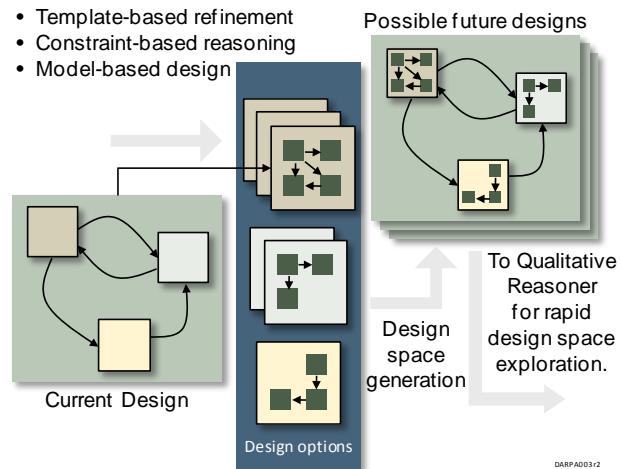
Extending this idea, KBE templates can contain suggestion rules. As examples consider the following triples:

**A contains B**

and

**A canContain B**

The first is an imperative and says that is the way it is, part of a fixed ontology. The second one is closer to the spirit of a design rule hint where someone has established that the two can go together but that it is not necessarily fixed. This *canContain* relation will also have all sorts of contingencies based on **A** and **B**’s specific properties, which points toward supplemental rules.



**Figure 7.6-26. ARRoW’s Template-Based Look-Ahead Concept**

Once composed these relationships would show a definite connectivity within the master model.

When PTC describes templates in Pro/E, or Dassault with CATIA, this kind of reasoning takes place, but significant amounts of clarity emerge when we tie the KBE template approach into ideas such as ontological organization, description logic, set-based engineering concepts, and design space exploration algorithms.

Combining ontological classification with design rules makes the synthesis more solid and maintainable. The maintenance issues are addressed by the ontological organization and description logic. The freedom from a strict design policy is addressed by a set-based methodology and the multi-objective optimization iterations and DSE that we systematically apply via the ARRoW process.

One of the reasons why EDA has worked so well in the commercial industry is that the ontology was actually *built-in* to the libraries. If someone needed an **OR** gate, they would know exactly what to look for, they would just specify an **OR** gate. That is not the case with general engineering design, where the ontological classification can help immensely to navigate through the variety of design choices.

#### **7.6.1.3.4 Requirements-driven Test Archetypes**

We consider several different reasoning strategies:

1. Controlled natural language conversion of requirements to tests
2. Requirements analysis archetypes
3. Verb-based tests, drawing from requirements
4. Automated testing scaffolds, such as with Maven and Phoenix Integration's ModelCenter.

The first two cases are discussed in section 7.1. Decomposing requirements is in general a difficult problem because of subjectivity and the difficulty in exposing intent or the original requirements team. The latter strategies are more amenable to automation in that test cases are often more explicit and objective.

Test cases in the ideal situation are a set of pre-conditions followed by an expected set of post-conditions. If we set up rules to map the pre-conditions, which could be environmental or parametric input, against the variables and sensor inputs found in a master model, then we could definitely use a reasoner to find potential matches. The same would be needed for the post-condition side, where the mapping would be between the expected value and the appropriate sensor output of the model.

So tests become object-predicate-subject triples on the input side: *object-stimulates-subject*; and then a kind of complement to this on the output side: *subject-provides-object*. The predicates become test-commands or verbs that often derive from the requirements vocabulary. For example, if a requirement had an active-tense than the verb describing this tense would become part of the test vocabulary as well.

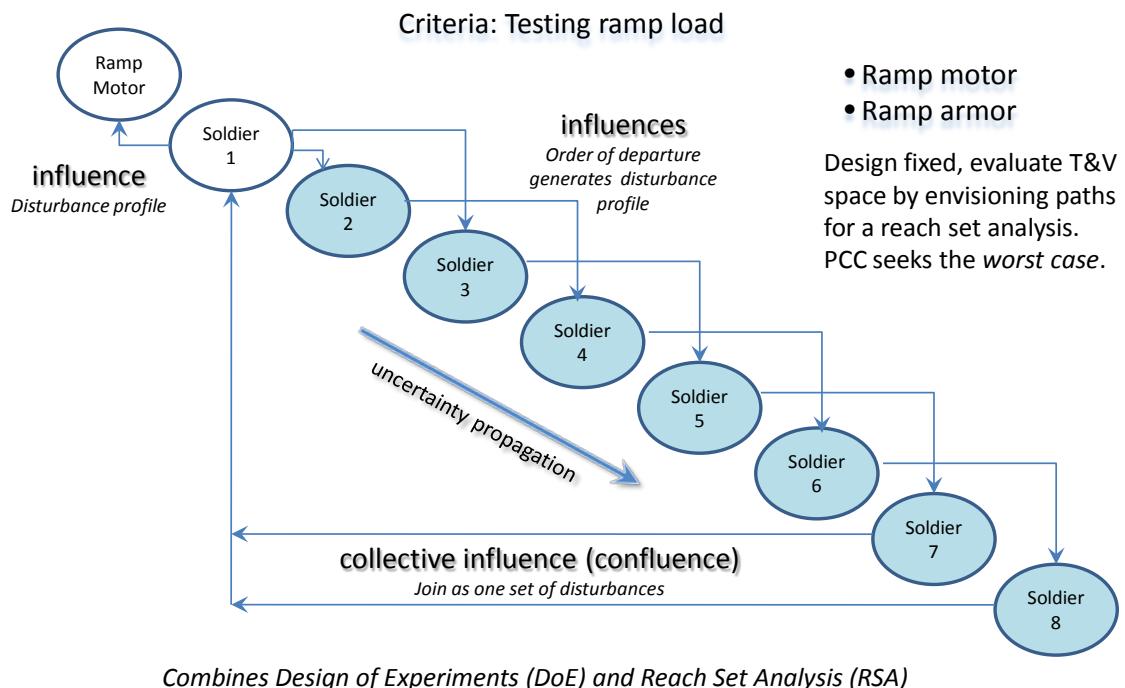
As finding the right verb for the test situation is time-consuming without automated aids, the archetypal rules would entail searching through the master model and the knowledgebase to find pieces that match. The initial corpus would be the entire knowledgebase of components, test-cases, and context models. This space would get reduced as the state of the design matures.

### 7.6.1.3.5 Test Space Exploration for PCC

The flip side of design space exploration is test space exploration. The number of design choices a development team faces will only be exceeded by the combinatorial number of contexts that a tester will have to evaluate. Many approaches, ranging from the practice of Design of Experiments, to usage based models can be applied here.

An example we have used for demonstrating test-space exploration for Test and Verify (T&V) is shown in Figure 7.6-27.

## Influence Diagram and Qualitative State Plan



**Figure 7.6-27. Test Space Exploration Uses Similar Combinatorial Techniques to Search the Test Space as Used for DSE**

The ARRoW process will always have available commercial and open-source tools for testing.

### 7.6.1.4 Application of Reasoning Languages

#### 7.6.1.4.1 AMIL

ARRoW Model Interconnection Language (AMIL) represents relationships between model elements across heterogeneous models. Like the triple store, it is a foundational capability designed for growth and extension.

As a language that describes the connections between models, AMIL can be used as the basic structure for building a co-analysis/co-simulation model or master model, as per the “3-view” diagram in Figure 7.6-1. At the most fundamental level AMIL will support analysis and simulation of a partially or fully dynamic set of interconnected models.

**Data transfer.** For a master model, we do not necessarily need active nodes, because the interconnections can be made descriptive and declarative, and we may not need the data to be actively transferred in a static representation. However, the analysis and will need data exchange. Analysis may require less data throughput because we may only need to evaluate discrete modes of the system, such as for design space exploration. Simulation, and specifically co-simulation, will often require continuous flow of data between nodes.

AMIL is capable of supporting both of these situations, although in different ways.

For **co-analysis**, AMIL active nodes can be used to represent relationships between model elements across heterogeneous models. The active nodes will re-evaluate depending on the latest set of dependencies. As the complexity of the problem grows, AMIL can be augmented with mechanisms, such as process workflows, to assist in dealing with scale. For example, there could be circular dependencies that will make un-orchestrated or un-choreographed updates a challenging proposition. ESKER and ECTo have both been used to orchestrate the evaluation of AMIL nodes.

For **co-simulation**, AMIL could be used as a routing table information provider. Thus, AMIL will not partake in the actual data flow but it will configure the communications links. AMIL can model the adaptors required for the tagged signal semantics of heterogeneous co-simulation. So AMIL acts as a static routing and interconnect configurator and helps with the choreography of a co-simulation.

**Process Workflow.** Modeling the process is important for analysis and simulation because this will guide the flow of data and of the executing process. A workflow model is thus needed to avoid the race conditions and recursion problems that could afflict the AMIL active nodes. The solution for workflow lies in the analysis archetypes that describe process behavior, exemplified by the composable workflows that we have prototyped. These can be knowledge-based in the spirit of ESKER, the crowd-sourcing tools can include off-the-shelf approaches such as VisTrails and Kepler or commercial tools such as ModelCenter.

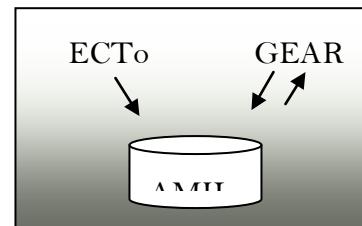
**Data Structures.** AMIL is essentially a graph and a graph can describe a variety of data structures. AMIL's Neo4J data store is actually more powerful than a triple store, so it can easily accommodate Semantic Web type data structures and any kind of directed graph. A graph data visualizer in place for AMIL helps to navigate the structure.

**Persistent Storage.** When we use AMIL it operates directly with a persistent data store. This is useful for both the development of analysis models and even more so for a master model. Tools will be required to handle version control and controlled access.

**Demonstration of Ontological Reasoning.** As a demonstration of the META language toolset, we want to apply a customized archetypal reasoner (GEAR) to help with the down-select from a set of ECTo design alternatives using AMIL as the knowledge store.

The pre-condition is that the reasoner needs to know what part of the conceptual model to filter on. The user of the ECTo model facilitates this by tagging those elements with a "DESIGN\_SET" property. This then gets saved into the current graph database as additional triples to the individual elements. The "DESIGN\_SET" becomes a property of its parent.

The reasoner next accesses the graph database to get all the relevant information needed to make a decision. It does this by making an HTTP request to the `ArrowWebServices` entry called `/arrow/arrowGraphData`. The response to this call is a JSON text string that contains the representation of the internal triple-store data (with the active content executed).



**Figure 7.6-28. AMIL as a Knowledge Store**

The JSON is parsed and then stored in the GEAR reasoner's local knowledgebase as an equivalent set of triples (the active content has been executed so no loss of information occurs at this point<sup>10</sup>). The predicate logic for doing this is in the code snippet (Figure 7.6-29).

```

process_triples(_, []).
process_triples(Subject, [(Pred=Obj)|Rest]) :-
    assertz(triple(Subject,Pred,Obj)),
    process_triples(Subject, Rest).

clean_triples :- retractall(triple(_,_,_)).

scan_amil([]).
scan_amil([json([A,(B=json(List))]) | Rest]) :-
    process_triples(B, [A|List]),
    scan_amil(Rest).

scan_amil([First | Rest]) :- %% If we dont understand discard
    write(user_error, First), nl(user_error),
    scan_amil(Rest).

scan_amil(_) :- write(user_error, 'Not in AMIL format\n').

stream_json(Stream) :-
    clean_triples,
    write(user_error, '% starting to stream\n'),
    json_read(Stream,Text),
    write(user_error, '% finished stream\n'),
    json_to_prolog(Text,Prolog),
    write(user_error, '% converting to prolog\n'),
    scan_amil(Prolog).

```

**Figure 7.6-29. Code Sample 1: JSON Parser and Store Rules**

This essentially scans the JSON stream, pulling out each of the AMIL data items and saving these as (*subject, predicate, object*) triples. Anything not recognized in this format is discarded (such as AMIL create, preconditions, and postconditions).

---

<sup>10</sup> The underlying active semantics of AMIL involves the notion of executing a call to a remote application depending on the evaluation of the node. This happens before we retrieve the AMIL store because the JSOM parser can only parse.

At this point, all the information is available locally to do sophisticated reasoning against. A snippet of logic queries (Figure 7.6-30), the local knowledgebase for individual elements of a “DESIGN\_SET”, next reasons about specific derived properties, and then applies a set of criteria to those properties. This is essentially a logic+control strategy, with a search through the description logic filtered through control predicates.

For the demo, we keep it simple, with the understanding that this ontological rule can easily be changed. For this case, we decide to filter for the design element with the largest value of mass density.

```

triple_num(Subj, Pred, Obj) :- % convert atom to number
    triple(Subj, Pred, O),
    atom_number(O, Obj).

get_design_set(S, mass, Value) :-      % triple store lookup
    triple(S, parent, 'DESIGN_SET'),
    triple_num(S, mass, Value).

get_design_set(S, volume, Value) :-     % lookup with computation
    triple(S, parent, 'DESIGN_SET'),
    triple_num(S, height, V1),
    triple_num(S, width, V2),
    triple_num(S, length, V3),
    Value is V1*V2*V3.

get_design_set(S, density, Value) :-    % higher-level rule
    get_design_set(S, mass, V1),
    get_design_set(S, volume, V2),
    Value is V1/V2.

```

**Figure 7.6-30. Code Sample 2: GEAR Reasoner Which Finds the Maximum Density from Elements in a Set**

The logic predicates labeled `get_design_set` polymorphically match to the ontological constraints of mass, volume, and density. The mass predicate looks up the mass property directly. The volume predicate requires matches for the three dimensions of height, width, and length and then computes the volume after successfully binding to values for each property. The even higher-level density predicate combines the mass and volume predicates to calculate an element density (*could easily change this to a power density which is an important design criteria*).

A powerful potential for reuse of rules exists via ontological classification.

A reusable rule called `mazimize_value` searches a set of paired tuples for the maximum value after the meta-call `findall` provides a list of potential candidates.

The top-level rule “`find_maximum_density(Subj, Density)`” is potentially invoked as a web service or as a command line invocation.

For example, from this server logic, the HTTP query

<http://localhost:5000/load?amil=http://localhost:8080/ArrowWebServices/arrow/arrowGraphExport>

will load from the local AMIL server and `http://localhost:5000/query` will execute the `find_maximum_density` query.

The reasoner code is high-level enough that knowledge engineers can quickly adapt design rules and analysis archetypes as customized GEAR rulebases and store these in CML.

```

server(Port) :-  
    http_server(http_dispatch, [port(Port)]).  
  
    :- http_handler(root(home), index_page, []).  
    :- http_handler(root(load), load_data, []).  
    :- http_handler(root(query), max_density, []).  
  
index_page(_) :-  
    reply_html_page(title('Home'),  
        [ h2(a(href('load?file=JSON_graph.txt'), 'Load data from a file')),  
          h2(a(href('load?amil=http://wcsn262:8001/backup/JSON_graph.txt'),  
              'Load data from an arbitrary web-served file')),  
          h2(a(href('load?amil=http://localhost:8080/ArrowWebServices/arrow/arrowGraphExport'),  
              'Load data from a local AMIL server')),  
          h2(a(href(query), 'Query data example')) ]).  
  
load_data(Request) :- %% File name version  
    http_parameters(Request, [file(Name, [ optional(true) ])]),  
    nonvar(Name),  
    load_json(Name),  
    write(user_error, Request), nl(user_error),  
    reply_html_page(title('File loader'), [p(Name), p(' load completed.')]).  
  
load_data(Request) :- %% URL version

```

**Figure 7.6-31. Code Sample 3: GEAR Web Server with Handlers Pointing to Rules**

The division between the structural ontological world of AMIL and the inferencing world of reasoning is given by a few analogies:

- AMIL choreographs/suggests/navigates the design
  - Design elements are proposed from the ontologies and archetype available
  - Possible design linkages are made available
- Reasoners in GEAR orchestrates/directs/steers the design
  - Combinations of design elements that meet requirements and constraints
  - Utility functions that select the most adaptable and robust combination

Again, the combination is that of organization via graph and then a specific search approach for reasoning.

Thus, AMIL is well suited for working in concert with reasoners, readily supports co-analysis, and is capable of supporting co-simulation for configuration options (refer to 7.6.3.4) and to explore the test-space (refer to 7.6.1.3.5).

#### **7.6.1.4.2 Ontological Rule-Based Languages**

Many patterns that exist in the engineering process can be expressed as archetypes. These archetypal patterns can exist as requirements, analysis process artifacts, design rules, and template architectures. We need a standard way of generating the engineering products through these archetypes, from accessing the knowledge, reasoning on data and rules, and generating instances of the archetypes, i.e. as ectypes.

The aims for selecting a knowledgebase language suitable for reasoning include:

- To optimize expression of data and logic (i.e., rules) in the same language
- To allow graph connectivity to be expressed
- To allow rules to be integrated smoothly with the graph
- To allow statements about statements to be made (i.e., meta-logic\_)
- To be as readable, natural, and symmetrical as possible

We have selected a style of reasoner development which we refer to as Onto-Logical Programming (OLP) because it combines *ontologies* with *logic programming*. Since an ontological schema such as OWL already uses Description Logic (DL) then enhancing this with the more general Logic Programming allows quite a bit of flexibility for developing general purpose GEAR archetype reasoners [SMV11].

The goal is to provide a language environment that allows us to quickly and efficiently develop domain-specific reasoners. The data can come from ontological sources such as a dedicated semantic web, while the rules come from tacit and declared knowledge culled from domain experts. This eventually becomes part of the CML. This merges into DL, which used to describe components and their relationships via OWL, a semantic web version of DL.

The fundamental idea is to express design decisions and synthesis as logic. Design decisions have historically been expressed in natural language. Kowalski<sup>11</sup> has stated eloquently that: *Natural Language = Logic + Control*. To meet this need for a logical foundation followed by a control element, the constrained expressiveness of Description Logic extended by logic programming seemed a natural fit.

Description Logic by itself is extended propositional logic which has no active semantics (other than SWRL), so we depend on the control provided by a logic programming environment<sup>12</sup>.

<sup>11</sup> R. Kowalski, “Logic for Problem Solving”, North-Holland, 1979.

<sup>12</sup> To understand the difference between logic and control consider the following declarative pieces of information. “*Mary likes you if you give her presents and be kind to animals.*” That is pure logic, but only top-down control allows us to solve a problem. Thus a solution combining logic and control “*If you want Mary to like you then give her presents and be kind to animals.*”, which has an active element of control.

This combination allows us to mimic the human design decision process: Logically classifying and then acting on the info, i.e. applying design rules for composition or synthesis

As a declarative approach, logic programming works well, as we first evaluated the ESKER DSE framework with Prolog alone and then added the ontological component later. The declarative syntax is a big plus, as queries in Prolog look like relational SQL or SPARQL, but with more flexibility. As SPARQL views and SPARQL queries amount to the same thing in Prolog, users can quickly get up to speed without having to learn extra syntax.

The ontological reuse benefit has shown promise from the start. Historically, critics point out that domain knowledge is often encoded in a logic program, making it difficult to reuse<sup>13</sup>. However, patterns used with DL and ontological reasoners are standardized. This makes the domain knowledge logic more amenable to reuse and extension. So we see a huge productivity advantage in writing archetype-based synthesis tools via this approach.

As another reuse perspective, we see less of a need to re-implement algorithms for searching and optimization. We can take advantage of existing expert system frameworks (like ESKER), planning systems, theorem provers and design rule checkers, and natural language parsers. The latter can be domain-specific, allowing non-programmers potentially easier entry, as we can reversibly transform between the onto-logical style and controlled natural language.

Similar reuse advantages for Lisp and other interpreted languages with OWL interfaces, yet Prolog contains useful search and query mechanisms out of the box<sup>14</sup>. Having parsers for JSON and libraries available for OWL provides a fast track to solving problems.

**Conciseness of programs.** As a comparison of an “engine power” query in Lisp, SPARQL, and OLP, the OLP is arguably the most concise. The box below features several DL queries followed by an inline constraint which rules out under-powered engines. This is all specified declaratively with the OWL queries meta-interpreted by automatically identifying name-space keywords.

```
%import http://wcsn262:8001/demo/arrow.owl
%import http://wcsn262:8001/demo/CFV.owl
%import http://wcsn262:8001/demo/meta.owl

%% Rule to find component with power threshold

find_engine_with_power(Engine, Limit) :-
    meta:'Engine'(Engine),
```

---

<sup>13</sup> Also we need to assume a closed-world for LP and can only make decisions based on data available. Data not available will result in negation as failure. This is simple pragmatism, as we will never have complete data and perfect knowledge.

<sup>14</sup> Storrle, H., “A Prolog-based Approach to Representing and Querying Software Engineering Models”, In P. T. Cox, A. Fish, and J. Howse, editors, *VLL 2007 workshop on Visual Languages and Logic*, volume 274 of CEUR Workshop Proceedings, pages 71-83.

To query, invoke:

```
?-find_engine_with_power(E,12).
E='CFVEngine2'
```

In this example, the domain knowledge is contained in the ontological terms, making rules easy to write and understand.

The seamless syntactic construction that distinguishes an ontological namespace query from a logical call is quite simple. In the engine example, the ontological predicate is identified by a colon (:) separator, indicating the ontological namespace for the predicate.

- *Onto-logical call*  
**arrow:hasFeature(Engine, Feature)**
- *Logical call*  
**hasFeature(Engine, Feature)**

In general we apply logic and control via three schemes, a classifier, a description predicate, and a control rule.

- Classifier  
**meta:'Engine'(Engine)**  
*Find individual Engine which belongs to ontological class meta:'Engine'*
- Description Predicate  
**arrow:'hasFeature'(Engine, Feature)**  
*Pull all features from that Engine, one will be a power rating Feature*
- Control Rule  
**Power > Limit**  
*Cull the Power ratings greater than some value Limit, and top level will return the matching Engine*

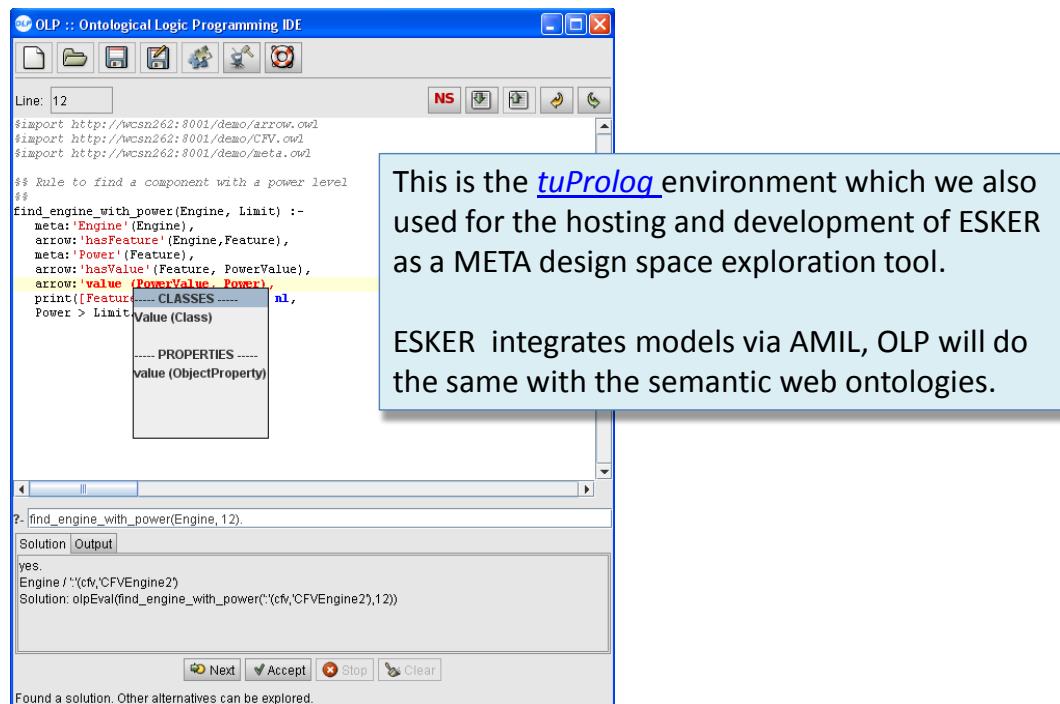
We can ask: What would the rule look like in a controlled natural language?

In Figure 7.6-32, the code to the left is Prolog and that to the right is a hypothetical controlled natural language. The implications and conjunctions map fairly well and we can use the capitalization to indicate the unbound variables quite naturally (Figure 7.6-33).

[YRG-24] Prolog Original	Controlled Natural Language
<pre>%import http://wcsn262:8001/demo/arrow.owl %import http://wcsn262:8001/demo/CFV.owl %import http://wcsn262:8001/demo/meta.owl  find_engine_with_power(Engine, Limit) :-     meta:'Engine'(Engine),     arrow:'hasFeature'(Engine, Feature),     meta:'Power'(Feature),     arrow:'hasValue'(Feature, PowerValue),</pre>	<pre>Import http://wcsn262:8001/demo/arrow.owl Import http://wcsn262:8001/demo/CFV.owl Import http://wcsn262:8001/demo/meta.owl  Find an Engine with a power Limit <b>if</b>     Engine is a meta:Engine <b>and</b>     Engine has a feature called Feature <b>and</b>     Feature is a meta:Power <b>and</b>     Feature has a value called Value <b>and</b></pre>

<pre>arrow:'value'(PowerValue, Power), Power &gt; Limit.</pre>	<p>Value evaluates to Power <b>and</b> Power is greater than Limit.</p>
--	---

**Figure 7.6-32. Difference Between Prolog (left) and a Controlled Natural Language (right)**



**Figure 7.6-33. An example of a Typical Assisted Editing Environment for Query Development**

Much of the context modeling world has adopted an ontological strategy for organizing and classifying environmental data. For the case of an analysis archetype for aerodynamics with the intent of verifying a PCC, we have a design model of a vehicle hull described by the META ontology combined with a wind speed context model which uses SWEET (Semantic Web for Earth and Environmental Terminology) for defining the environment ontology [YRG10]. These two ontologies are combined in Figure 7.6-34.

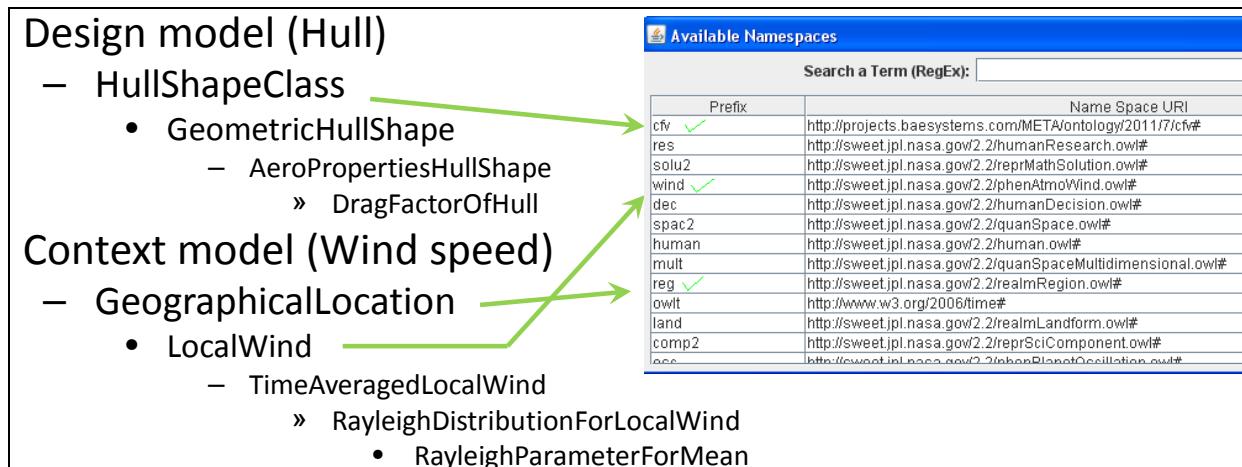


Figure 7.6-34. Loading META/CFV Ontologies and SWEET Ontologies

**Other GEAR platforms.** We have used the open source *tuProlog*, *SWIProlog*, *GNU Prolog* for evaluation. Also we have evaluated *AllegroLisp* which contains an *Allegrograph* graph database, with a query language that uses either Lisp, SPARQL, or a builtin-in Prolog interpreter for rule processing. This may be more industrial-scale applicable. *Protégé* was used for populating the ontologies, and an N3 to OWL converter for rapid ontology development.

**Connection to AMIL.** AMIL uses JSON syntax to describe relationships instead of RDF. By querying the AMIL graph database and parsing the JSON, we can convert that data as triple stores and reason with it the same as we can with OWL data. The information structure in AMIL is equivalent to a semantic web graph, apart from the behavioral semantics for evaluating node data (i.e. the extra semantics with respect to model node evaluation which can't be duplicated by RDF). The external logic programming rules can alternatively take care of the dynamic evaluation.

### Use of Meta-Logic.

With the Description Logic reasoner this meta-query:

```
arrow:'hasFeature'(Engine,Feature)
```

gets converted to the DL API call:

```
dl_query(bind_state_of_Engine, 'arrow:hasFeature', bind_state_of_Feature)
```

The overhead work that goes into this call involves determining the bindings in the triple-store lookups and of parsing down to the namespaces. If the binding of **Engine** is known then we get back **Feature** matches. If the binding of **Feature** is known then we get back **Engine** matches. If all three are known it logically returns True or False. The case of the predicate not being known is less useful because this would involve an additional level of indirection that a domain reasoner would not have much use for (unless it was searching for predicates that were close to the intended predicate, in which case relationships could be used to hone in on the desired predicate).

In general practice, the three necessary API calls include:

- dl\_query(Subject,Predicate,Object)
- dl\_assert(Subject,Predicate,Object)

- `dl_retract(Subject,Predicate,Object)`

The assert and retract only operate on the in-memory state of the knowledge. If we needed overall knowledgebase control

Then this abstraction:

```
%import http://wcsn262:8001/demo/arrow.owl
```

transforms to

```
dl_load(http://wcsn262:8001/demo/arrow.owl)
```

and this complementary call:

```
dl_unload(http://wcsn262:8001/demo/arrow.owl)
```

#### 7.6.1.4.3 Correct-by-Construction

Combining ontologies and formal rules merges the ideas of Correct-by-Classification with Correct-by-Construction. The general principle is that by applying construction rules, semantic definitions, and constraints to a problem domain we can create a Correct-by-Construction design that we can *categorize* for potential re-use. That is the power of an ontological archetype in that the built-in classification scheme allows it to be more readily searchable and therefore a candidate for reuse.

To scope out the challenge, we take the ramp design example and look at what paths we can take with synthesis, co-analysis, and co-simulation:

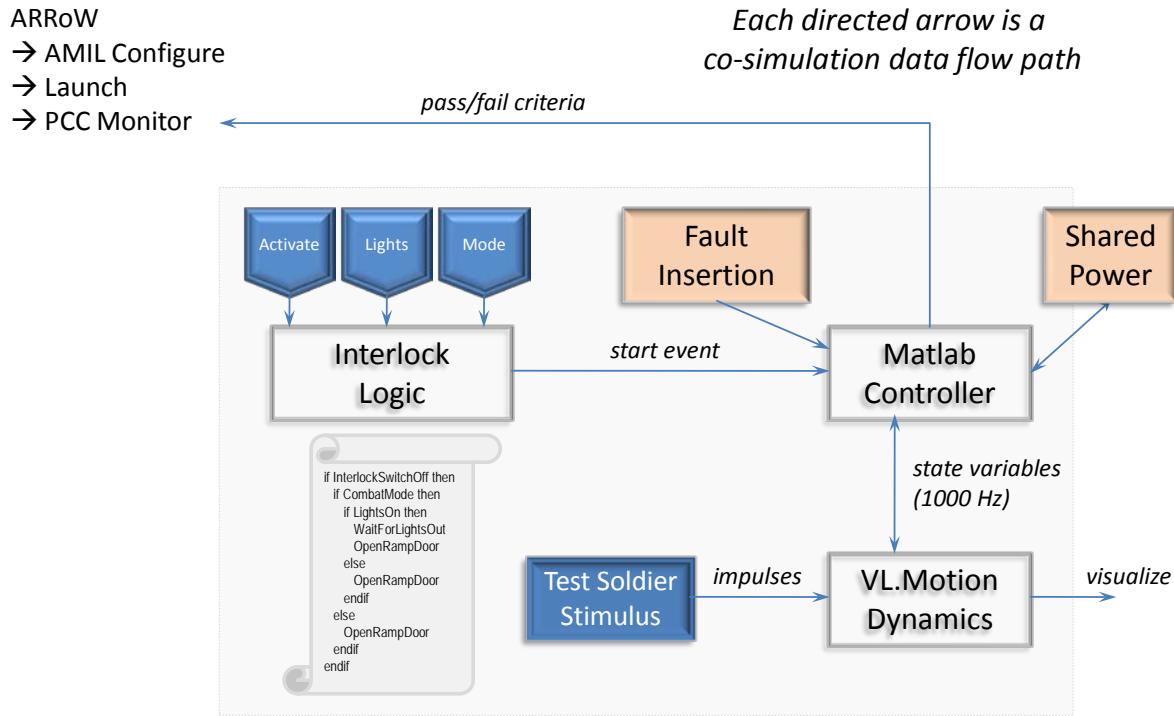
*Say the premise is that the ramp has an interlock with a switch to control the drive. Further extend this to a constraint such that the ramp can't open if the interior lights are on and if the vehicle is in combat mode. In simplest terms this reduces to some predicate logic based on valuations of a remote switch, a hardware sensor reading for the light, and a state machine for the system modes.*

To generalize from this scenario, we will require domain specific language to represent these rules. We can then extract the essential meaning as a template archetype.

A complete specification of the ramp model would include the logic, controller and plant as shown in the figure below. AMIL would serve to configure the co-simulation component pieces, and the T&V thread would monitor the execution if it were running in the context of a PCC quantification test.

One interpretation of the logic specification is shown to the right: the ramp opens if commanded to and the predicates shown apply. The composition logic would need to extract some meaning to indicate when to do what. Do we open the ramp if lights are on or off? If the lights are on and we are not in combat is it then OK? That is essentially what the requirements state.

```
if InterlockSwitchOff then
    if CombatMode then
        if LightsOn then
            WaitForLightsOut
            OpenRampDoor
        else
            OpenRampDoor
        endif
    endif
endif
```



**Figure 7.6-35. Correct-By-Construction Application Involving a Heterogeneous Multi-physics Interface**

**Archetypal Synthesis:** Following is an example of an archetype knowledgebase for building a ramp co-simulation.

```

%%% Example for Domain-Specific Archetypal Specification

%%
%% AMIL fact-base, constructed as Subject-Predicate-Object triples
%%
activates(switch, logic).                                     %
controls(controller, ramp).                                    %
requires(logic, state_machine).                                %
enables(logic, controller).                                 %
overrides(logic, interior_lights).                            %
stimulates(soldiers, plant_dynamics).                         %
modeled_by(plant_dynamics, 'VL.Motion').                      %
modeled_by(controller, 'Matlab').                            %
modeled_by(logic, 'Java').                                    %
generates(ramp, plant_dynamics).                            %
visualizes(renderer, plant_dynamics).                         %
monitors(test_oracle, max_torque_time).                      %
advises(max_torque_time, pass_fail).                         %
produces(controller, max_torque_time).                        %
logs(controller, max_torque_time).                           %
applies(controller, state_variables).                        %
shares(plant_dynamics, state_variables).                     %

%%
%% Archetypal behavior rulebase for specifying causality
%%
+(Subject, Predicate, Object) :-
    call_with_args(Predicate, Subject, Object),
    print(Subject), print(' <<'), print(Predicate), print('>> '), print(Object), nl.

```

```

logical_control(System) :-  

    +(switch, activates, Logic),  

    +(Logic, modeled_by, Exec),  

    +(Logic, requires, States),  

    +(Logic, overrides, Equipment),  

    +(Logic, enables, System).

hinged_slab_dynamics(Data) :-  

    +(plant_dynamics, modeled_by, Exec),  

    +(Slab, generates, plant_dynamics),  

    +(Forces, stimulates, plant_dynamics),  

    +(Renderer, visualizes, plant_dynamics).

hinged_slab_control(State) :-  

    logical_control(System),  

    +(System, modeled_by, Exec),  

    hinged_slab_dynamics(Data),  

    +(Plant, shares, Data),  

    +(System, applies, Data),  

    +(System, controls, Slab),  

    +(Logic, enables, System),  

    +(System, produces, State),  

    +(System, logs, State).

test_controller(PCC) :-  

    hinged_slab_control(State),  

    +(test_oracle, monitors, State),  

    +(State, advises, PCC).

```

### Execution example

```

| ?- test_controller(PCC).

switch <<activates>> logic
logic <<modeled_by>> Java
logic <<requires>> state_machine
logic <<overrides>> interior_lights
logic <<enables>> controller
controller <<modeled_by>> Matlab
plant_dynamics <<modeled_by>> VL.Motion
ramp <<generates>> plant_dynamics
soldiers <<stimulates>> plant_dynamics
renderer <<visualizes>> plant_dynamics
plant_dynamics <<shares>> state_variables
controller <<applies>> state_variables
controller <<controls>> ramp
logic <<enables>> controller
controller <<produces>> max_torque_time
controller <<logs>> max_torque_time
test_oracle <<monitors>> max_torque_time
max_torque_time <<advises>> pass_fail

PCC = pass_fail

```

**Figure 7.6-36. Analysis Archetype Rules**

**Co-Analysis:** It may also be possible to construct dynamic rules in AMIL to execute the behavior according to results from other components.

- MainSwitch -- Activates the ramp driver

- InterlockSwitch -- Act like a deadman switch to defeat the opening of the door
- CombatMode -- A state in the vehicle state diagram
- LightsOn -- Senses the light in vehicle for survivability, followed by a condition variable to hold on until light is off.

AMIL is a language for interfacing to a graph database – augmentation with higher-level rules riding on top of the AMIL layer allows us to construct this logic automatically.

**Co-Simulation:** AMIL could provide a routing table to, say, the remote functional response to a CombatMode query. In this case AMIL would declare which node the states and modes subsystem exists within. When CombatMode is queried the AMIL runtime would execute and serialize the necessary call and response to the remote node.

The question of co-simulation is thus: Do we want to exercise AMIL so that it can make decisions based on these kinds of rules and state data, or do we want to defer this logic to the Simulink or other controlling simulation?

- **Basic Synthesis:** If we defer this to the controlling simulation, we lose the ability to compose the model from the design elements since the development will need to be done in the language of the controlling simulation (e.g. Java or Simulink).
- **Basic Co-Analysis:** If we do this composition using AMIL, then we will need mechanisms to stitch the ramp drive, switch, sensor, and state machine together with customized dynamic rules, and then verify that it works according to the test criteria.
- **Configured Co-Simulation:** The other alternative is to employ AMIL as a conventional run-time configuration language and use the AMIL API when we need it to call out from the controlling simulation. In this case, the Simulink code would be comprised of many external links, one for each of the logic elements. We lose the ability to compose but can use AMIL to pull external pieces into the main simulation.

This argument essentially places AMIL into a specific role for co-simulation - it provides us the basic configuration file or a data path for the primary simulation language that engineers can use.

This provides for composition in and integration of other co-simulation languages, with an assortment of composition knowledgebases, one for each language, i.e. a data-flow composer for Simulink, an object-oriented one for Java and so on. This allows the generation of this kind of decision logic without having to write the if-then-else rules.

## 7.6.2 Co-Analysis and Exploration

### 7.6.2.1 Principles behind GEAR

The prominent idea behind GEAR is to apply similar rule-based semantics in the context of developing archetypes for analysis, design, and implementation. The goal is to extend the information laid out in the AMIL and leave it in a symbolic format, suitable for mapping into more concrete representations. The symbolic representation thus forms an “archetype” for the specified behavior.

The general concept is to start with a domain model of the behavior that we want to specify. The domain model is separated into two classes: (1) fundamental atomic actions or behaviors at the low-level and (2) archetypal rules at a higher-level which serve to stitch the actions together. The actions at the low-level can be constructed from “subject-predicate-object” triples, which denote causal relationships between intent and a symbolic realization. The symbolic realization could be retained as an abstraction or a set of possible alternatives.

So a typical triple may be represented as:

```
operating_environment <<targets>> destination
```

Here (*operating\_environment*, *targets*, *destination*) illustrates an example of the (*subject*, *predicate*, *object*) triplet pattern we have in mind. Understandably, this relationship by itself is fairly meaningless until we place it in the context of a larger-scale behavior. So the archetypal behavior starts in motion when several of these individual triples form a conjunction that accomplishes a larger task and executes to show self-consistency and correctness.

The symbology of the subjects and predicates will always represent things that we can build or reuse – whether they are software, hardware, or human actors depends on what best the ARROW process decide that the symbols eventually map to.

#### 7.6.2.1.1 GEAR Approach

We start out with a domain model of some sequence (potentially concurrent) of steps that may build into a cyber-physical realization. The main theme for the sequence is that it forms a set of behaviors that would typically reproduce a human’s action (automation) or improve on some already automated realization. In practice, these sequences draw from typical or archetypal behaviors that have stood the test of time. The key is that we do not want to reinvent the wheel each time an engineering development needs to implement a behavior. Instead we can extract from the repository of behavioral recipes and apply them to a start-up design task thereby reducing our development time.

The need then is for a description that can generate concrete realizations based on the behavioral archetypes and requirements.

Let’s start with an example drawn from a typical need for any vehicle equipped with a sophisticated armament system. The need is to create a projectile fly-out model suitable for analysis, design, and integration testing. A directed graph of the archetype for accuracy analysis is shown earlier in Figure 7.6-22.

On its own, this directed graph provides the developer with a general flow for data and actions to accomplish a complete weapons effectiveness analysis. What is missing from it are formal

semantics and causal ordering relationships for the arrows. In other words, we have an idea for the flow of data but it remains imprecise and open to interpretation, and therefore ambiguous.

The goal is to take this *domain model* of fly-out accuracy analysis and create an archetypal knowledgebase from it based on AMIL subject-predicate-object triples and higher-order or composite rules representing the main stages of the behavior.

We start by creating a set of triples describing the individual behaviors that we need to implement in Figure 7.6-22. Starting from the left side of the diagram, we note that (*operating\_environment*, *targets*, *destination*) is a reasonable behavior to include. In the text box below, we gather these low-level behaviors together with that triple as the first entry. (Note that this is realized in a concise syntax, with the predicate leading the triple as a functor of the subject and object). This first set forms the AMIL factbase.

Listed below the factbase, we provide a set of archetypal rules governing how the predicate actions fit together. We have three high-level rules corresponding to the three main stages of an analysis model – the gun pointing, the projectile trajectory fly-out, and the weapon target lethality.

```
%%
%% AMIL fact-base, constructed as Subject-Predicate-Object triples
%%
targets(operating_environment, destination).
resides_on(operating_environment, terrain).
modeled_by(flyout, '3DOF').
initialized_by('3DOF', aero_data).
influenced_by('3DOF', met_data).
guided_by('3DOF', round_corrections).
generates('3DOF', aimpoint_accuracy).
stabilized_by(gun_pointing, stabilization_algorithm).
starting_on(stabilization_algorithm, terrain).
points_at(stabilization_algorithm, destination).
compensating(stabilization_algorithm, drive_model).
perturbed_by(stabilization_algorithm, moving_vehicle).
produces(stabilization_algorithm, pointing_accuracy).
```

```

%%

%% Archetypal behavior rulebase for specifying causality

%%

pointing_accuracy(Pointing) :-  

    resides_on(operating_environment, Terrain),  

    targets(operating_environment, Destination),  

    stabilized_by(gun_pointing, Alg),  

    starting_on(Alg, Terrain),  

    points_at(Alg, Destination),  

    compensating(Alg, drive_model),  

    perturbed_by(Alg, moving_vehicle),  

    produces(Alg, Pointing),  

    selected(ammo, Caliber),  

    conditioned_by(Caliber, gun_geometry),  

    ignites(interior_ballistics_model, Caliber),  

    dispersed_by(Pointing, interior_ballistics_model).

projectile_flyout(Aimpoint) :-  

    pointing_accuracy(Pointing),  

    selected(ammo, Caliber),  

    initializes(Pointing, Flyout),  

    modeled_by(Flyout, DOF),

```

**Figure 7.6-37. Behavioral Archetype**

An executable query to test the rules results in the following valid response:

```
?- weapon_effect(Effect).
Effect = lethality+error
```

This by itself is not too interesting other than it demonstrates that the set of rules is executable and self-consistent. If we wish to add a level of introspection, we can add the following meta-rule to describe a more natural subject-predicate-object syntax:

```
+ (Subject, Predicate, Object) :-  

    call_with_args(Predicate, Subject, Object),  

    print(Subject), print(' <>'), print(Predicate), print('>> '), print(Object), nl.
```

Restructuring our top-level rules with this new approach following a domain-specific syntax:

```

pointing_accuracy(Pointing) :-  

    +(operating_environment, resides_on, Terrain),  

    +(operating_environment, targets, Destination),  

    +(gun_pointing, stabilized_by, Alg),  

    +(Alg, starting_on, Terrain),  

    +(Alg, points_at, Destination),  

    +(Alg, compensating, drive_model),  

    +(Alg, perturbed_by, moving_vehicle),  

    +(Alg, produces, Pointing),  

    +(ammo, selected, Caliber),  

    +(Caliber, conditioned_by, gun_geometry),  

    +(interior_ballistics_model, ignites, Caliber),  

    +(Pointing, dispersed_by, interior_ballistics_model).  
  

projectile_flyout(Aimpoint) :-  

    pointing_accuracy(Pointing),  

    +(ammo, selected, Caliber),  

    +(Pointing, initializes, Flyout),  

    +(Flyout, modeled_by, DOF),  

    +(Caliber, provides, Aerodata),

```

**Figure 7.6-38. Alternate Rule Layout**

This capability for introspection on the calls allows us to manipulate the triples and do meta-processing on the causality chain. In this specific case, we are simply reformatting the matching of the high-level archetypal behavior to the low-level action, changing the prefix predicate notation *predicate(subject, object)* to a possibly more preferable infix style of *(subject, predicate, object)*.

We can also do more extensive processing, as for example in generating code, in which case an execution of the logic will fire predicates that match against instance patterns.

```

| ?- weapon_effect(Effect).
amm0 <<selected>> medium_caliber
medium_caliber <<causes>> lethality
operating_environment <<resides_on>> terrain
operating_environment <<targets>> destination
gun_pointing <<stabilized_by>> stabilization_algorithm
stabilization_algorithm <<starting_on>> terrain
stabilization_algorithm <<points_at>> destination
stabilization_algorithm <<compensating>> drive_model
stabilization_algorithm <<perturbed_by>> moving_vehicle
stabilization_algorithm <<produces>> pointing_accuracy
ammo <<selected>> medium_caliber

```

```

medium_caliber <<conditioned_by>> gun_geometry
interior_ballistics_model <<ignites>> medium_caliber
pointing_accuracy <<dispersed_by>> interior_ballistics_model
ammo <<selected>> medium_caliber
pointing_accuracy <<initializes>> flyout
flyout <<modeled_by>> 3DOF
medium_caliber <<provides>> aero_data
3DOF <<initialized_by>> aero_data
3DOF <<influenced_by>> met_data
3DOF <<guided_by>> round_corrections
3DOF <<generates>> aimpoint_accuracy
circular_error <<required_by>> aimpoint_accuracy
aimpoint_accuracy <<derives>> error

Effect = lethality+error

```

In this flyout example, variants of the trajectory model could encompass the lower fidelity 3DOF model and a higher fidelity 6DOF model.

#### **7.6.2.1.2 Applying GEAR to domains**

There are several benefits to using the ontological/AMIL predicate logic approach on domain-specific problems.

For one, the archetypal specification of a system is easily understandable –convenient naming can be used and the specification is directly executable if something is not understood.

A single language can be applied for specifying requirements, defining architecture, behavior modeling, as well as simulation configuration. This can be extended for specifying testing archetypes, which essentially follows the (test\_case, stimulate, system) triple paradigm.

It also meets the desire for simplicity, non-ambiguity, and completeness in a single integrated specification. We thus have a support infrastructure that is common between tools and the domain language.

In "Mathematical Models for Computing Science," C.A.R. Hoare states (August 1994):

- *Propositional and predicate logic provide all the basic concepts needed for a systematic engineering design methodology.*
- *The operation of each component can be described scientifically by a separate predicate.*
- *A non-deterministic product is described by the disjunction of predicates describing its alternative modes of behavior.*

By using logical constructs instead of the algebraic ones we can greatly simplify the architecture specification and model development. Our examples illustrate several practical applications of logical constructions leading to causal linkages between chained assumptions and guarantees. These are the initial steps to synthesis of models via domain-specific archetypes (i.e. a realization of the Galileo functionality for ARROW).

The supplemental exercises below provide more in-depth examples and explanation on how this archetypal behavior modeling can be applied. The first example borrows from the planning realm and describes a reference architecture with details on the predicate logic. The second and third example describes a simple assembly tree construction, and suggests the path from predicate logic to ontological logic.

Once accepted, the more elaborate reasoning tools such as ECTo and ESKER will allow quick prototyping and automated evaluation to determine a potentially feasible and/or optimal assembly. The plan is to apply this approach to a variety of archetypal domains.

## GEAR suite

	Ontological Reasoner (with RDF)	Model Reasoner (with AMIL)	Concepting Reasoner (with AMIL)
Design Space Exploration, Analysis of Alternatives		ESKER	LP
Vehicle Spatial Layout			ECTo
Requirements Allocation	OLP		
Analysis Composition	OLP		
Design-rule-based Synthesis	OLP		
Template Architectures	OLP		
Test Case Generator	OLP		
PCC Evaluation		ESKER	

- AMIL has extra semantics with respect to model node evaluation which can't be duplicated by RDF
  - If that is not required then ESKER becomes a type of OLP, with nodes evaluated by the logic program
- The concepting reasoner visualizes spatial representations, which is difficult to do in pure logic

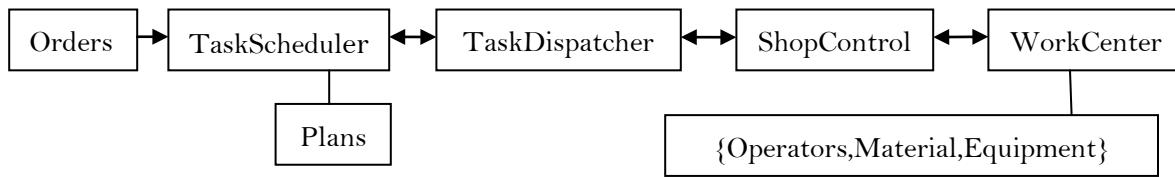
**Figure 7.6-39. Gear Suite**

### 7.6.2.1.3 #1: Manufacturing Example

The first example is a simple industrial manufacturing scenario which we use to stretch our understanding of how the language can be used. In this example, we show how a logic-based requirements and architecture specification can provide an integrated framework for the development and reuse process.<sup>15</sup> As an illustration, we will apply typical industrial decisions, resource capabilities, and resource sharing (refer to BBN's work) and allocation to a task workflow.

**Application Domain.** Our example is set in a manufacturing domain. An abstract representation of this manufacturing environment is shown in Figure 7.6-40.

<sup>15</sup> Logical specification does not introduce artifacts in the specification, such as complex diagrams and symbols, unique node representations, specialized networks, complex database schema representations, etc. Instead, it deals directly with the goals, plans, capabilities, and constraints of the application domain. The specification does not require information modeling as the first step in developing the specifications. Thus, there is no need for object diagrams, DFDs (Data Flow Diagrams), E-Rs (Entity-Relationship Diagrams), STDs (State Transition Diagrams), etc. All of these representations are implicit in the declarative-style logical specification.



**Figure 7.6-40. Abstract Representation of the Operating Environment**

When manufacturing orders are received, they are passed through the scheduling-dispatching channel to the shop-control activity. One of the activities that shop control is responsible for is 'material move' operation. We will assume that the manufacturing plant uses a just-in-time approach, where an immediate response to all of the incoming work orders is required. Thus, when a move order is received from the shop control specifying a certain material to be moved, the work center accepts the order if it has the necessary resources; otherwise the shop control is notified that sufficient resources are not available and that the material move order cannot be accepted. In the latter case the shop control will try other alternatives.

Our work center consists of small warehouse, transportation equipment, and transportation equipment operators. To keep our illustration simple, only some of the normally available resources will be considered.

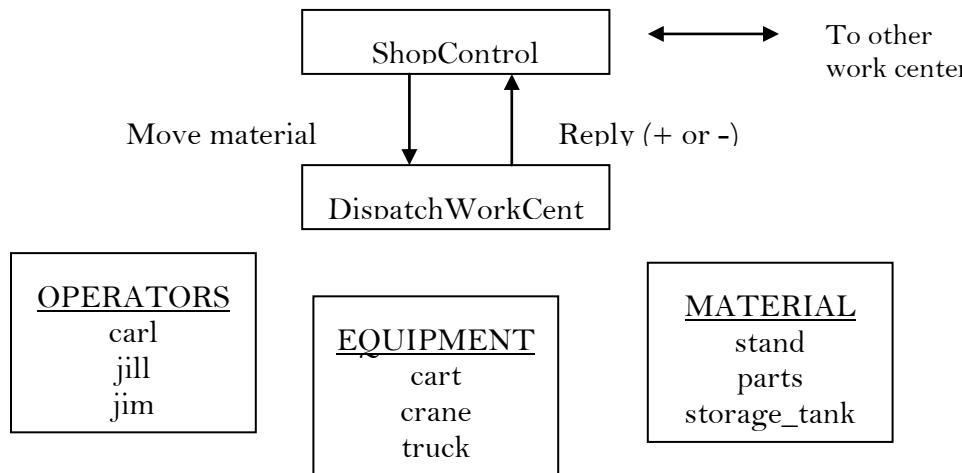
**Domain Model.** A domain model specifies the behavior of the entities making up the problem space. As such, the domain model is also the representation of the process, because it describes what is taking place in the domain. It considers functions, data, rules, and the various entities (objects) and their capabilities and constraints.

The primary purpose of the domain model is to describe generic problems, which can then be represented as archetypal candidates for reuse. Our 'material move' operation can be considered in this context, because it can represent a family of 'move' operations that have the same generic structure, and therefore are considered archetypal.

A domain model is logical because it does not assign functions to specific components. It only specifies the applicable interface and the information that flows through this interface.

A domain model can be specified informally (textual description), graphically (diagram description), or formally. We started with an informal description of the manufacturing domain, Figure 7.6-41 shows a block diagram representation with instance data, and then we will look at the formal (logical) representation.

One immediately sees the classification of the categories Operator, Equipment, and Material with several candidate instances associated with these classes.



**Figure 7.6-41. Dispatch Work Center**

The next step is to represent the above model formally. The external interface of our work center is simple. The top-level domain model for the material move function (work order request) can be expressed as:

```
move(Material)
```

where 'Material' is a parameter that will be specified when the move order is requested.<sup>16</sup>

**Reference architecture.** The reference architecture specifies the allocation of tasks needed to meet the desired goal. It also supports the development of a family of related systems. Thus, it is important to keep it in a generic parameterized form, so that it can be extended, reconfigured, and reused. We can also look at the reference architecture as an extension of the domain model. Whereas the domain model specifies **WHAT** is needed, the reference architecture specifies **HOW** to accomplish it. The logical architecture model is potentially layered, with each layer providing additional detail and less abstraction.

The reference architecture uses a logical specification. Instead of developing a new architecture description language, we will use a conventional logic programming language.<sup>17</sup> But before we get too far, a few comments about Prolog notations and its unique conventions are necessary.

Prolog is basically a first order logic (FOL) programming language with a number of non-logical extensions. A Prolog predicate sentence has a head (left side) and a body (right side), separated by the symbol '`:`' (meaning if). The Prolog declaration, '`p :- q, s.`' states that the assertion p is true if the assertions q and s are true. From a goal viewpoint we can interpret the above assertion as a goal and two subgoals.

---

<sup>16</sup> The 'move' function, of course, is only one of the many functions applicable to our work center. Others include restocking, acquisition of transportation equipment, etc.

<sup>17</sup> Our choice in this case is Prolog, but another logic programming language could be used as well.

In an alternate interpretation, we could consider the goal as a responsibility and the individual plans as obligations associated with the specific responsibilities. Yet in another interpretation, the goal could be considered to be a strategy<sup>18</sup>.

In the conventional Prolog notation, names starting with a lower-case letter represent predicates and constants and those starting with an upper-case letter represent variables (these are initially unbound). Lists are enclosed in square brackets. Thus [a,b,c] is a list consisting of three elements a, b, and c. The individual elements may represent parts, equipment, etc. Lines beginning with a '%' are comments.

**Decomposition of problem.** The solution should naturally come out of the problem domain. A logical specification can support system decomposition at all levels of representation. Hierarchy levels are variable and different parts of the system may not be expanded to the same level. As we will note later, there will be both horizontal and vertical decomposition.

The same decomposition approach also provides the capabilities for including a number of alternatives, such as alternate plans, backup tasks, or error recovery actions in the process plan. These alternatives are selected if a goal cannot be satisfied using the primary path (normal operating procedures). Since pre-conditions and post-conditions are implicit in the declarations, special notation is not required. However, any applicable constraints must be declared explicitly.

We can express the relationship between the domain model and the reference architecture as a logical predicate:

```
move(Material) :- % Implicit Response T/F
    find_transport(Material,Transport),
    find_operator(Transport,Operator),
    make_assignment(Operator,Transport,Material).
```

The top level goal (or responsibility) is 'move(Material)'. The right hand side of the predicate states the specific tasks (or obligations) that need to be performed to determine whether the move request can be accepted by the work center. The goal is satisfied if we can find the proper transport, locate an available qualified operator, and then make the move assignment (resulting in an affirmative response). Note, that these are also the preconditions for a successful material move operation. The above expression represents a horizontal composition, with the meaning of the individual terms to be defined at the next lower level in the definition hierarchy.

For example, to determine if the proper transportation means are available, we have to find out first what kind of transport is needed to move the requested material and then we have to check if that particular transport is available. Similar considerations apply to the selection of an operator, where we first have to determine if an operator is available and then have to check if that particular operator has the needed operational skills.

However, if one or more of these requirements are not met, then the work center will need to issue a negative response (i.e. an implicit failure). In a full system implementation this message

---

<sup>18</sup> The specific interpretation will normally be application and user environment dependent.

would retry and go to ShopControl, where alternate means would have to be found to move the needed material to the shop floor.

```
move(+Material) :-  
    decline_request(+Material).
```

In the logical specification resource allocation constraints are defined by the rules that will assign an operator and the appropriate transport equipment to the material move operation. This assignment will occur only if the resource availability constraints are met.

The dispatch work center spec is presented below. The write statements are not actually part of the specification, but are included for animation and testing purposes.

```
% ----- Logical Specification -----  
% Dispatch Work Station Logical Specification  
% move is the responsibility; find_transport, etc. -- are obligations  
  
move(Material) :- % Response T  
    find_transport(Material,Transport),  
    find_operator(Transport,Operator),  
    make_assignment(Operator,Transport,Material),!.  
  
move(Material) :- % Response F  
    decline_request(Material).  
  
find_transport(Material,Transport) :- % Response T  
    move_by(Material,Transport),  
    equipment(Transport,available).  
  
find_operator(Transport,Operator) :- % Response F  
    operator(Operator,available),  
    operator_skills(Operator,Skills),  
    have_skill(Transport,Skills).  
  
make_assignment(Operator,Transport,Material) :- % Response T  
    assign_operator(Operator),  
    assign_transport(Transport),  
    issue_work_order(Operator,Transport,Material).  
  
decline_request(Material) :-  
    write('resources are not available to move '),  
    write(Material),nl.  
  
assign_operator(Operator) :-  
    retract(operator(Operator,available)),  
    assert(operator(Operator,busy)).
```

```

assign_transport(Transport) :-
    retract(equipment(Transport,available)),
    assert(equipment(Transport,busy)).

issue_work_order(Operator,Transport,Material) :-
    assert(assignment(Operator,Transport,Material)),
    write(Operator), write(' has been assigned to move '),
    write(Material), nl.

move_completed(Operator) :-
    assignment(Operator,Transport,_),
    release_transport(Transport),
    release_operator(Operator),
    assignment_completed(Operator),!.

release_operator(Operator) :-
    retract(operator(Operator,busy)),
    assert(operator(Operator,available)).

release_transport(Transport) :-
    retract(equipment(Transport,busy)),
    assert(equipment(Transport,available)).

assignment_completed(Operator) :-
    retract(assignment(Operator,_,_)),
    write(Operator), write(' has completed move assignment'), nl.

```

```

%auxiliary logic (used to find if an operator has the needed skill)
have_skill(X, [X|_]).  

have_skill(X, [_|Y]) :- have_skill(X,Y).

```

```

% ----- instance specification -----
% Sample Dispatch Work Station knowledge base facts, store as triples

```

```

% transportation needs (capabilities)
move_by(storage_tank,crane).
move_by(stand,truck).
move_by(parts,cart).

```

```

% operator availability status
operator(carl,available).
operator(jill,available).
operator(tom,available).

```

```
% operator skills (capabilities)
operator_skills(carl,[truck,cart]).    %% triples with object a list
operator_skills(jill,[crane,cart]).
operator_skills(tom,[truck,cart]).  
  

%equipment availability status
equipment(crane,available).
equipment(truck,available).
equipment(cart,available).  
  

% ----- test specification -----
% a simple behaviour test scenario  
  

test:-
    move(storage_tank),
    move(parts),
    move(stand),
    move_completed(jill),
    move(parts).  
  

% ----- End of Logical Specification -----
```

**Figure 7.6-42. Material Move Specification in Prolog**

We note that the top level specifies the "WHAT" type requirements (responsibilities and obligations). The "HOW" details are defined and elaborated at the lower levels in the specification. The "WHO" question will be answered during the system implementation phase.

It is important to note the "information hiding" aspects of the logical representation. In our example 'move(Material)', declared at the top level, is the only link to the external environment and hides the internal operations.

It is also important to note that all the potential interconnections are part of the hierarchical representation. Thus, we see no need for a specialized module interconnect language or separate data flow diagrams.

Our problem representation is an executable specification because it is an operational Prolog program. To test it (using the simple 'test' predicate which is defined at the end of the specification listing), we can load it and run it under a Prolog interpreter or compiler. For example, using the SWI Prolog compiler, we obtain the following results:

```
-----  
Welcome to SWI-Prolog (Version 2.5.2)  
Copyright (c) 1993-1996 University of Amsterdam. All rights reserved.
```

```

1 ?- [msi7].
msi7 compiled, 0.17 sec, 5,624 bytes.

Yes
2 ?- listing(test).

test :- 
    move(storage_tank),
    move(parts),
    move(stand),
    move_completed(jill),
    move(parts).

Yes
3 ?- test.
jill has been assigned to move storage_tank
carl has been assigned to move parts
tom has been assigned to move stand
jill has completed move assignment
resources are not available to move parts

```

Yes

---

#### Figure 7.6-43. Testing the Logical Specification

Examining the results of this test (actually a trace), we note that the first three move requests are satisfied, but the fourth fails because all of the needed resources have already been allocated. Note that although Jill is available as an operator, the cart is still in use. Thus, the fourth request fails.

A similar approach can be used to specify other manufacturing operations. For example, we could develop a specification for a make operation, 'make(Gizmo,Quantity)'. The higher level specification, for example, could include a number of make and move operations. Again, the individual tasks (goals) will be decomposed as needed.

Our example illustrated a simple approach to handling constraints and capabilities. The constraints considered included the availability of suitable transportation equipment and a qualified operation. In a more complex situation, we will need to consider other types of resources that may be physical (consumable or permanent) or logical (such as support activities). Resources also may be shared by several tasks. In this case the resource model contains the specifications and the status of all resources -- material, equipment, as well as their current capabilities.

In addition to resource and capability constraints, it will be necessary to specify time and priority constraints, such as response time to a task request or the task priority level. In some situations, response time will be expressed as a function of priority. When developing the specifications, we must remember that all of these constraints need to be expressed in a logical form.

Logic described in this fashion leads to architecture reusability. For example, we could modify our dispatch example for a different application, such as machine selection. In this situation, we would consider machine operators instead of equipment operators. A task in this case would denote an operation performed on a machine and the skill list could specify the capabilities associated with a specific machine or a machine operator.

Logical specifications are easily extendable (composable). For example, in our illustrative example we did not consider material availability. However, this additional constraint can be included by modifying the top level specification as shown below (note the added line #2):

```
-----
move(Material) :-  
    material_available(Material),  
    find_transport(Material,Transport),  
    find_operator(Transport,Operator),  
    make_assignment(Operator,Transport,Material).
-----
```

**Figure 7.6-44. Adding Material Availability Constraint**

and then adding the appropriate expansion for 'material\_available(Material)'.

**Implementation Architecture.** The logical specification developed above is only a generic exemplar. However, it provides the foundation for developing specific applications. The applications-level architecture definition provides a precise statement of the specific problem, it clearly defines components, their interconnections, specific application constraints, etc.

Implementations should not appear in the reference architecture definition apart from instance data. This approach will permit the same abstract model to serve as a framework for implementation in different operating environments using different architectural styles.

The development of the applications architecture is the last step in the stepwise refinement process that starts with the domain model (highly abstract) and ends with a definition (highly specific) that is suitable for implementation in components. This example was left in a non-ontological format so that we can see how the transition to an ontology-based description plays out. This is described in the next two examples.

#### 7.6.2.1.4 #2: Structured Synthesis Archetype

Another example of an archetype is the structural formulation. The pattern is one of having a generic blueprint for how the parts fit together within the context of an assembly tree. The AMIL triple, denoted as an *assembly*, provides the subject-predicate-object low-level actions that the higher level construction rules act on.

What this demonstrates is the concept of a *variant*. The variant essentially gives a several concrete realizations to the abstract basicpart.

```

%%% Simple example of a bike assembly tree, with variants included in Prolog.
%%%
%%% This is partially decomposable whereby the "variant" functor is left open
%%%
%%% to invoke execute: "parts_of(bike, M)?"
%%%
%%% This will generate all the possibilities of a bicycle variant

basicpart(rim).
basicpart(spoke).
basicpart(rearframe).
basicpart(handles).
basicpart(gears).
basicpart(bolt).
basicpart(nut).
basicpart(fork).

assembly(bike,[wheel,wheel,frame]).
assembly(wheel,[spoke,rim,hub]).
assembly(frame,[rearframe,frontframe]).
assembly(frontframe,[fork,handles]).
assembly(hub,[gears,axle]).
assembly(axle,[bolt,nut]).

% basicpart could be defined as assembly(x,[]).

variant(rim, deep_rim).
variant(rim, shallow_rim).
variant(spoke, bladed_spokes).
variant(spoke, circular_spokes).
variant(rearframe, titanium).
variant(rearframe, aluminum).
variant(rearframe, carbon).
variant(rearframe, steel).
variant(handles, drop_bar).
variant(gears, 5-gears).
variant(gears, 6-gears).
variant(bolt, steel_bolts).

```

```

variant(nut, steel_nuts).

variant(fork, Fork) :- variant(rearframe,F), atom_concat(F, '_fork', Fork).

%%% Rules

parts_of(X,P) :-
    parts_cumulative(X,Parts,[]),
    one_of_each(Parts, P).

parts_cumulative(X,[X|Hole],Hole) :- basicpart(X).
parts_cumulative(X,P,Hole) :- assembly(X,Subparts),
    parts_cumulative_list(Subparts,P,Hole).

parts_cumulative_list([],Hole,Hole).
parts_cumulative_list([P|Tail],Total,Hole) :-
    parts_cumulative(P,Total,Hole1),
    parts_cumulative_list(Tail,Hole1,Hole).

one_of_each([],[]).
one_of_each([H|T],[H1|Ts]) :- variant(H,H1), one_of_each(T,Ts).

```

The result of a run will generate a collection of all possibilities of the bike fitted with different components. The following query represents only the first few matches for possible assemblies:

```

| ?- parts_of(bike,L).

L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,titanium,titanium_fork,drop_bar] ? ;
L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,titanium,aluminum_fork,drop_bar] ? ;
L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,titanium,carbon_fork,drop_bar] ? ;
L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,titanium,steel_fork,drop_bar] ? ;
L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,aluminum,titanium_fork,drop_bar] ? ;

```

```
L = [bladed_spokes,deep_rim,5-gears,steel_bolts,steel_nuts,bladed_spokes,deep_rim,5-
gears,steel_bolts,steel_nuts,aluminum,aluminum_fork,drop_bar] ?
```

### 7.6.2.1.5 #3: Ontological Synthesis Archetype

The next step is to merge the above structured archetype into an ontologically structured archetype. To do this, we first need a classification scheme which is provided by the following OWL N3 declaration:

```
@prefix : <http://localhost:2000/bike.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

: a owl:Ontology .
:basicpart a owl:Class .
:structure a owl:Class .
:assembly a owl:ObjectProperty .

:rim a :basicpart .
:spoke a :basicpart .
:rearframe a :basicpart .
:handles a :basicpart .
:gears a :basicpart .
:bolt a :basicpart .
:nut a :basicpart .
:fork a :basicpart .

:concept a :structure ;
:assembly :wheel ,
:frame .

:frame a :structure ;
:assembly :rearframe ,
:frontframe .

:frontframe a :structure ;
:assembly :fork ,
:handles .
```

```

:wheel      a      :structure ;
            :assembly :spoke ,
                        :rim ,
                        :hub .

:hub       a      :structure ;
            :assembly :gears ,
                        :axle .

:axle      a      :structure ;
            :assembly :bolt ,
                        :nut .

```

Then we can follow this with the declaration of the variant parts:

```

:deep_rim      a      :rim .
:shallow_rim   a      :rim .
:bladed_spokes a      :spoke .
:circular_spokes a      :spoke .
:titanium      a      :rearframe .
:aluminum      a      :rearframe .
:carbon        a      :rearframe .
:steel         a      :rearframe .
:dropbar       a      :handles .
:five_gears    a      :gears .
:six_gears     a      :gears .
:steel_bolts   a      :bolt .
:titanium_fork a      :fork .
:aluminum_fork a      :fork .
:carbon_fork   a      :fork .
:steel_fork    a      :fork .
:steel_nuts    a      :nut .

```

Once the classifiers and instances are defined then we can reason on the parts using an ontological logical inference engine.

```

%import http://localhost:2000/bike.owl
%import http://www.w3.org/2002/07/owl
%import http://www.w3.org/1999/02/22-rdf-syntax-ns

```

```

parts_of(X, List) :-
    findall(Parts, part_of(X, Parts), PartsList),
    one_of_each(PartsList, [], List).

part_of(X, X) :-
    bike:'basicpart'(X).

part_of(X, Y) :-
    bike:'assembly'(X, Subpart),
    part_of(Subpart, Y).

one_of_each([], List, List).
one_of_each([Type|Rest], Input, List) :-
    rdf:'type'(Obj, Type),
    one_of_each(Rest, [Obj|Input], List).

```

The intent is the same as the purely structural example, yet the logical code comes out more clean and concisely because of the classification structure imposed via the OWL organization and instancing approach. There are fewer choices allowed to describe the system according to the description logic semantics, so that the logical rules tend to be easier to understand. There is also a greater possibility for reuse and less maintenance requirements due to the longevity imposed by a good classification scheme.

### 7.6.2.2 ESKER

This section lays out the approach that we applied to building the Expert-System Knowledgebase Evaluation Reasoner (ESKER) as a tool for design space exploration. In particular, we describe how we have tied together AMIL and logical semantic reasoning to facilitate DSE, with ESKER containing the engine that drives the search. The semantic web reasoners available use a similar inference engine (Prolog) to that which is described here. ESKER also uses a declarative form, making it very compatible with triple-store and description logic.

ESKER evaluates utility criteria for a given set of components selected from a *set of variants*. We initially assume that the model components would fit together; a precursor archetypal model actually establishes the specification for components that *can* get integrated together, which is also what ECTo does from vehicle structural design rules.

#### 7.6.2.2.1 Background

The approach described in this section provides a reusable pattern to optimizing systems that require an evaluation of alternatives and design space exploration, either in terms of concepts or of design choices.

System optimization has historically remained a challenging problem because the complexity involved in simply choosing between alternatives of any significant number makes a purely quantitative approach prohibitive. Although algorithmic automation approach can alleviate the

bookkeeping, several challenges remain, especially in terms of integrating results from a set of tools that provide the intermediate decision support.

**Concepting and Design Phases.** The general statement of the problem is concisely framed in a basic two-dimensional design space. The scenario typically occurs with the design of any sufficiently detailed product, such as a ground vehicle or a weapon system and it involves selecting alternatives with respect to some set of criteria. Within the first dimension, we have a set of concept or design alternatives. Some examples may include:

- Capacity of vehicle in terms of different count of troops
- Tracked vs. Wheeled
- Gun Caliber
- Engine Type
- *Etc.*

In the second dimension we have a set of optimization criteria, in which we use to arrive at the best choice of element alternatives. The criteria can have various requirements and constraints associated with their description and typically fall into a set of established categories, such as:

- Cost
- Reliability
- Performance
- Weight
- *Etc.*

The system engineering puzzle is to choose which alternatives fit best together within a given set of criteria. The major difficulty in doing this from a *global* perspective is that both the product and optimization categories cross a broad spectrum of disciplines and we will likely have to integrate a number of disciplines and analysis tools together to provide the most effective solution. That is the nature of system engineering, and why a cross-disciplinary approach is vital.

The results of this implementation shows that an expert system backed by a dynamic knowledge base is well suited for the optimization task. We will explore the basis for this in a bit more detail but by simply meeting the following objectives, we can provide a formal mechanism to rationalizing the engineering decisions that we make.

- Declarative Knowledge
- Structured Decisions
- Human still in the loop
- Generate a narrative for explanation and regression (i.e. a *provenance* capability)

**A search optimization problem.** The problem boils down to optimizing among the alternatives considering constraints, requirements, and various measures of effectiveness. Most of these measures either come about through heuristics, analysis models, or simulation of the

alternative being studied. We definitely need an approach that will selectively lock choices to prevent explosion of alternatives<sup>19</sup>.

Take the following case under consideration and you can see how important constraining the design space becomes:

- For a given concept vehicle you will have varying sets of design alternatives for every system component.
- Each of these alternatives will have benefits in terms of some established criteria, such as *accuracy, reliability, cost, ... etc.*
- A ground vehicle system can easily have 200 components with an average of 3 design alternatives per component.
- In this case, the number of evaluations required to find a global optimum with respect to the summed criteria is  $3^{200}$
- This value exceeds the number of atoms in the universe,  $4 \times 10^{78}$  by many orders of magnitude.
- Bottom-line is that a human is still required in the loop to exercise judgment and to prune the computational space. An optimization tool can speed the process in relative terms.

Figure 7.6-45 outlines the optimization architecture that we have prototyped as part of this effort.

Originally, the expert system organization was predicated on a two-stage process. The first stage included heuristics and straightforward calculations (cost lookup, first-order rules, etc). The second stage would feature more elaborate simulations, via connections to external tools. The plan was to eventually allow the second stage outcomes to get adopted as first stage heuristics as our tacit knowledge matures.

Maintaining a two-stage process requires a long-term maintenance commitment as the updates from the external tools have to be periodically translated manually to heuristics as the basis information changes. For that reason, we have elected to consolidate the stages as much as possible. The figure shows the external apps that we have integrated at different times, both commercial and open source, and in Figure 7.6-46, the figure demonstrates how an external tool was hooked up in more detail.

---

<sup>19</sup> A spreadsheet-based approach, although table-driven, is untenable since it lacks : (1) Large-scale maintainability, with the “if-then” rules particularly difficult to implement and (2) Customizable extensibility to outside tools. The latter strongly suggests that flexible reasoners could play a vital role. Interesting to note that, despite decades of development of decision support systems and methodologies, spreadsheets are still popular as primary tools for decision making.

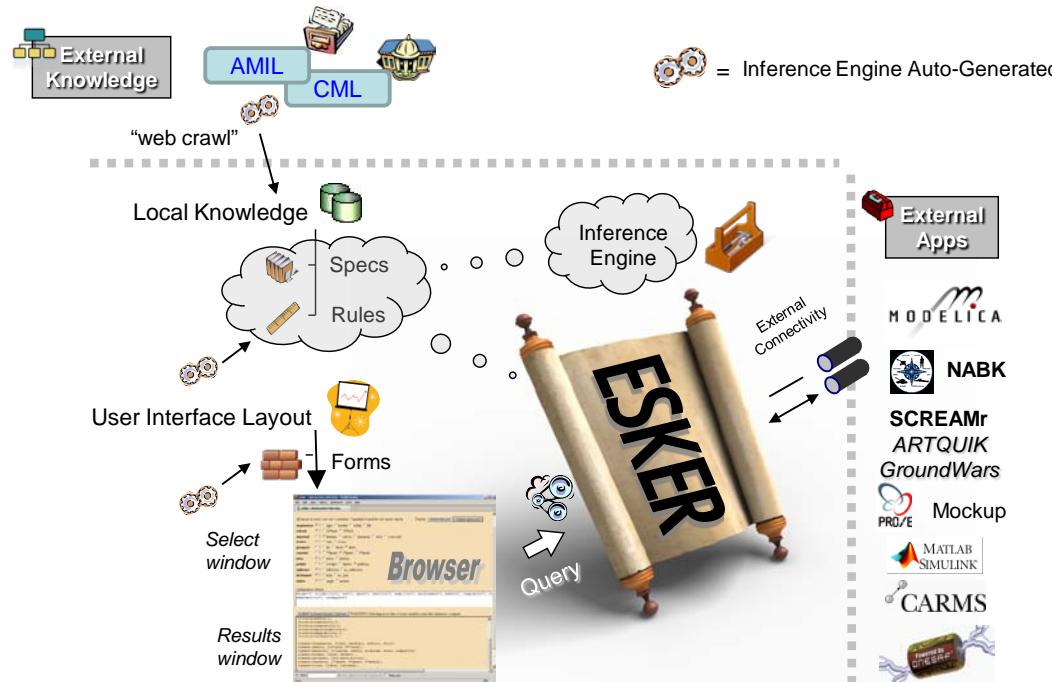


Figure 7.6-45. Architecture of the Optimization Shell

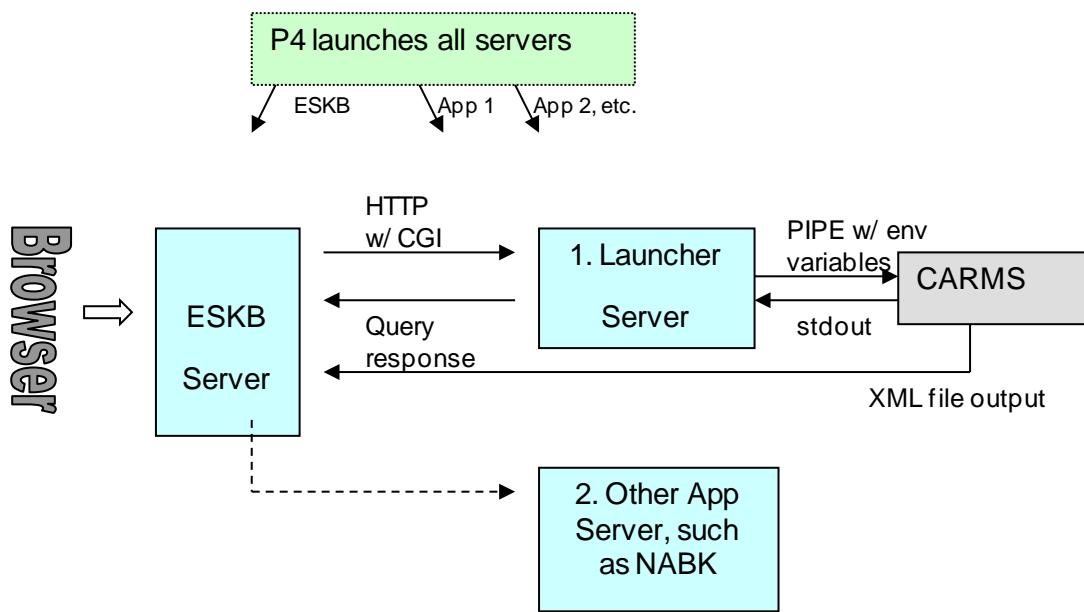


Figure 7.6-46. AMIL-like Connections Between the Main Application and Server Applications

In general, there is no limit to the number of application servers that one could have running as long as they provide a common way of dealing with input and generate a structured format for the output data stream. The use of HTTP with XML and delimited data formats makes this approach workable.

The set of servers can be launched with an execution monitoring tool. A logical launching specification file can be used to generate the locations and ports of the servers. Since the session file uses the same formatting as the rest of the knowledge-base, the specification can be shared with the rest of the rules governing the communication paths. This becomes important considering that the run location of the external simulations is independent of the domain-specific rules used to evaluate alternatives. If the information is shared between the launch configuration and query execution configuration, it makes the system much easier to maintain or to migrate.

To show the proof of concept, we have generated a suite of optimization problems that run the gamut of conceiving spaces, from small to large. This suite also exercises many of the external tools.

**Table 7.6-2. List of Prototype Optimization Problems**

Item	Description
Ground vehicle	Generates a random set of heuristics in a large concepting space, with 23 criteria and over 150 elements with at least two alternatives for each element
Brigade combat team vehicle	A very small model to compare against
Mountain bike test	Consumer application of a buy decision based on component selection
Bradley example w/SCREAMr	Tests out SCREAMr as an external tool
Jeep spare tire example	Tests out CARMS as an external tool
Fire Control NABK example	Tests out NABK as an external tool
Artillery Artquik example	Tests out Artquik as an external tool
Ground Wars example	Tests out GroundWars as an external tool
Matlab Interface example	Connects to a generic Matlab executable

Typically, analysis tools such as the NATO Armaments Ballistic Kernel (NABK) can be used as interfacing elements where the scope of the knowledgebase is not extensive enough. Where enough information is available we can simply declare that in the local knowledgebase. So a simultaneous path involves building a knowledgebase to enable system level trades. Queries to the knowledgebase will therefore execute specific performance analyses, initially via heuristics, and later via external connectivity, but will produce good relative performance metrics in each case. The recommendation is to follow the elicitation steps described in Section 7.6.1.3.2.

### 7.6.2.2.2 Optimization via Expert Systems

The goal is to verify that design concept has been through a rigorous and, even better, a formal analysis for optimality. The proven way to do this is through a set of first-order logical rules. An analyst can accomplish this by carefully considering all the alternatives in his/her mind, but the complexity and dimensionality of the decision space usually precludes this, save for the simplest case of a handful of rules. By applying good decision engineering practices, one can overcome a decision making "complexity ceiling" while maintaining a readable set of rules. Parts of the evaluation can remain in terms of abstracted domain alternatives that exist solely as empirical rules, heuristics, and other analysis results.

The key to making the knowledgebase approach easy to work with is in the methodical practice of presenting all information declaratively. Thus all facts (i.e., data) and rules have clear visibility and rely on symbolic notation. The facts are easily separable from the rules, yet you do not need special file reading mechanisms to get started. The only thing that sets a knowledge-based approach apart from a database query is that the database lacks rules.

**Performance.** Expert systems are usually not compiled. Even though interpreted rules may reduce the performance speed, the sophistication of the decision support algorithm selected likely has a bigger impact on the speed of the search. An optimization approach such as Dynamic Programming is easily accomplished in a shell environment. (Large benefits via such simple dynamic programming mechanisms as tabling and partial evaluation.) If we need even more speed than Dynamic Programming then the rules can be recompiled as an executable. However, compilation is not typically desired since modifying the rules and data is essential for

interactive experiments. This is a wash in terms of what constitutes the best approach – pick either interpreted for short build cycle or compiled for faster results.

**Robustness.** A benefit of a knowledgebase is that the rules and data can combine to perform formal type checking, which makes the rules manageable as the set gets larger. In practice, this is very easy to specify. In the example below is a set of data and corresponding rules that make the type matching very easy to follow as we add the dimensional constraints of horsepower (hp) and metric volume ( $\text{m}^3$ ):

```

engine_volume_per_horsepower('conventional turbo diesel', 0.0006496*m3/hp).
engine_volume_per_horsepower('AIPS diesel', 0.0003398*m3/hp).
engine_volume_per_horsepower('turbine', 0.0*m3/hp).
engine_volume_per_horsepower(_, 0.0*m3/hp).

engine_v_func(HP*hp, Powerpack_Type, Volume*m3) :-
    engine_volume_per_horsepower(Powerpack_Type, Density*m3/hp),
    Volume is HP*Density.

transmission_volume('conventional turbo diesel', 0.98826*m3).
transmission_volume('AIPS diesel', 0.98826*m3).
transmission_volume('turbine', 0.0*m3).

transmission_v_func(HP*hp, Powerpack_Type, GVW*tons, Volume*m3) :-
    transmission_volume(Powerpack_Type, Base*m3),
    Volume is 0.75*GVW/60 + 0.25*(HP/1500)*Base.

```

The declarative pattern matching on the caller and callee sites make it impossible to generate an incorrect mixture of dimensions.

### 7.6.2.2.3 Specifying the search problem

**Specifying the alternatives.** Say that we boil down the problem into a simpler set of data and rules for the sake of explanation. The following is an enumerated set of variants that we wish to consider. Each *variant* (or *element*) has its own set of alternatives; for example the `traction` on a vehicle could be either `wheeled` or `tracked`.

```

variant(traction, [wheeled,
                   tracked]).

variant(number_of_wheels, [6*wheels,
                           8*wheels,
                           10*wheels]).

variant(crew_size, [2*men,

```

```

3*men,
4*men]).

variant(max_speed, [40*mph,
50*mph,
60*mph]). 

variant(engine, [diesel,
hybrid]). 

```

These actually thinly disguised triple stores where the object list is enclosed by brackets [ ], to cut down on the verbosity. So the notional triple store is (subject, hasVariant, object).

**Weighing the alternatives.** Next, we can specify the outcomes of physical weight for each of the alternatives. We have some flexibility here as the weight of the wheels can get rolled up in the other rule. The engine rule is expressed as a simple fact.

```

weight(wheeled, W) :- 
    variant(number_of_wheels,L),
    member(N*wheels,L),
    W is N*100+100.

weight(tracked, W) :- 
    variant(number_of_wheels,L),
    member(N*wheels,L),
    W is N*90+1000.

weight(N*men, W) :- 
    variant(crew_size, L),
    member(N*men,L),
    W is N*100.

weight(_*wheels, 0).
weight(_*mph, 0).
weight(diesel, 2000).
weight(hybrid, 1500).

```

A similar mix of rules and facts can be assembled for the power evaluation (below). Note that the power scales linearly with the maximum speed desired, independent of the engine type. Yet the idling power differs for `diesel` and `hybrid`. This set of rules can be easily changed so the max speed power distinguishes the two correctly.

```

power(wheeled, P) :-  

    weight(wheeled,W),  

    P is W*10.  
  

power(tracked, P) :-  

    weight(tracked,W),  

    P is W*12.  
  

power(N*men, P) :-  

    weight(N*men,W),  

    P is W*5.  
  

power(_*wheels, 0).  

power(N*mph, P) :- P is N*1000.  

power(diesel, 10000).  

power(hybrid, 15000).

```

**Optimizing the alternatives.** The final ingredient is to set up some evaluation rules and optimization criteria. This is actually a very compact algorithm as the data rules that we have declaratively specified accomplishes most of the heavy lifting. The expert system sweeps through the set of variants in search of a subset that best meets the optimization criteria we have selected. In this case, we want to maximize the combine weight and power according to a weighting function. This essentially gives the worst case solution. The best case would involve searching for a minimum. The following set of rules is an early version that gives an idea of the recursion involved.

```

:- dynamic(max/2).  

:- dynamic(stored/1).  
  

max(0.0, empty_set).  
  

criteria(W,P,T) :-  

    T is W*1.5+P*1.2.  
  

one_of_each([],[]).  

one_of_each([H|T],[H1|Ts]) :- member(H1,H), one_of_each(T,Ts).  
  

sum_terms(Goal, [],Total,Total).  

sum_terms(Goal, [Item|Rest],Sum,T) :-  

    sum_terms([Item|Rest],Sum1,T), Sum is Sum1 + Item.

```

```

call_with_args(Goal, Item, P),
Total is Sum+P,
sum_terms(Goal, Rest, Total, T).

check_maximum(T, Set) :-
    max(Total, Current),
    T > Total,
    retract(max(Total, Current)),
    asserta(max(T, Set)).

optimize(_, Set) :-
    findall(V, variant(_, V), All),
    one_of_each(All, Set),
    sum_terms(power, Set, 0.0, P),
    sum_terms(weight, Set, 0.0, W),
    criteria(W, P, T),
    check_maximum(T, Set),
    fail.

optimize(Value, Which) :-
    max(Value, Which).

```

The rule named `optimize` recursively evaluates the alternatives by automatically selecting a combination of one member from each variant and then evaluating the criteria, and finally checking for the maximum to meet the criteria we have established. Once the set is exhausted, the alternative `optimize` rule gets evaluated and it returns the `max` value and `which` elements of the subset it contains.

```

optimize(Value, Which)?
Value = 125460.0
Which = [tracked,6*wheels,4*men,60*mph,hybrid]

```

Or we can constrain one of the outputs (`traction = wheeled`) and find alternative values from the built-in pattern matching. No new programming is required due to the backward-chaining nature of the symbolic processing. This result of such a query results in an alternate lower value result subject to the new constraint:

```

optimize(Value,[wheeled,N,Men,MPH,Engine])?
Engine = hybrid
MPH = 60*mph
Men = 4*men
N = 6*wheels

```

```
Value = 110100.0
```

This specific example turns out fairly trivial as this set derives from each of the heaviest components available. In a real situation, the actual set of rules will become more complex and the criteria for setting an optimum configuration will become less obvious to the eye. In certain cases, the discrete choices may be expanded along a finite domain. In that case, a Finite Domain (FD) solver comes out of the Prolog box and it allows the expert system to prune the search space more efficiently.

#### 7.6.2.2.4 Issues in Optimization

The looming issue that confronts us is how best to manage the set of concept alternatives. It's a given that to formally prove that you have reached an optimum or maximum with respect to some measure, that you have to sweep through *all* of the alternatives. Techniques exist that can seek out local minimum or maximum (via gradient search, etc) yet these do not guarantee a global extreme value. Not every problem is convex. So the options are to either scan exhaustively or to search selectively and perhaps stop when some criteria is met<sup>20</sup>.

To provide an example, say that we have 14 variant functionalities which contain 3 alternatives each. The process of finding a peak exhaustively amongst this set requires  $3^{14}=4,782,969$  unique selection to be evaluated (subject to all the values having an independent effect on the solution, otherwise this number can be reduced). This rather modest set is perfectly acceptable to evaluate on a stock computer. Yet, what path should we take when we provide 20 variants instead of 14 and find that the number jumps to  $3^{20}=3,486,784,399$ , or if we add an another alternative to each of the variants, then it goes to  $4^{20} \sim 1$  trillion computations? This results in excessive computational complexity as the search combinatorially explodes.

The combinatorial space on the table below is  $2^{60} * 3^{14} * 4^3 = 3.5*10^{26}$  for a total of 77 element classes. Interesting to consider that two valid options are to either have a feature or *not* have it. This can grow the space greatly even when variants are not considered.

Element Class	Variant 1	Variant 2	Variant 3	Variant 4
traction	tracked	wheeled		
wheels	4*4	6*6	8*8	10*10
power_train	mechanical	hydraulic	hydro_mechanical	
troops	4*soldiers	6*soldiers	8*soldiers	10*soldiers
turret	manned	unmanned		
gun	gun_a	gun_b		
battery	lead_acid	hybrid_battery		
engine	diesel	turbo_diesel	hybrid_electric	
final_drive	3*ratio	4*ratio		
loader	auto_loader	manual_loader		
range_device	laser_ranger	lidar_ranger		
crew	1*crew	2*crew	3*crew	

<sup>20</sup> The technique of simulated annealing via Monte Carlo sampling is often used for this purpose

Element Class	Variant 1	Variant 2	Variant 3	Variant 4
armor	armor_a	armor_b		
fuel_tank	200*gallons*fuel	300*gallons*fuel		
afes	crew_compartment	weapons_compartment	engine_compartment	all_compartments
width	2.5*m*width	3.0*m*width	3.5*m*width	
height	3.5*m*height	4	4*m*height	5*m*height
length	7*m*length	8*m*length	9*m*length	
chassis	material_1	material_2		
kit	optional_kit	no_kit		
clearance	0.25*m*clear	0.5*m*clear	1*m*clear	
payload	800*rounds	1000*rounds	1200*rounds	
water	bottled	synthesized		
vision	direct	indirect		
hatch	manual_hatch	no_hatch		
coolant	30*gallons*coolant	40*gallons*coolant	50*gallons*coolant	
voltage	15*v	28*v		
high_voltage	270*vdc	500*vdc		
skirt	with_skirt	without_skirt		
horsepower	800*hp	1000*hp	1200*hp	
oil_capacity	15*gallons*oil	20*gallons*oil	25*gallons*oil	
fuel_type	diesel_fuel	jet_fuel		
engine_displacement	15*liters	20*liters	25*liters	
suspension	suspension_a	suspension_b		
generator	primary	with_backup		
crew_stations	1*station	2*station		
compartment	isolated	non_isolated		
ignition	laser_ignite	other_ignite		
egress	easy_out	hard_out		
ingress	easy_in	hard_in		
software	500000*loc	1000000*loc		
firmware	firm	no_firm		
auxiliary_power_unit	primary_apu	backup_apu		
drive_by_wire	dbw	no_dbw		
night_vision	night	no_night		
secondary	20*caliber	50*caliber		
prognostics	onboard	offboard		
diagnostics	software_diagnostic	bit		

Element Class	Variant 1	Variant 2	Variant 3	Variant 4
gps	gps	no_gps		
ins	ins	no_ins		
iru	iru	no_iru		
fire_control	reuse_fire_control	custom_fire_control		
powerpack	fixed_powerpack	removable_powerpack		
pmcs	pmcs	no_pmcs		
ietm	ietm	no_ietm		
power_distribution	networked_power	fixed_power		
periscopes	viewer	windshield		
crew_position	forward_crew	aft_crew		
weapon_view	weapons_camera	no_weapons_camera		
obstacle_avoidance	ans	no_ans		
gun_stow	forward_stow	reverse_stow		
route_follow	ans_follow	guided_follow	manual_follow	
methods	los	blos		
emplACEMENT	aided_emplace	no_emplace		
planning	offboard_plan	onboard_plan		
scheduling	auto_schedule	no_schedule		
local_security	crew_assisted_security	auto_security		
touch_screens	touch	bump		
yoke	joystick	wheel		
noise_suppression	noise_suppress	no_noise_suppress		
sleeping_provisions	sleep	no_sleep		
heads_up_display	hud	no_hud		
resupply_mode	auto_resupply	manual_resupply	pda_resupply	
towing	tow_provisions	no_tow		
pivot_steer	pivot	no_pivot		
amphibious	amph	no_amph		
driver_redundancy	driver_backup	no_driver_backup		
signature	suppression	no_suppression		

So the implications for optimization are significant if we can't control the state space. This also holds for sensitivity analysis. In the case of the most rudimentary analysis, the variants should include a subset of lower-range, nominal, and upper-range values. This will trend at least according to  $3^N$ . So that if we combine concept alternatives with sensitivity alternatives the sweep volume can grow unmanageable.

**Limiting Scope.** The explosion of states has always been the problem with DSE. Clearly we can reduce the space by constraining most of the states to nominal values and then varying the

rest. The Catch-22 in all this is that if we knew that we could optimize by adjusting one variable while keeping all the other variables constant, this would also mean that we actually knew what the optimal frontier was before we got started. Yet if we knew the optimal region initially, we wouldn't need the assistance to begin with! In this case, intuition provides all the optimization we would ever need, and extra sweeping is overkill. All we are doing is testing the optimization gradient around a minimum or maximum of some criteria. It wouldn't take that long to do this by hand. So the issue boils down to whether we think we will ever have sufficient intuition to arrive at an efficient optimization frontier without doing at least some sweep searching.

Many of the complex calculations can reduce easily to interpolations over a design space. Calculating recoil and the interactions with stroke length and volume displacement is a good case in point. Everything about the calculation is deterministic so you may need to do it for a number of configurations, and then interpolate as necessary.

A DSE tool will not always reconcile certain design criteria such as lethality and survivability with all the other criteria that are missing, chief among them reliability and cost. Reliability is potentially even more challenging than the Newtonian analysis because it consists of probabilities, and interpolations don't work for probabilities. And cost can be a subjective criteria. So collectively the only way around this is to make the criteria and alternatives fairly uniform, and to often keep them mushy at a heuristic level. So we have to essentially normalize the rules to promote fair-play. If we ever get a situation that one calculation causes an execution bottle-neck on the order of seconds, the total execution time will grow quickly.

When a logic program generates results or makes calls to external tools, it can selectively apply the concepts of Dynamic Programming to avoid spending too much time re-computing unnecessarily. This will lay out all the design choices made with respect to some measures of optimality, and further allows one to control the complexity and performance constraints of the search space. So, if we ever get into a bind where we have to trade-off too much computation against the use of mushy heuristics, we will have a path laid out.

**The Role of Heuristics and Partial Calculations.** The issue comes up that we may have to iterate to determine some optimal criteria, and the times it takes becomes the basis for when to use dynamic programming and tabling techniques. The example given is determining the number of wheels needed to support a vehicle weight. Since the wheels themselves will add to the weight then you might naively imagine that some iteration needs to be performed to calculate the optimal configuration. In fact, this is plain linear algebra that can be worked out instead of using some iteration scheme.

#### 7.6.2.2.5 Prototyping

For a knowledgebase that consists of 23 criteria and 78 design elements with between 2 and 3 alternative designs each, we generated about 4,000 rules to test against. That set of data is too large to be able to run a full optimization, so for testing we ran in a partial optimization setting before it started to execute too slowly. This is always a combinatorial problem so once it hits a threshold the slowdown is apparent.

The majority of the design elements were left fixed, so as an example we take only 10 of the elements as unknown alternatives. This query took less than 20 seconds in *gprolog* to execute in an interpreted shell, and less than 10 seconds in a compiled shell. For only 2 or 3 unknowns it

runs in well under 1 second. Adding more sophisticated rules won't add much to the running time, so unless it invokes external calls, the response time should be tolerable.

As an illustration of how sensitive the optimization performance is, we used a different expert system shell, **eclipse**, and it returned the same answer for 10 unknown variables but it took only 1.5 seconds instead of 18 seconds. With this shell it can optimize for 16 unknowns in 95 seconds before the combinatorial blow-up occurs.

Also any constraints and requirements that will eventually be added should shorten the running time by pruning the results. The UTRC META team apply constraint pruning as a valid approach to reducing the state space. Working smart and dealing with compartmentalized sets as described in section 7.6.1.2.6 is the solution to combinatorial problems.

#### 7.6.2.2.6 Example of Optimization Query input

We initially prototyped an HTML page for the inputs. Once this was done, the data within the page was auto-generated from the knowledge-base so that it does not require extra coding besides the HTML boilerplate and some JavaScript and CGI.

The basic structure for the knowledgebase resides in a "**vehicle-kbase.pro**" file which consults three other files (1) "**engine.pro**" which contains the inference engine, (2) "**vehicle-specs.pro**" which contains the schema spec for criteria and elements, and (3) "**vehicle-rules.pro**" which contains the set of rules. A "**vehicle.xml**" main page pulls together the boilerplate form entry from an XML stylesheet and specific form elements in "**vehicle-form.html**". As default, it uses a generic web server and gprolog for evaluation.

The following is an example of an optimization query input form. For each element, a set of radio buttons is provided to allow one to select from a list of alternatives. If the ? radio button is selected, this is an unbound choice and it will be optimized with respect to the criteria represented in Figure 7.6-47. According to the rules in the expert system, all inputs in lower case are constraint and capitalized variables are unbound inputs and will generate a query output. This form was generated automatically from the knowledgebase specification and an XML stylesheet template.

**Optimization Selection query : ground\_vehicle**

All inputs in lower case are constraints. Capitalized variables are query outputs.

Display [ground\_vehicle-rules.pro]  
Display [ground\_vehicle-specs.pro]

<b>traction</b>	<input checked="" type="radio"/> ? <input type="radio"/> tracked <input type="radio"/> wheeled
<b>wheels</b>	<input type="radio"/> ? <input checked="" type="radio"/> 4*4 <input type="radio"/> 6*6 <input type="radio"/> 8*8 <input type="radio"/> 10*10
<b>power_train</b>	<input checked="" type="radio"/> ? <input type="radio"/> mechanical <input type="radio"/> hydraulic <input type="radio"/> hydro_mechanical
<b>troops</b>	<input type="radio"/> ? <input checked="" type="radio"/> 4*soldiers <input type="radio"/> 6*soldiers <input type="radio"/> 8*soldiers <input type="radio"/> 10*soldiers
<b>turret</b>	<input type="radio"/> ? <input checked="" type="radio"/> manned <input type="radio"/> unmanned
<b>gun</b>	<input type="radio"/> ? <input checked="" type="radio"/> gun_a <input type="radio"/> gun_b
<b>battery</b>	<input type="radio"/> ? <input checked="" type="radio"/> lead_acid <input type="radio"/> hybrid_battery

Figure 7.6-47. Part of the Prototyped HTML ESKER Query Interface

### 7.6.2.2.7 Elicitation Table

The objective is to come up with an optimal configuration of design elements based on objective criteria. These criteria could be constraints, requirements, or quantitative measures of effectiveness.

Fundamentally, there is little difference between a constraint and a requirement. The only distinction that matters is that a constraint has a direct connection to the reality of the environment around us, whereas a requirement is an artificial gauge of what needs to be done. Thus a constraint can be considered an environmental requirement.

Constraints		Requirements	
Air Drop	BLOS	Target Cross Section	
Tunnel Width	Resupply	Upgrade P3I	
Gun	Mine	HP/Weight	
Elevation	Turn Radius	Op Temp	
Gun Traverse	SROF	Un - deploy	
Wall	Deploy	MROF	
Obstacle	First Shot Out	Cost	
Side Slope	Remote	Weight Curb + Loaded	
Clearance	Survive	Transport	
Ford	MPG	Response Time	
Runway Distance	Highway Speed	X - Country Speed	
Atmosphere	Temperature	MTBF MSCP	
Pressure	Wind	Accuracy Moving	
Roll	Latitude	Accuracy Stationary	
Pitch	Humidity		
Gap Obstacle			
Latitude			
Humidity			
Wind			
Temperature			
Soil			
Terrain			
Mission Time			
Red Force Range			

All the quantitative measure of effectiveness need to be aggregated and constrained to the requirements. Each design element or part on board the system contributes to some effectiveness measure (otherwise it should not be on board the vehicle) and these need to be quantitatively evaluated. In certain cases, a qualitative intangible needs to be included. The following PartCrit table (Table 7.6-3) serves as a checklist to determine which elements we need to include.

**Cells:** A part will either be applicable or not-applicable with respect to some optimization criteria. So during the elicitation stage of knowledge engineering, we need to place an **X** in the cell if it is measurable. If it is not measurable or doesn't apply, we place a—or **N/A**. If something can be handled by an outside tool, then we can indicate that within a cell by **S** or some other marker. If it is indirectly used by another cell, then **I** indicates an influence relationship.

**Part List:** If no credible alternatives exist for a part, then there is no use compiling the measure for that part as it is an invariant with respect to the other measures. So we need at least two numbered alternatives for every part. It is also possible that the alternative is not to include such a part; that would then give the two required alternatives. Qualitatively, if a row is completely solid corresponding to a part, then that element is likely very critical across all criteria.

**Criteria List:** Looking at the criteria, if a column is sparse after analysis, then that particular optimization criteria is very narrowly applied to certain design elements. For example,

trainability may prove to have a minimal impact on an optimal solution, as it will only draw from specific elements.

This table does not tell the whole story. For example, the causal structure and effects implied by how the design elements fit together needs to be taken care of by rules added to the knowledgebase. The rules therefore will generate the quantitative measures that we will optimize against, and the requirements that we need to comply with. The reason that we do not extend this table into a spreadsheet is that the rules are unwieldy if kept to within the confines of a cell. The requirements also will cause this table to grow to a third dimension. In other words better abstraction mechanisms are available, such as the influence diagrams described in 7.6.1.2.6. We use this table only for elicitation of optimization criteria.

**Table 7.6-3. Elicitation Table**

**Design Elements with Optimization Criteria**

Part \ Crit	Power	Weight	Reliable	Lethal	Survive	Transport	Sustain	Cost	Speed	Mobility	Fuel	Usability	Performance	Volume	Safety	Complexity	Common	Maintain	Logistic	Adapt
Traction																				
1. Tracked	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. Wheeled																				
Wheels																				
1. 4x4																				
2. 6x6	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
3. 8x8																				
4. 10x10																				
Power Train																				
1. Mechanical	x	x	x	-	x	-	x	x	x	x	x	-	x	x	x	x	x	x	x	
2. Hydraulic																				
3. Hydro-Mechanical																				
Troops																				
1. 4																				
2. 6	x	x	-	-	x	-	-	x	x	x	x	-	x	x	x	x	-	-	-	
3. 8																				
4. 10																				
Turret																				
1. Manned			x	s	s	x	-	x	x	x	-	x	x	x	x	x	x	x	x	
2. Unmanned																				
3. Fore/Aft																				
Gun																				
1. Single	x	x	x	s	s		x	x	-	-	-	x	x	x	x	x	x	x	x	
2. Double																				
Battery																				
1. Lead Acid	x	x	x	-	x	-	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. Hybrid																				

Engine																									
1. Diesel	x	x	x	x	x	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. Hybrid																									
Final Drive			x	x			-		x	x	x	x			x							x	x	x	
1. 4:1																									
Loader																									
1. Auto	x	x	x	x	x	-	x	x	x	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. Manual																									
Ranging																									
• Laser																									
• PTS																									
Guidance																									
1. Yes																									
2. No																									
Crew																									
1. 1	x	x	x	x	x	x	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. 2																									
3. 3																									
Fuel Tank																									
1. 300 gallon	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. 200 gallon																									
AFES																									
1. Crew																									
2. Weapons	x	x	x	-	x	-	x	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
3. Engine																									
4. All																									
Width																									
1. 3 meters		x																			x				
Part \ Crit	Power	Weight	Reliable	Lethal	Survive	Transport	Sustain	Cost	Speed	Mobility	Usability	Performance	Volume	Safety	Complexity	Common	Maintain	Logistic	Adapt						
Height		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
2. 4 meters																									
Length		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
1. 8 meters																									
2. 7																									
Chassis		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
3. Metal 1																									
4. Metal 2																									
Kit		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
1. Optional																									
2. None																									
Clearance		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
1. 0.5 meter																									

Payload		x		S	S	x	x	x	x	x	x	-	x	x	x	x			x
1. 100 rounds																			
Water																			
1. Refilled	x	x	x	-	x	x	x	x	-	-	x	x	-	x	x	x		x	x
2. Generated																			
Vision																			
1. Indirect	x	x		x	x	-	-	x	x	x	x	x	-	x	x		-	-	x
2. Direct																			
Hatch																			
1. Yes		x	x	x	x	x	-	-	x	x	x	x	x	x	x	-	x		x
2. No																			
Coolant			x	x	-	-	-	x	x	-	-	x	-	-	-	x	-	x	x
1. 40 gallons																			
Voltage																			
1. 28 VDC	x	x	x	-	-	-	x	x	-	-	x	-	-	-	x	x	x	x	-
High Voltage																			
1. 270 VDC	x	x	x	-	-	-	x	x	-	-	x	-	-	-	x	x	x	x	-
Skirt																			
1. With		x		-	x	x			x		x	x	-	x	x	-	-	x	x
2. Without																			
Horsepower		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-
• 1000 hp																			
Oil Capacity		-	x	x	-	-	-	x	x	-	-	-	-	-	x	x	x	x	-
• 20 gallons																			
Fuel Type			x		x	-	x		x	x	x	x	-		-	x	-	x	-
1. Diesel																			
Displacement			x	x		x	-	-	x	x	x	x	-	x	x		-		-
1. 20 liters																			
Generator		x	x	x	x	x	x	-	x	x		x	x		x	x	x		x
1. 1+1																			
Crew stations			x	x	x	x	x	-	-	x	-	x	-	x	x	x	-	x	x
1. 2																			
2. 3																			
Compartment																			
1. Isolated	x	x	x	-	x	x	-	x	-		-	x		x	x	x	-	x	x
2. No iso																			
Ignition		x	-	x	x	x	-	-	x	-	-	-	-	x	x	-	x	x	-
1. Laser																			
Egress			x		-	x	-	x	x	-	-	-	-	x		x	x	x	-
1. Easy																	-	x	x
2. Hard																			
Ingress			x		-	x	-	x	x	-	-	-	-	x		x	x	x	-
1. Easy																	-	x	x
2. Hard																			
APU		x	x	x	x		-	x	x	x	-	x	x		x	x	x		x
1. 1																			

Part \ Crit	Power	Weight	Reliable	Lethal	Survive	Transport	Cost	Speed	Mobility	Fuel	Usability	Performance	Volume	Safety	Complexity	Common	Adapt	Logistic	Maintain
Drive by wire																			
1. Yes	x	x	x	x	x	-	x	x	x	x	x	-	x	x	x	x	x	x	x
2. No																			
Night Vision																			
1. Yes	x	-	x	x	x	-	x	x	x	-	x	x	x	x	x	x	x	x	x
2. No																			
Secondary																			
1. 20 cal	x	x	x	x	x	x	-	x	-	x	x	x	x	x	x	x	x	x	x
2. ?																			
Prognostics																			
1. On-board	x	-	x	-	x	-	x	x	-	-	x	-	x	x	x	x	x	x	x
2. Off-board																			
Diagnostics																		x	x
1. Software	x	-	x	-	x	-	x	x	-	-	x	-	x	-	-	x	x	x	x
GPS																			
1. Yes	x	x	x	x	x	-	x	x	x	x	x	x	x	x	x	x	x	x	x
INS																	x	x	x
1. Yes	x	x	x	x	x	-	x	x	-	x	x	x	x	x	x	x	x	x	x
IRU																		x	x
1. Yes	x	x	x	x	x	-	x	x	-	x	x	x	x	x	x	x	x	x	x
Fire Control																			
1. Reused	x	-	x	x	x	-	x	-	x	-	x	-	x	-	x	x	x	x	x
2. Common																			
3. Custom																			
Powerpack																			
1. Fixed	x	x	x	-	x	x	x	x	x	-	x	-	-	-	x	x	-	x	x
2. Removab																			
PMCS																		x	x
1. Yes	x	-	x	-	x	-	x	x	x	-	-	-	-	-	x	-	-	x	x
IETM / EPSS																		x	x
1. Yes	x	-	x	-	x	-	-	x	x	-	-	-	-	-	x	-	-	x	x
Power Dist																			
1. Networked	x	x	x	-	x	-	-	-	x	-	-	-	-	-	x	x	x	-	x
2. Fixed																		x	x
Periscopes																			
1. Viewer		x	x	x	x	x	-	-	x	x	x	x	x	x	x	x	x	x	x
2. Windshield																			
Crew position																			
1. Turret		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	-	x
2. Hull																		x	x
Weapon View																			
1. Camera	x	x	x	-	x	-	-	x	x	-	-	-	-	-	x	-	x	x	x
2. None																		x	x

#### **7.6.2.2.8 Demo Configuration**

The vehicle ramp design model of Figure 7.6-9 was used as a demonstration. The communication between ESKER and external applications is through either web services or external AMIL stores. Knowledge is served from local Prolog facts, AMIL immediate stores, or through ontological queries. The latter can be extended to web-services via OWL-S, SSWAP, or customized services.

The element set and optimization criteria were kept to a minimum to show the interactions most clearly. The term *element* was used instead of *variant* for the alternatives, and the criteria were equally weighted at the top-level.

% Set of triple-stores to reason against

%

```

criteria(power,1).
criteria(weight,1).
criteria(survivability,1).

element(ramp_thickness, [0.5*inch,1*inch,2*inch]).
element(ramp_motor_size, [rating_168, rating_268]).
```

The rules were then expressed by cross-coupling the individual criteria to the current set of alternatives, transcribing the influence diagram of Figure 7.6-9 into logical rules Figure 7.6-1:

```

% AMIL Nodes to Model Library
% -----
calculate_weight_of_ramp(Thickness, Weight) :-
    % Replace this with URL to ProE
    Weight is Thickness*1000. %% TODO: Placeholder value from model

calculate_survivability_of_ramp(Thickness) :-
    % Replace this with URL to Armor model
    Thickness is 4. %% TODO: Placeholder value from model

% Generic Rules
% -----
constraint(Rule, 1) :- call(Rule), !.      % general constraint rule giving unity weighting if TRUE
constraint(_, 0).

% Constraints
% -----
weight_rating(rating_160, 5500).          % Motor rated 160 has a smaller weight handling capability
weight_rating(rating_260, 11000).
weight_margin(100).                      % Provide a weight margin for soldiers, too low 1st go round
                                         % Then up the number for second go round

% Element Rules
% -----
power(ramp_thickness:_, _:X) :- X is 0.          % example of a don't care, i.e. N.A.
power(ramp_motor_size:Motor,L:X) :-
    member(T*inch, L),                         % which motor to use for ramp
    calculate_weight_of_ramp(T, W),             % depends on ramp weight selected
    weight_rating(Motor, Weight_Limit),
    weight_margin(Weight_Margin),
    constraint((W + Weight_Margin) < Weight_Limit), X). % Only allow values that meet constraints

weight(ramp_thickness:T*inch, _:X) :-           % Weight of ramp from parametric ProE model
```

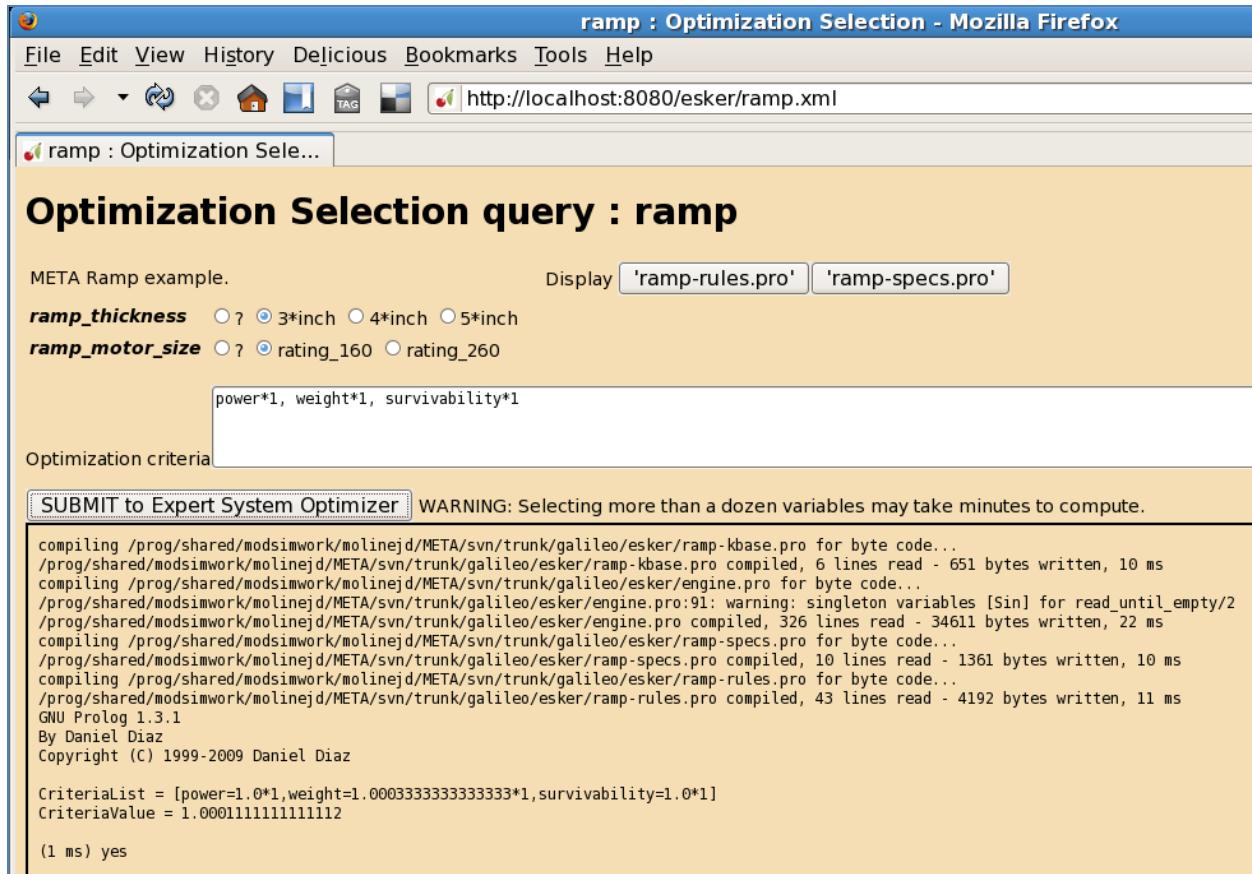
```

calculate_weight_of_ramp(T, Weight),
X is 1/Weight.                                % For optimizing, reduced weight is increasing
weight(ramp_motor_size:rating_160,_:1).        % Simple lookups for motor weights
weight(ramp_motor_size:rating_260,_:0.5).

survivability(ramp_thickness:T*inch,_:X) :-      % Survivability of ramp from armor model
    calculate_survivability_of_ramp(Thickness),
    constraint(T > Thickness, X).                % Only allow values that meet constraints
survivability(ramp_motor_size:rating_160,_:1).    % The two motors are equally survivable.
survivability(ramp_motor_size:rating_260,_:1).

```

The placeholders for external calls were left in place and the initial set was evaluated within a web-served engine.



**Figure 7.6-48. Web-Served Version of ESKER Used to Populate the Knowledge Base**

The version of ESKER used for the demonstration was recompiled into a Java server executable via the *tuProlog* Java class library and evaluation data was retrieved through AMIL stores, either immediate or externally computed.

This proved the concept of embedding AMIL into a design space exploration reasoner. The conversion to an ontologically-friendly engine is trivial, as the essential data elements are all based on triple-stores.

**DSE in practice.** A generic DSE starts from an arbitrary set of top-level design elements that we are interested in, along with a few decision criteria and weighting factors. Then an automated User Interface (UI) builder pulls out the element variants from CML and generates the place-holders for rules. This would be the kick-start for having a limited crowd-source of engineers to fill out decision rules that calculate criteria values.<sup>21</sup> Eventually, the rules can go back into the CML repository as weighted contracts that can get reused.

decision criteria = weighted contracts

Thus, we generate contracts that are not iron-clad but that have weighting attached to them that allow components to be evaluated for fitness-of-purpose and down-selecting.

### 7.6.2.3    Ontology-based Logic Reasoners

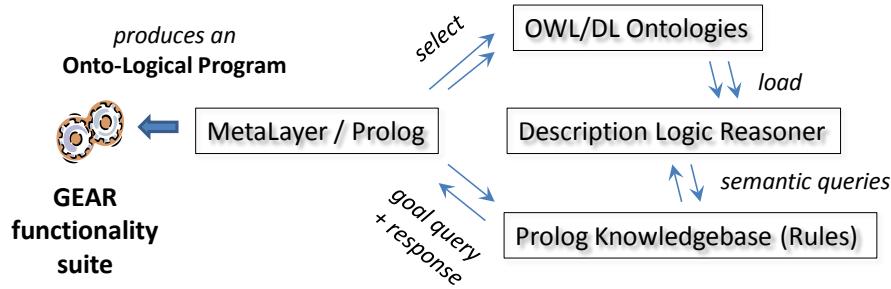
The ESKER application provides an example of a design space exploration reasoner. The general utility of reasoners lies in their applicability to a variety of engineering problem-solving areas. This section demonstrates several examples of the ontological pattern that we initially formulated in Section 7.6.1.4.2.

The uniform approach we take is illustrated in Figure 7.6-49. The pattern is one of importing a set of ontologies, applying a description logic reasoner API to extract the triple-store data, and then applying further reasoning through customized Prolog rules. The top meta-layer maintains a clean and transparent separation between ontological data and the local knowledge contained within the reasoner.

---

<sup>21</sup> This beats a single spreadsheet that is under the control of a single person.

# GEAR Logical Programming Architecture



The MetaLayer consists of interpreted Prolog rules that understands interactions with the ontology namespaces and the semantic reasoner.

**Figure 7.6-49. Ontological Reasoners Follow a Similar Pattern of Encapsulating the Deeper Semantic with a Uniform Logical Front-end**

Because of the overall similarity among all of these reasoners, we gather them together as the GEAR suite, useful for a knowledge engineering framework.

### 7.6.2.3.1 Weight, HorsePower, Speed Example

This is a more extended example with declarative semantics, i.e. we can calculate whether a specific engine with weight and horsepower rating will meet a specific speed criteria. This rule embeds aerodynamic drag to first order but could also include rolling resistance

Figure 7.6-50 shows a mock-up ontology written in N3 and the equivalent knowledgebase that would generate.

```

@prefix : <http://wcsn262:8001/backup/c.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
schema#> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
.

:           a      owl:Ontology .
:Engine     a      owl:Class .
:hasWeight  a      owl:DataProperty .
:hasHorsePower a    owl:DataProperty .
:Chassis    a      owl:Class .
:hasAero    a      owl:DataProperty .

:engine1    a      :Engine,
              owl:NamedIndividual ;
:hasWeight  1.0 ;
:hasHorsePower 1000.0 .

:engine2    a      :Engine,
              owl:NamedIndividual ;
% c:'Engine'(engine1).
% c:'Engine'(engine2).
% c:'Engine'(engine3).
% c:'Engine'(engine4).
% c:'hasWeight'(engine1, 1.0).
% c:'hasWeight'(engine2, 1.1).
% c:'hasWeight'(engine3, 1.2).
% c:'hasWeight'(engine4, 1.3).
% c:'hasHorsePower'(engine1, 1000.0).
% c:'hasHorsePower'(engine2, 1100.0).
% c:'hasHorsePower'(engine3, 1200.0).
% c:'hasHorsePower'(engine4, 1300.0).
% c:'Chassis'(nominal).
% c:'hasAero'(nominal, 234.0).

```

**Figure 7.6-50. N3 Ontology for Engine and the Equivalent “invisible” Triples They Represent (To the Right)**

The rules operating on this ontology are shown below. This uses declarative semantics to the effect of providing the same interface for querying a horsepower given a speed criteria as for querying a speed given a horsepower criteria.

```

find_engine_with_criteria(Eng,HP,50.0,1.0,vehicle(8000.0,1.0),context(F,A)).
find_engine_with_criteria(Eng,80.0,V,1.0,vehicle(8000.0,1.0),context(F,A)).

```

```
%import http://wcsn262:8001/backup/c.owl
%import http://wcsn262:8001/backup/sweet.owl

%%% Query with unknown HP but desired speed, V, or unknown speed but given HP
% find_engine_with_criteria(Eng,HP,50.0,1.0,vehicle(8000.0,1.0),context(F,A)).
% find_engine_with_criteria(Eng,80.0,V,1.0,vehicle(8000.0,1.0),context(F,A)).

calculate_horsepower(Speed,EffWind,AeroFactor,Weight,EffG,HorsePower) :-
    nonvar(Speed),
    SpeedWithWindDrag is Speed + EffWind,
    Drag is (SpeedWithWindDrag*AeroFactor)**3.0,
    HorsePower is (Drag+EffG)*Weight.

calculate_horsepower(Speed,EffWind,AeroFactor,Weight,EffG,HorsePower) :-
    var(Speed),
    Drag is HorsePower/Weight - EffG,
    SpeedWithWindDrag is (Drag ** (1.0/3.0))/AeroFactor,
    Speed is SpeedWithWindDrag - EffWind.

find_engine_with_criteria(Engine,HorsePower,Speed,Margin,
                           vehicle(Weight,Aero), context(Region,Time)) :-
    sweet:'Gravity'(Gravity),
    sweet:'gravityValue'(Gravity, G),
    sweet:'Region'(Region).
```

**Figure 7.6-51. Weight, Horsepower, Speed Reasoner**

#### 7.6.2.3.2 CML Example

This example shows interactions with CML ontology. We invoke a query to the CML repository to find all models associated with a given component. This uses a built-in rule called *findall* which collects the (component, model) pairs and then prints out the results as a side-effect.

If we run the goal query: *print\_all\_components\_with\_hifi\_model?*

This results in the output:

```
[cfv,'CFVRamp2'] has a [cfv,'CFVRampSimulinkModel']
[cfv,'CFVRamp1'] has a [cfv,'CFVRampSimulinkModel']
```

```
%import http://wcsn262:8001/demo/arrow.owl
%import http://wcsn262:8001/demo/CFV.owl
%import http://wcsn262:8001/demo/meta.owl

%% Rule to find a component with a hifi model, this searches Meta for possible model
%% classes and Arrow to make modeling associations to the CFV library
%%
hifi(Component,Model) :-
    meta:'HighFidelityDynamicsModel'(Model),
    arrow:'hasModel'(Component,Model).

%% Utility to report on all hifi models
%%
print_all_components_with_hifi_model :-
    findall((C,M), hifi(C,M), List),
```

**Figure 7.6-52. CML Query Example**

#### 7.6.2.3.3 Parts Repository Example

The following is an example of a query to a semantic database for information on availability of engines from vehicle parts suppliers. We add a constraint that only parts with the certification '**MIL\_Cert**' get returned. The print debugging reveals that parts with other certification levels are considered but the final result will match only 3 instances.

```
%import http://www.kirkman-enterprises.com/sites/kirkman-enterprises.com/files/MSDL/MSDL-
Fullv1.owl

pr(A:B) :- print(A), print(':''), print(B).

p(A:B) :- print(B).

search_for(enines(E),supplier(S),certification(C)) :-
    'Ontology1236208666':'EnginesAndTurbines'(E),
    'Ontology1236208666':'hasProductFocus'(S,E),
    'Ontology1236208666':'hasCertification'(S,C),
    print('Supplier '),p(S),print(' has '),p(E),print(' with certification '),p(C),nl,
    C = _: 'MIL_Cert'.

% Expected results on print out, but only 3 of these have constraints of MIL_Cert
%
% Supplier ThuroMetalProducts has PartsforGoodsOfClassesEngineAndTurbine with certification
AS9100_Cert
%
% Supplier ThuroMetalProducts has PartsforGoodsOfClassesEngineAndTurbine with certification
ISO9001_2000
%
% Supplier WoolfAircraft has DieselEngines with certification ISO9001_2000
%
% Supplier WoolfAircraft has DieselEngines with certification MIL_Cert
%
% Supplier TXStateMfgCo has DieselEngines with certification ISO9001_2000
%
% Supplier TXStateMfgCo has DieselEngines with certification MIL_Cert
%
% Supplier BoyerMachine has FuelSystemComponents with certification ISO9001_2008
%
% Supplier BoyerMachine has FuelSystemComponents with certification ISO9001_2000
%
% Supplier OhioMachinedProducts has FuelSystemComponents with certification ISO9001_2000
%
% Supplier InnovativeMetal has Manifolds with certification MIL_Cert
```

**Figure 7.6-53. Parts Repository****7.6.2.3.4 Environment Example**

```
%import http://sweet.jpl.nasa.gov/2.2/sweetAll.owl
%import http://open-meta.com/process.owl.xml

earth_query(C,M) :-
```

The logic code to the left demonstrates calls using the SWEET ontology. SWEET is useful for classifying earth science data for vehicle context modeling.

**Figure 7.6-54. Call to the SWEET Ontology****7.6.2.3.5 Model Classification Example**

The logic code to the right demonstrates a query to ontologies of available mathematical modeling tools. The classification returns only those tools that provide the capabilities

```
%import http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl

model(A) :-  
  'VVO':'V&V'(A),  
  'VVO':'Analysis'(A),
```

indicated by the successive predicates. In this case, the *SPIN*, *UPPAAL*, and *CBMC* tools fit these categories. This is a classic SPARQL query as well.

A more extensive situation using the CML models is shown below. The library is populated with meta-information of modeling information related to particular engines (refer to Figure 7.6-55). Then a query is made to return a set of models associated with a specific engine type (refer to Figure 7.6-56). Note that this query will predictably fail if a model from each category is not found. Providing default behavior for queries must always be considered since Prolog operates under the closed-world assumption, that if it is not available it does not exist.

```

@prefix : <http://wcsn262:8001/backup/cml.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:
:aowl:Ontology .
:Engine :owl:Class .
:has_reliability_model :owl:DataProperty .
:has_maintainability_model :owl:DataProperty .
:has_diagnostics_model :owl:DataProperty .
:has_thermal_model :owl:DataProperty .
:has_logistics_model :owl:DataProperty .

:engine1 :Engine,
owl:NamedIndividual ;
:has_reliability_model "engine1_reliability_model";
:has_maintainability_model "engine1_maintainability_model";
:has_diagnostics_model "engine1_diagnostics_model";
:has_thermal_model "engine1_thermal_model";
:has_logistics_model "engine1_logistics_model".

:engine2 :Engine,
owl:NamedIndividual ;
:has_reliability_model "engine2_reliability_model";
:has_maintainability_model "engine2_maintainability_model";
:has_diagnostics_model "engine2_diagnostics_model";
:has_thermal_model "engine2_thermal_model";
:has_logistics_model "engine2_logistics_model".

:engine3 :Engine,

```

**Figure 7.6-55. Component Model Library N3**

```
%import http://wcsn262:8001/backup/cml.owl

%% Example workflow
% We have a set of engine components that we want to analyze comprehensively.
% Domain models can be attached to the different components.
% The domains span a range of Design-For-X considerations.

get_models_for_engine(Engine, [RM,MM,DM,TM,LM]) :-
    cml:engine(Engine),
    cml:has_reliability_model(Engine, RM),
    cml:has_maintainability_model(Engine, MM),
    cml:has_diagnostics_model(Engine, DM),
    cml:has_thermal_model(Engine, TM),
    cml:has_logistics_model(Engine, LM).
```

**Figure 7.6-56. CML library query**

#### 7.6.2.3.6 Process Workflow Example

The following are snippets of OWL-S process-service groundings. These are intentionally made similar as the process-to-service logic follows a pattern independent of the application. That is what makes automation of web-services practical in the sense that if the flow of data is predictable and it needs to be in a certain format, then automation can help.

```
%import http://www.mindswap.org/2004/owl-s/1.1/BookFinder.owl

%% OWL-S is the process-based ontology useful for generating workflows

pr(A:B) :- print(A), print(':''), print(B).

p(A:B) :- print(B).

find_book_process(P,B,I,S,G,A) :-
    process:'AtomicProcess'(P),
    process:'hasInput'(P,B),
    process:'hasOutput'(P,I), % ObjectProperty
    process:'Input'(B),      % Just a check
    service:'describes'(P,S),
    service:'supports'(S,G),
    grounding:'hasAtomicProcessGrounding'(G,A),
    grounding:'WsdlAtomicProcessGrounding'(A),
    grounding:'owlsProcess'(A,P),
    print('Applied a '), p(P), print(' to '), p(B), print(' giving '), p(I), nl,
    print('We can use a '), p(S), print(' to instantiate a '), p(G), print(' using '), p(A), nl.

%%% Expected output for unbound query:
%
% Applied a BookFinderProcess to BookName giving BookInfo
%
% We can use a BookFinderService to instantiate a BookFinderGrounding using
BookFinderProcessGrounding
```

**Figure 7.6-57. Book Ordering**

The composable query for finding a book (above) is similar to that for looking up a terrain slope (below). This similarity and regularity in patterns allow composable workflows to be created. The Web Service Composer from <http://mindswap.org> uses a SWI-Prolog reasoner to compose a workflow from base ontologies and then generates the dynamic web service requests to execute queries.

```
%import http://www.laits.gmu.edu/geo/ontology/owl/owl/v2/slope_precondition.owl

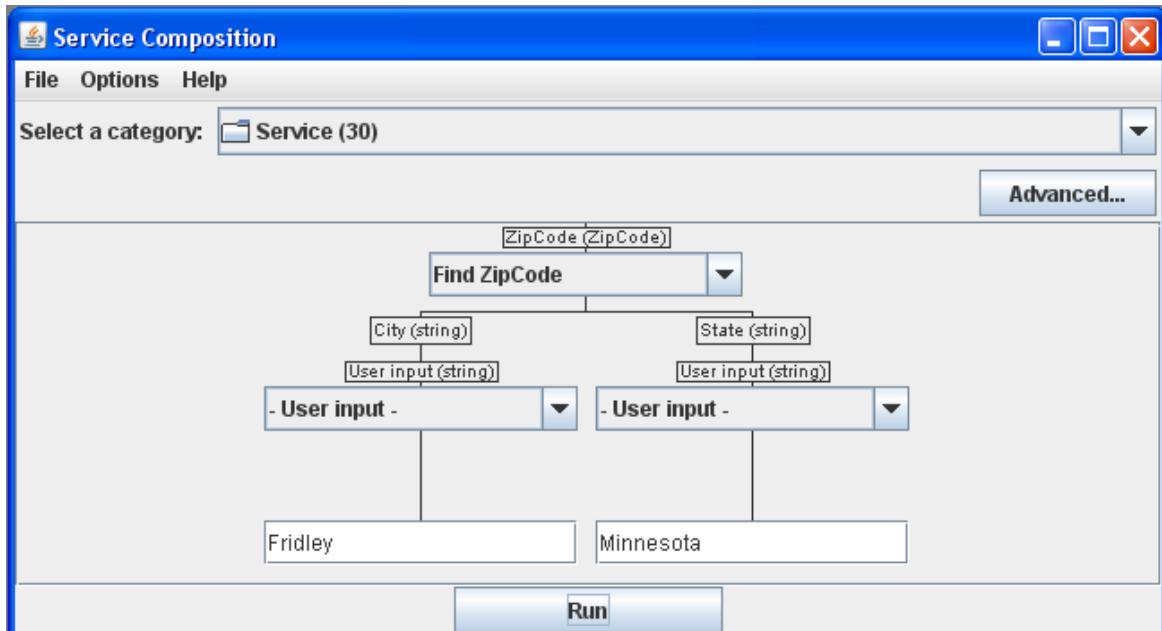
%% Example of OWL-S used in a context model for finding terrain slopes

pr(A:B) :- print(A), print(':''), print(B).

p(A:B) :- print(B).

find_slope(P,B,I,S,G,A,Ref) :-
    process:'Process'(P),
    process:'hasInput'(P,B),
    process:'hasOutput'(P,I), % ObjectProperty
    service:'describes'(P,S),
    service:'supports'(S,G),
    grounding:'hasAtomicProcessGrounding'(G,A),
    grounding:'WsdlAtomicProcessGrounding'(A),
    grounding:'wsdlOperation'(A,Ref),
```

**Figure 7.6-58. Terrain Querying Similar to Book Ordering**



**Figure 7.6-59. The Web Service Composer Will String Together a Sequence of Service Calls from an Ontology to Allow Flexibility in the Creation of a Composable Workflow**

#### 7.6.2.3.7 Design Deployment Example

Including adaptable Intelligence, Surveillance, Target Acquisition & Reconnaissance (ISTAR) capabilities on vehicles allows matching of mission sensing requirements with available sensor technologies [RKT05].

```
%import http://wcsn262:8000/istar.owl

getConfigurations(T,[P|S]) :-
    deployablePlatform(T,P),
    extendSolution(T,P,[],S).

deployablePlatform(T,P) :-
    istar:'Platform'(P),
    not((istar:'requiresOperationalCapability'(T,C),
        not(istar:'providesCapability'(P,C)))).

extendSolution(T,P,Prev,Next) :-
    requireSensor(T,P,Prev,X),
    % istar:'mounts'(P,X),
    A=[X|Prev],
    extendSolution(T,P,A,Next).

extendSolution(T,P,S,S) :-
    not(requireCapability(T,P,S,_)).

requireSensor(T,P,S,X) :-
    requireCapability(T,P,S,C),
    istar:'providesCapability'(X,C).
```

**Figure 7.6-60.** ISTAR Query (from [SMV11])

#### 7.6.2.3.8 Deep Inference Example

The reasoning illustrated in this example involves inferring extra information from limited meta-data. Consider the problem of estimating the mass of parts with the only information available limited to dimensions and properties such as density.

```
@prefix : <http://wcsn262:8001/comp.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:
:component a owl:Ontology .
:system a owl:Class .
:isHomogeneous a owl:Class .
:hasLength a owl:DataProperty .
:hasWidth a owl:DataProperty .
:hasRadius a owl:DataProperty .
:hasHeight a owl:DataProperty .
:hasDensity a owl:DataProperty .
:partOf a owl:ObjectProperty .

:sys a :system.

:a a :component,
     :isHomogeneous,
     owl:NamedIndividual ;
     :hasLength 1.0 ;
     :hasWidth 1.0 ;
     :hasHeight 1.0 ;
```

**Figure 7.6-61. Ontology for a Set of Components with Properties**

Figure 7.6-61 is a N3 OWL ontology for a “comp” hierarchy which consists of components with specific properties.

Figure 7.6-62 is a reasoner which can determine the mass of each of the components, but can only get this information indirectly by inferring the mass from other properties available. Thus if a component has a length, width, and height specified, then the reasoner assumes it is block shaped and computes the mass from the density of the material. If, on the other hand, a radius is specified, then a different formula for mass is used. This is an example of deeper reasoning than is available from pattern matching.

```
%import http://wcsn262:8001/demo/comp.owl

%% Example of an analysis archetype for finding the weight of a collection
%% of components comprising a system.

%% Depending on what a component has for sizing dimensions, a different weight
%% calculation will get invoked. It is polymorphic, but guided by constraints

calc_weight(System, Weight) :- %% weight for a block
    comp:'component'(A),
    comp:'partOf'(A, System),
    comp:'isHomogeneous'(A),
    comp:'hasLength'(A, L),
    comp:'hasWidth'(A, W),
    comp:'hasHeight'(A, H),
    comp:'hasDensity'(A, D),
    print('block '), print(A), nl,
    Weight is D*L*W*H.

calc_weight(System, Weight) :- %% weight for a cylinder
    comp:'component'(A),
    comp:'partOf'(A, System),
    comp:'isHomogeneous'(A),
    comp:'hasLength'(A, L),
    comp:'hasRadius'(A, R),
    comp:'hasDensity'(A, D),
    print('cyl '), print(A), nl,
    Weight is D*L*3.1415*R*R.
```

**Figure 7.6-62. Reasoner Which Understands How to Derive Mass from Shape Properties**

### 7.6.2.4 ECTo

The Early Concepting Tool (ECTo) guides a vehicle design process by applying abstract components that fit together *a priori* as a result of applying archetypal rules. Any down-select process that excludes incompatible components is made possible by doing DSE as a successor stage. The ECTo conceiving reasoner encodes and visualizes spatial representations, complex space claims, and articulations, which are difficult to represent and reason with in pure logic.

As a system design tool ECTo enables editing of a master model primarily through the hierarchical assembly and manipulation of components from the CML. It is focused primarily on empowering a designer in the early design phase to be able to incorporate and manipulate major design drivers and rapidly assess the qualities of system concepts. The resultant concepts can be used as the basis for more detailed design.

### 7.6.2.5 Co-Analysis Flow using GEAR

For the example of a vehicle ramp design, we included automated elements along the control flow axis as well as the data flow axis (refer to Figure 7.6-63). The data flow followed from ontological sources of data as well as tacit facts stored in a knowledgebase. The control flow is orchestrated by reasoners which do design space exploration and the test-space exploration.

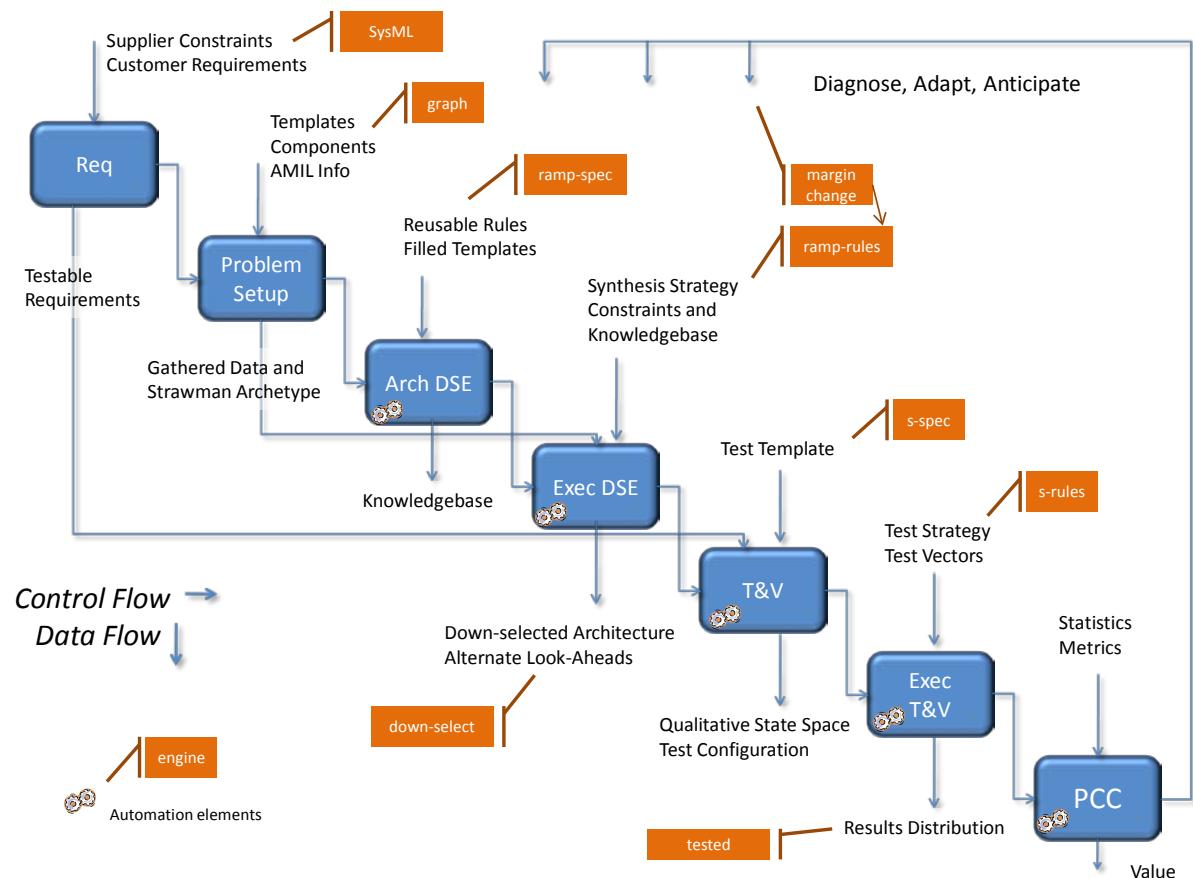
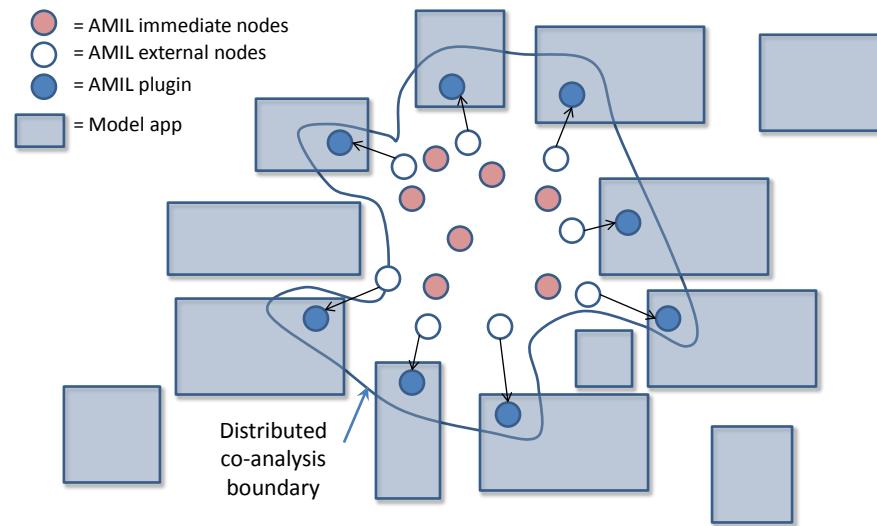


Figure 7.6-63. Flow of Co-analysis from Initial Requirements Using Automation where Possible

This sets the stage for the co-simulation required to shake-out and verify the concepts and optimized designs.

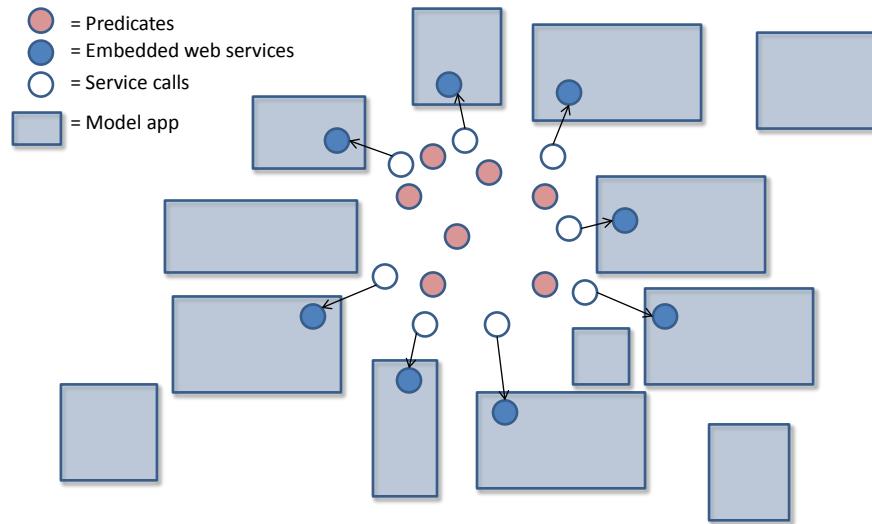
### **7.6.3 Co-Simulation and T&V**

To understand the needs of co-simulation and how it differs from co-analysis, consider a typical AMIL-based architecture in Figure 7.6-64. The executable models in the analysis are wrapped with AMIL-aware plug-ins as needed so that they can communicate with the main graph database.



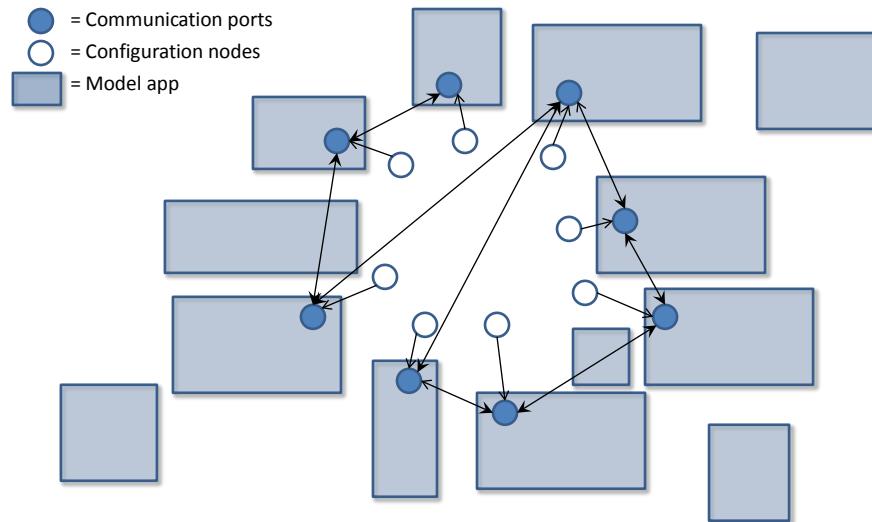
**Figure 7.6-64. AMIL Nodes Serve as Plug-Ins Within a Distributed Co-Analysis**

The boundary of the co-analysis includes immediate nodes, external nodes, and whatever plugins are required for execution. Structurally, this differs little from the service-oriented architecture shown below, which can be used as a pattern for a workflow composed via OWL-S and SSWAP elements:



**Figure 7.6-65. Composable Workflow Analysis Which is a Loosely-coupled, Service-Oriented Architecture**

The mechanisms for co-simulation differ in that the models often need to communicate *directly* with one another as opposed to through an intermediary of a graph database (in the case of AMIL) or through a reasoner and its knowledgebase (in the case of a composable workflow). A real co-simulation architecture will thus align more closely to Figure 7.6-66.



**Figure 7.6-66. Typical High-speed Co-simulation Network with Direct Data Paths Between Communication Ports**

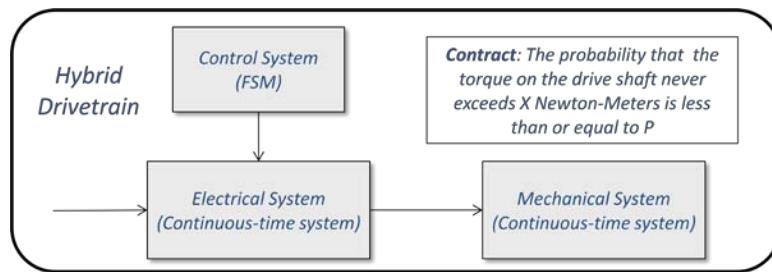
Note that the nodes are replaced with communication ports and the simulation model applications have direct communication links to provide the least amount of latency and the highest throughput possible. The links to any part of the graph database or knowledgebase are restricted to configuration nodes. The co-simulation has now transformed into more of a choreographed exercise than the orchestrated co-analysis.

The choreography is evident in that the individual models know what to do because they are simulating the actual design execution and the appropriate environmental context, and synchronizing these interconnects will happen at the cyber-physical level. This also becomes the environment for test and verification, as we move from a highly-abstracted world to a high-fidelity world. In other words, the orchestrated hand-holding of co-analysis is no longer operable, and we need realistic mechanisms to be simulated for determining a PCC.

This more restrictive architecture is not a limitation to how we apply AMIL. For example, a co-simulation model for AMIL can potentially be based on a Distributed Command Pattern (DCP) for data communication and synchronization. The DCP is very simple yet powerful in that it can generate basic building blocks for Models of Computation and Communication (MoCC) in any object-oriented base language (Java, C++, etc.). The significant advantage for the ARRoW process is how well it fits in with the AMIL framework, particularly as a routing table for distributed nodes. If the throughput of AMIL is insufficient for direct heterogeneous and multi-physics computing applications, it will be certainly useful for a DCP application, as this requires an intelligent and organized routing configuration, something which AMIL excels at providing.

### 7.6.3.1 MoCC and Heterogeneous Simulation

The fact that heterogeneous simulations have an unbelievably rich set of possible interactions leads to the META concept known as Models of Computation and Communication (MoCC). Referring to this set as MoCC allows us to categorize the ways in which simulations have been architected. For a typical cyber-physics problem, each simulation may need to talk to a concurrently executing simulation, so that a path toward making the communications as uniform or adaptable as possible is a good one to follow.



**Figure 7.6-67. MoCC Applied to Modeling of a Drivetrain**

### 7.6.3.2 Tagged Signal Model

#### 7.6.3.2.1 AMIL for Co-Simulation

The language semantics behind AMIL allow a model developer to access the underlying graph database, which contains information on the computational nodes comprising an extended model, and of the edges/links connecting these nodes.

When a model tries to emulate the behavior of something as substantial as a complete vehicle operation with associated multi-physics, the demands placed on the interconnect language grow accordingly. The complete vehicle simulation is difficult enough to construct that the simulation developer does not want the interconnect layer to become an additional hindrance.

Instead, it should be based on a pattern as simple as writing a procedural call or tapping into a data stream.

So we want a messaging pattern that fits in with the simplicity of AMIL. Let us consider first the DCP, which has particular relevance to an interconnected model. The elegance of the pattern stems from the conciseness of its use. The typical invocation takes two lines of code:

```
msg Object.Command;
DCP.Send(msg);
```

The first line declares the desired command as a message and the second line sends the message. The routing table configuration is opaquely hidden to the client software by the DCP dispatching logic (what is referenced within the **Send** procedure), so that only the problem domain is exposed and not the solution domain.

The routing configuration is simple as well and isolated from the static program code by a knowledgebase, or an AMIL graph database. A typical set of routing configuration rules may look like the following, where the destination address is given by a logical node number.

```
connection(object:command, 2).
host_node(2, '192.168.10.1').
```

This information is accessed at run-time only, and it essentially instructs the DCP dispatcher to send all messages of class **object.command** to a node logically defined as 2 (which then maps onto an IP address in this example).

The interconnect language is powerful in that it will allow general pattern matching. Thus, if we wanted to indicate that all commands owned by object resided on node 2, we would declare this:

```
connection(object:_, 2).
```

The underline ('\_') character indicates a wild-card match for all messages belonging to this object.

The language can also incorporate sophisticated rules. For example, say that the object needs to co-reside on the same node as a specific server object. Then we can declare the rule by the predicate logic:

```
connection(object:_, Node) ←
    connection(server:_, Node).
```

This says to pattern match the node number for the object against the node that the server resides on. We only need to declare the server elsewhere, as the implication follows a rule:

```
connection(server:_, 4).
```

Fail-over semantics are accomplished by providing an alternate route should the main connection fail:

```
connection(object:_, 2).
hot_backup(object:_, 3).
```

In our variation of the DCP the Node labeled 0 has special meaning. It is reserved to indicate that a message is routed to a local object. For this reason, all executables are assigned a *Node* number; they then can perform introspection on their own routing behavior and not invoke distributed communication unless needed.

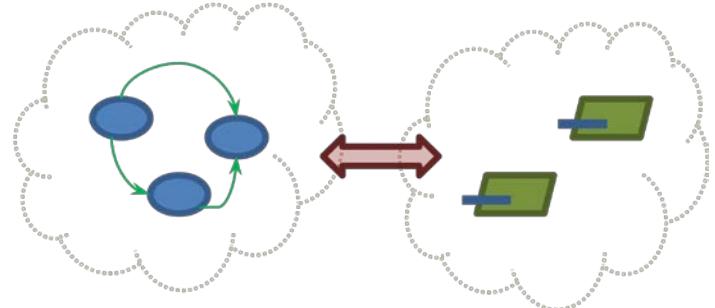
#### 7.6.3.2.2 Tagging for Synchronization

The DCP message depends on inheritance from a specific base class. A refined message can add data elements so that information can be transmitted along with the command. Therefore variants of the basic DCP message can include pure commands, queries (i.e., request-replies), and status requests (i.e. output only).

Since the messages can contain extended information, they fit in well with the *Tagged Signal Model* (TSM) advocated by Lee and Sangiovanni-Vincentelli. TSM was introduced to address the problems brought on by heterogeneous modeling and co-simulation. The heterogeneous environment defined a set of mix and match computational models which can include combinations of:

- Continuous time with discrete time
- Sequential with concurrent, *etc.*

A foundational consideration exists behind TSM. Making heterogeneous simulations work efficiently boils down to a need for a general adaptive mechanism to uniformly communicate between models. This is related in scope to the DCP strategy, which applies polymorphism of data-types to implement message passing between models. As we will demonstrate, message-passing facilitates not only distributed computing, but the fundamental polymorphism allows us to exercise many other TSM-like behaviors, such as synchronization, signaling, re-entrant-safe data handling, *etc.*

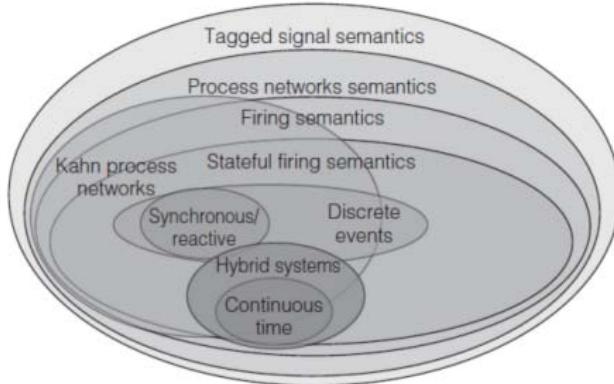


**Figure 7.6-68. The tagged-signal model allows interoperability of different MOCC.**

The basic entity in the tagged signal model is an event, which is a value/tag pair. The value is essentially the data that a message can contain and the tag establishes the name and unique class that the message belongs to.

Based on their classification, tags can establish ordering relationships in time, and so different models of time appear as structure imposed on the intersection between sets of all possible tags. Describing a process can then appear as relations between signals as sets of events, for example, a synchronized transaction can occur if a condition variable is attached to a tagged signal.

The character of such relations follows from the type of process it describes. The figure below represents a categorization of temporal systems that can fall under the TSM umbrella.



**Figure 7.6-69. The Tagged-signal Model Encompasses a Family of Simulation Behaviors (from Lee and Sangiovanni-Vincentelli).**

According to the categorization, the family of tagged signal models intuitively consist of processes that run concurrently, which is what you would expect in a co-simulation. Constraints imposed on the shared signals' set of tags define communication among the processes. Tags can represent a broad range of annotated relations, such as total orders in timed systems and partial orders in untimed systems.

Consider a few examples from typical system simulations. In the first case, we may want to interface an event-driven base simulation to a continuous time simulation such as what Simulink can produce. These data-flow systems typically solve differential equations, so they operate on intervals of time,  $\Delta t$ . One could set up the Simulink simulation to generate a step-wise solution by requesting a result at a  $\Delta t$  interval in advance, or one could advance the clock and request a solution for the current time. Even though the time interval is generally small, not factoring in the temporal shift and ordering properly can lead to mismatches in expected output of a heterogeneous simulation. A tagged signal model can account for this as the hand-shaking built-in to the connection can adapt to the difference. For that reason, the TSM connections can include what are called *adaptors*.

Another case involves the modeling of edge detection as would happen in the simulation of digital logic systems. Logic gates typically synchronize on rising edges of binary signals, so the synchronization of multiple gates in a circuit consisting of clocks and cascading logic is critical. The referenced paper describes in detail the semantics of a modeled digital signal which effectively emulates that which would be found in a VHDL simulator. [PL07]

The combination of message-passing and tagged-signals turns out to be very powerful in that remote and local variations of synchronization are easy to model. We thus have no problem modeling and simulating the following types of synchronizing behaviors:

- Read-modify-write
- Bounded/unbounded buffered FIFO (e.g., mailboxes)
- Rendezvous

These kinds of behaviors are the bread-and-butter building blocks for designing cyber-physical systems that have any degree of automation.

### 7.6.3.2.3 Merging TSM and DCP

Importantly, we can unify the separate notions of tagged signals and distributed commands. Consider distributed commands as objects which possess a homing instinct based on the unique identity of their tag. Thus we can assert several properties of a tagged distributed command:

- Tagging allows messages to dispatch to their correct destinations
- Tags can be used as an index to route to computational nodes
- Tags are built into OOP languages via virtual dispatch tables, i.e., *vtables*

This leads to the declaration of a routing configuration table as we described earlier. The names declared in the text correspond precisely to the names uniquely defined for the tagged signals, with the object dot notation used to define the names. Combining the tagged DCP with the tagged synchronization primitives allows a large diversity in realizations.

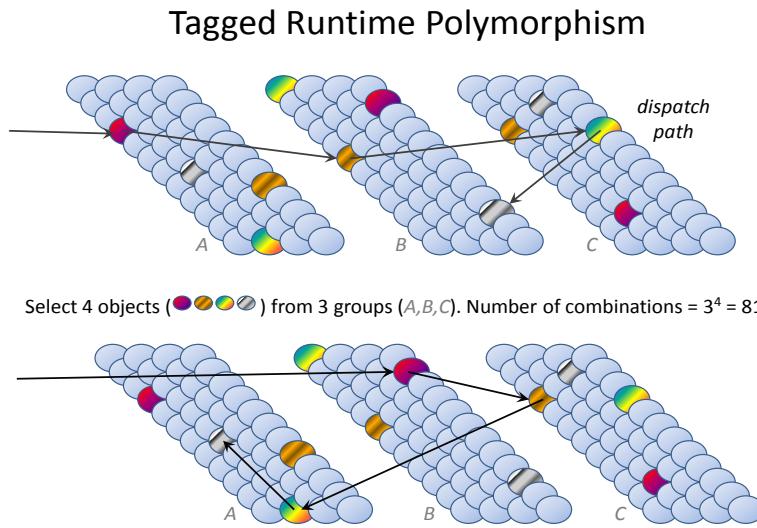
- Different computational models hosted in appropriate threads and processes
- Synchronization primitives define temporal behaviors

This approach has been prototyped in a kind of middleware library we refer to as the patterned AMIL command environment. This uses the distributed-command pattern as a foundation. All messages are tagged with an object identity and contain time ordering slots encapsulated in their base class. Messages at the most basic level can refer to signaled events, hence the correspondence to the Tagged Signal Model.

A co-simulation becomes an agent-driven process network with all message-passing and synchronization semantics based on passive or active objects derived from tagged types. A wide variety of MoCC systems can be invoked with fundamental tagged patterns. As with the basic distributed command pattern above, the patterns at most involve a few lines of code; in particular:

- Concurrency
- Mailboxes, rendezvous, publish-subscribe, etc.
- Discrete Event Engine (Degas[LP06]) → hybrid cyber-physical systems

A possible configuration to demonstrate involves AMIL-controlled dispatching to objects across remote clusters. The idea is to maintain variants of objects in different clusters depending on how we want to experiment with performance, fidelity of representation, or design space exploration. In the figure below, we show two possible configurations which each connect four active objects. Considering this in the context of a design space exploration, a total of 81 combinations are possible via DCP reconfiguring.

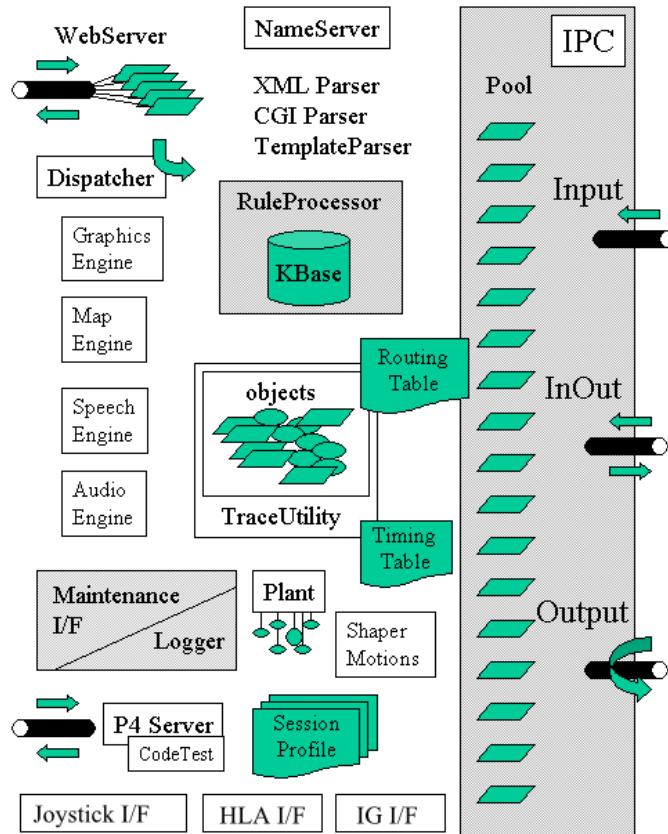


**Figure 7.6-70. Tagged Signal Runtime Polymorphism**

The tagged signal model lies at the heart of an integrated simulation. As objects refer to a message containing a full informational record, it has general expressiveness to capture many relevant behaviors. The generality of the synchronization environment allows various levels of behavioral polymorphism, such as dataflow, time-triggered, discrete-event (through *DEGAS*), communicating sequential processing, process networks, and push-pull messaging.

The connection to AMIL is now very apparent as AMIL allows us to map the connectivity of a co-simulation:

- The distributed command pattern allows destinations to occur on any node of the network.
- A connection-oriented configuration approach comes along with the pattern.
- The configuration describes a run-time modifiable routing table.
- By encoding the configuration to reside within a knowledgebase, we have a set of rules to enable sophisticated pattern matching techniques to be employed.

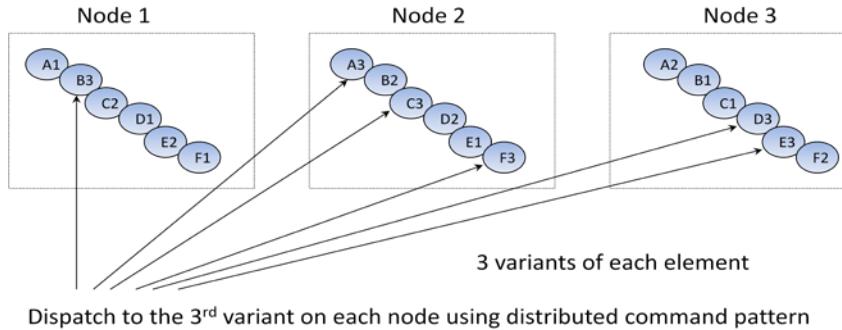


**Figure 7.6-71. A Typical Architecture of an Agent-Based Co-Simulation**

The full co-simulation consists of actors or agents that work concurrently and cooperatively, which emulates the architecture of a ground vehicle. The combination of an AMIL configuration with run-time polymorphism through tagged signals disambiguates intent in both information transfers and behaviors initiated. This becomes an ideal heterogeneous computing environment for analysis and migration toward a virtual prototype and that can support various models of computation.

A natural extension is to incorporate design space exploration and adaptability into this architecture, with very little additional effort. As per Figure 7.6-14, we may need to successively investigate a sequence of alternatives and how they may best reduce the complexity of the state space.

As one approach to take, consider that the design-space exploration is really just a concise application of a distributed command pattern with supplemental decision rules. This is an excellent exercise in demonstrating dynamic CML loading because we can use the command pattern dispatching mechanisms to insert the appropriate component in our environment.



**Figure 7.6-72. Distributed Command Pattern for Selecting Among Alternative Implementations**

As only a few options exist for performing automated design space explorations, the primary enabler in this case is to concentrate on components and functionality that work plug-compatibly and have contract-style interfaces. As similar connection configuration rules play a part in the operation of the DCP architecture, the implication is that design space exploration can be analyzed in terms of a heterogeneous simulation, *and very late in the game*. This has significant benefits considering that set-based concurrent engineering and adaptability are important factors for META.

#### 7.6.3.2.4 Evaluation

#### 7.6.3.2.5 Evaluation

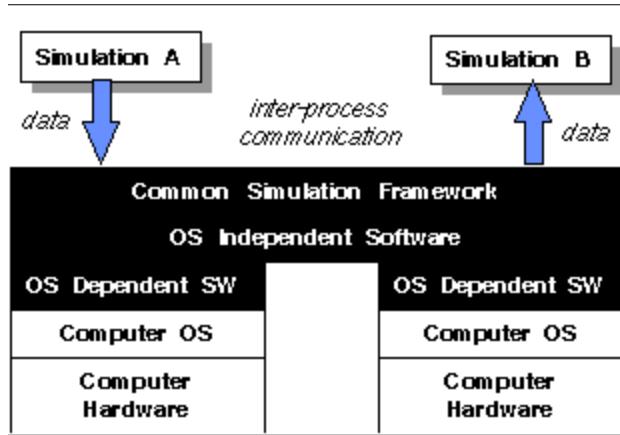
The co-simulation of a vehicle-scale cyber-physical system will definitely contain complexity, but the building blocks benefit from simple patterns.

- Advanced distributed command-pattern middleware
- Embedded knowledgebase (AMIL) for distributed communication
- Heterogeneous tool integration
- Integrated launching environment
- Hybrid Discrete Event Simulation and Testing Framework
- Design-space exploration
- Advanced 3-D visualization (refer to ECTo) and collision checking
- Run-time instrumentation and post-processing of artifacts
- Potential additions such as High-Level Architecture (HLA) integration

By combining the capabilities in different ways, we can accomplish combined goals. For example, we can consider design-space exploration as an application of a distributed command pattern with a front-end emphasis on expert system reasoning and integration. This also demonstrates dynamic Component Model Library loading as we can use the command pattern dispatching mechanisms to get to the correct component in our environment.

### 7.6.3.2.6 Patterned AMIL Command Environment

This section contains a list of adaptors and synchronization primitives that certainly apply to a practical distributed co-simulation. The communication problem we want to generalize is to provide a rich set of service adaptors for inter-process communication as shown below.



**Figure 7.6-73. The Application of Middleware**

If we were to define a paradigm for how distributed command patterns typically fit together, a passable description may be to call the end result a “class-based messaging architecture”. The specific variation is to lean on a class-based command pattern, which also has the potential for language interoperability (especially with common *vtable* implementations among open-source compilers). Such an essential pattern provides benefits in (1) instrumenting code and (2) allowing for other building block patterns to be constructed. What we call the set of PACE patterns combines inheritance and class-wide operations on the base message classes. The base classes contain time stamps, task IDs, and other identifiers which can be used to trace execution. This is the basic architectural pattern that would provide the infrastructure for a MOCC co-simulation, intended to support both discrete event simulation and real-time simulation. Other patterns evolved based on needs of simulating a real cyber-physical design, rich with concurrent constructions.

The Tagged Signal Model at the heart of the messaging consists of a set of attributes that are inherited by all derived messages. These include node numbers (for routing), thread ID's, synchronization and delivery enumerations, and time measures (absolute and relative). More attributes can be added to specialize the services, but these are the essential ingredients in combining a distributed command pattern with a tagged signal model.

**Description of Patterns.** The Apache open source project uses an integration framework called Camel. The basic mechanisms are described as a set of [Enterprise Integration Patterns](#). They are fairly comprehensive for the application domain of social communication that they have targeted, but it falls well short of the intended application domain of tool co-simulation. The following provides a set of possible PACE patterns presented in the same fashion as the Apache Camel patterns. The code snippets attached to each entry demonstrates the intended conciseness of pattern usage, with the icon representing a tagged signal pattern and representing a task or process.

**Command (or Message) Pattern**

Command messages are declared in a module specification, with at least one of the three available primitive dispatching operations also specified. The module implementation requires the filled-out operational code for the selected primitives.

```
type Msg_Name is new Pace.Msg with ...; // defined in Msg_Name_Module
procedure Input (Obj : in      Msg_Name); // optional primitive #1
procedure Inout (Obj : in out Msg_Name); // optional primitive #2
procedure Output(Obj :       out Msg_Name); // optional primitive #3
```

In formal terms, the automatically inherited operations include `Input`, `Output`, and `Inout`. Depending on the parameter mode desired (whether data is "in", "out" or "in out"), one or more of these operations can be declared for each subclass'ed command `Msg`. The *command* pattern is also mentioned in the "Gang of Four" patterns book, but different language can implement it slightly differently, particularly with respect to pointer manipulation. On the client side, instancing of command messages is straightforward.

```
use Msg_Name_Package;
Msg : Msg_Name;

Input (Msg);
```

In many cases, a "class-wide" procedure declaration is used in conjunction with a command message. The class-wide to primitive operation dispatching allows further building block utilities to be created from the command pattern. These utilities include such patterns as Trace and Proxy.

In terms of contract-style programming the mutability of the command pattern data flow indicates its use.

"inout"	General queries (with potential side effects)
"input"	Commands (state changes)
"output"	Monitoring (no side effects or state change)

The purely functional style is missing from the command pattern.



### ***Dispatched Command Pattern***

This pattern is a slight variation of the client-side invocation of the command pattern. Instead of calling the primitive operation directly, a class-wide operation is used which effectively enables redispatching to the appropriate primitive. The target of the indirection can be changed by registering a different class-wide callback.

```
Msg : Msg_Name_Package.Msg_Name;

Pace.Dispatching.Input (Msg);
```

Note that these first two patterns are the essence of the architectural DCP paradigm.



### ***Trace (or Instrumentation) Pattern***

The library contains class-wide utilities which apply to any derivation of `Pace.Msg`. One of these, the class-wide operation called `Pace.Log.Trace` provides a convenient way to instrument the code, as it accepts instances of the class-wide `Pace.Msg` as its parameter.

```
procedure Input (Obj : in Msg_Name) {
    Pace.Log.Trace (Obj);
}
```

On trace output some of the synchronization semantics can be logged, which may include: => simple, >> synch, -> asynch, <> release following SysML notation.



### ***Unit Identification Pattern***

This pattern allows one to declare a character string identifier which automatically matches the enclosing module name. This is typically declared in a body.

```
function ID is new Pace.Log.Unit_ID;
```

In general, placing character strings in code to identify packages is a poor idea; for example if maintenance occurs the strings may need to be updated. This provides an automated approach.

Along the same lines, PACE uses object tags in hash tables to textually associate data with messages. This can be demonstrated by combining the Tag Identification Pattern and Message Lookup Patterns.



### ***Agent (or Task) Identification Pattern***

This pattern allows one to register the name of a task. Use this with the Unit Identification pattern. All tasks are released when the default `Pace.Log.Agent_ID` is called from the main procedure.

```
function ID is new Pace.Log.Unit_ID;

task Agent {
    Pace.Log.Agent_ID (ID);
    ...
}
```



### ***Tag Identification Pattern***

If the need arises to get the character string representation of a command message, use the following pattern:

```
procedure Input (Obj : in Msg_Name) {
    Pace.Log.Put_Line (Pace.Tag (Obj) & " called");
}
```



### ***Log Exception Pattern***

Log a descriptive string without blocking, allowing a concurrent task to monitor. In practice, each task exception handler calls `Pace.Log.Ex` with an optional text string. The monitoring task needs to call the function `Pace.Log.Wait_For_Ex` to retrieve strings placed on the queue.

```
exception when E : others =>
    Pace.Log.Ex (E, "extra info");
```

In practice it is useful to log all exceptions, and make sure there is an exception handler in every task.



### ***Synchronized Message Passing (Rendezvous) Pattern***

Based on C.A.R. Hoare's communicating sequential processes, if the implementation of Input makes a call to a task rendezvous, the transaction is **synchronized**.

```
// Agent Task
// [REDACTED]
task Agent {
    accept Input (Obj : in Msg_Name) {
        Pace.Log.Trace (Obj);
    };
    ...
};

// Client Task [REDACTED]
//
procedure Input (Obj : in Msg_Name) {
    Agent.Input (Obj); // Transfer control and data to Agent
};
```

The calling Client must not be the same task context as the Agent, otherwise the execution will deadlock. In practice, this does not cause a problem because it is practical to detect deadlock through either code inspections or executable tests.



### ***Surrogate (Asynchronous) Message Passing Pattern***

This pattern uses a surrogate task to emulate an asynchronous message passing protocol, this is also often referred to as "send and forget" semantics. If a dispatching operation called `Input` is declared, then the Asynchronous send pattern can be applied. There are two flavors to this pattern, one that uses a dedicated task and one that uses a pool of surrogate tasks. Both approaches call the dispatching operation named `Input`. Note that since the asynchronous protocol does not require a return, neither `Output` and `Inout` calls are applicable since they do expect data back from the called primitive. The easiest flavor of the pattern to use is the pooled variation. One task in the task pool calls the class-wide operation as determined in the `Pace.Surrogates` package.

```
Msg : Msg_Name;

Pace.Surrogates.Input (Msg);
```

The other variation is to use a dedicated task to manage the handoff from the client to the surrogate, who can then asynchronously deliver the message.

```
package Async is new Pace.Surrogates.Asynchronous (Msg_Name);
...
```

```

Msg : Msg_Name;

Async.Surrogate.Input (Msg);

```

The recommendation is to always use the pooled version pattern since it requires less code. A configuration variable PACE\_MAX\_SURROGATES controls the number of tasks in the pool.



### **Mailbox Pattern**

This pattern pairs up sending and receiver tasks via a command message. Application developers typically won't use this pattern directly this but it is needed by the Notify pattern. It is thus more of a building block pattern for use by other PACE abstractions.

```

package Mailbox is new Pace.Msg_IO (479); // prime number for hash table size

// Server side ██████████
Msg : Msg_Name;
Mailbox.Await (Msg);

// Client side ██████████
Msg : Msg_Name;
Mailbox.Send (Msg);

```



### **Notify Pattern**

This simple but very powerful pattern provides a trigger/suspend pair on a message. The message has two pre-built operations (`Input` and `Inout`) built up from the Mailbox pattern. The inherited `Inout` primitive will suspend on the subscription message, while the inherited `Input` provides a built-in trigger (or publish) mechanism from a concurrently executing task.

```

type Msg_Name is new Pace.Notify.Subscription with
{
    Data_Field : ...
};

// Server-task suspend ██████████
task Agent {
    Msg : Msg_Name;

    Inout (Msg);
    Msg.Data_Field := ...

    // Client-task trigger ██████████
    Msg : Msg_Name;
}

```

```
Msg.Data_Field := ...  
Input (Msg);
```

This is useful to allow the task rendezvous mechanism to be extended to regions outside the task body.



### ***Queue and Guarded Queue Pattern***

The Guarded Queue pattern provides a re-entrant safe mechanism to pass constrained data.

```
package Q is new Pace.Queue (Item_Type);  
package Guarded is new Q.Guarded;  
  
Guarded.Get (Item); // Task #1 [REDACTED]  
  
Guarded.Put (Item); // Task #2 [REDACTED]
```

The queue operations Guarded.Put and Guarded.Get are available.



### ***Mutex Pattern***

This pattern protects data from simultaneous access via an automatically scoped semaphore object. This is a fairly common textbook mutual exclusion pattern.

```
My_Mutex : aliased Mutex;  
  
...  
  
// Lock [REDACTED]  
L : Lock (My_Mutex'Access); // Task #1 access locked data until end of scope  
  
...  
  
// Contend [REDACTED]  
L : Lock (My_Mutex'Access); // Task #2 access locked data until end of scope
```



### ***Pooled Resource Pattern***

This pattern provides a pool of guarded keys that can be used to access task or re-entrant critical data. Currently, it is used as a building block pattern for other PACE abstractions, such as the Pace.Socket module.

```
type Pool_Range is range 1 .. 10;  
package Pool is new Pace.Resource (Pool_Range);  
  
Key : Pool_Range;  
  
Key := Pool.Get;  
// access locked data until end of scope
```

```
Pool.Free (Key);
```



### **Single Event Wakeup Pattern**

Instance a Pace.Signals.Event protected type in a package body that has access to multiple threads of control.

```
Evt : Pace.Signals.Event; [red bar]
...
Evt.Suspend; // Task #1 [black bar]
...
Evt.Signal; // Task #2 which wakes up suspended Task #2
```



### **Multiple Event Wakeup Pattern**

For waking up multiple threads of control. An enumerated type or other ranged scalar type can be used to specify the desired signalling states.

```
type Colors is (Black, Red, Blue);
package Evts is new Pace.Signals.Multiple (Colors);
```

Three variations of multiple signal control exist. The first specifies waiting on a specific enumeration value:

```
Evts.Await (Black); // Task #1 [red bar]
...
Evts.Signal (Black); // Task #2 [black bar] wakes up suspended Task #1
```

The second describes waiting on any enumeration:

```
Color : Colors;

Evts.Await_Any (Color); // Task #1 [red bar]
/* post-condition -> Color returns Black

...
Evts.Signal (Black); // Task #2 [black bar] wakes up suspended Task #1
```

The last variation describes waiting on all enumerations:

```
Evts.Await_All; // Task #1 [red bar]
...
Evts.Signal (Red); // Task #X [red bar]
...
Evts.Signal (Blue); // Task #Y [blue bar]
...
Evts.Signal (Black); // Task #Z [black bar] which finally wakes suspended Task #1
```



### **Data Wakeup Pattern**

This is another pattern to use instead of a rendezvous. To use this pattern, first instance a `Pace.Signals.Shared_Data` protected type in an implementation that has access to multiple threads of control.

```

Obj : aliased Msg_Name;
...
Data : Pace.Signals.Shared_Data (Obj'Access);

task Agent {           // Task #1 [REDACTED]
    Msg : Msg_Name;

    Data.Read (Msg);
exception
    when Pace.Signals.Data_Mismatch => // Uh-oh, types don't match
};

Msg : Msg_Name;          // Task #2 [REDACTED] wakes up Read by calling Write
Data.Write (Msg);

```

Of all the patterns defined so far, this one has the potential for data type mismatches. If the reader waits on a message that doesn't match the type of the writer, an exception will be raised. Compare this to the Synchronized Message Passing or Notify pattern and you can see that the extra complexity of the Shared Data Wakeup pattern makes the type-safe rendezvous or notify a much better choice.



### **Task Wakeup Pattern**

This pattern uses built-in task identifiers to associate waiting tasks with triggering clients (TID stands for Task ID).

```

Id : Pace.Thread;           // Static variable
...
task Agent {           // Task #1 [REDACTED]
    Id := Pace.Current;
    Pace.Signals.Tid.Wait;
...
Pace.Signals.Tid.Signal (Id); // Task #2 [REDACTED] wakes up suspended Task #1

```

The Agent Wakeup pattern is used in the `Pace.Signals.Buffers` service. Each subclassed `Msg` contains a TID field which is used to pass thread identifiers between active objects. This is a direct application of the Tagged Signal Model.



### **Channel (Unconstrained Command) Pattern**

This pattern allows heterogeneous and unconstrained command messages to be mixed in a safe and controlled way. The convenience operator "+" is defined to allow simple construction of the "channel" messages.

```
Msg : Msg_Name;
Chan : Pace.Channel_Msg := +Msg;
...
Input (+Chan); // Dispatches to primitive operator
```

The Channel pattern is used in the `Pace.Signals.Buffers` service and in the command callback pattern.



### ***Buffered Command Pattern***

This pattern uses the Channel pattern and the Agent Wakeup pattern to enable synchronized passing of heterogeneous/unconstrained data. This is accomplished by buffering messages in a protected queued data structure.

```
type Msg_Name is new Pace.Msg with
{
    Char : Character;
};

Queue : Pace.Signals.Buffers.Buffer; // Contains Class-wide Queue

task Agent {           // ████
    Item : Pace.Channel_Msg; // Single Class-wide element

    Pace.Signals.Buffers.Get (Queue, Item);
    // Convert from class-wide to specific type
    Pace.Log.Put_Line ("Char = " & Msg_Name (+Item).Char);
    ...

    Msg : Msg_Name;          // other task ████

    Msg.Char := 'c';
    Pace.Signals.Buffers.Put (Queue, Msg);
```

The Buffered Command pattern is useful for composing Command messages out of other Command messages, where the components can be heterogeneously defined. On the receiving end, the individual channeled components can be dereferenced and then polymorphically dispatched to their primitive command operation.



### ***Proxy (Socket) Pattern***

This simple pattern creates a Proxy to enable message sending via a socket. The interface is structurally similar to the pooled asynchronous call, but the class-wide operation redirects to the socket IPC protocol instead before dispatching at the remote site.

```
Msg : Msg_Name;
```

```

...
Pace.Socket.Send (Msg); // use the Ack flag if synching is important

```

The protocol is two-way if the `Send_Inout` or `Send_Out` is used and either synchronous or asynchronous if the `Send` operation is used.

Run-time configuration is provided by the AMIL configuration knowledgebase. Refer to the separate section 7.5.3.4 on this topic. Good use of patterns eliminates the need for "wizard" code-generation algorithms. Whereas marshalling of data is a difficult problem, a command-based proxy pattern is simple to implement and simple to enforce via coding guidelines.



### **Publish-Subscribe (Asynchronous Notify) Pattern**

Publish-Subscribe is effectively a second-order pattern that requires an extra level of protocol on top of the basic command Input pattern. The subscription protocol is much like the Notify pattern; however an intermediate step for maintaining a subscription list is required. To identify the protocol, we subclass a `Subscription` message from `Pace.Msg`.

```

package Status_Pkg ..

type Status is new Pace.Msg with
{
    Data : ...;
};

procedure Input (Obj : in Status);
-- In the body, implement the subscription list:
List : Pace.Socket.Publisher.Subscription_List (Max_Subs); -- Max defaults to 1

procedure Input (Obj : in Status) {
    Pace.Publisher.Subscribe (List, Obj);
}

task Agent {
    Local_Status : Status;

    Local_Status := ... // update the locally persistent state information.
    Pace.Publisher.Publish (List, Local_Status);
}

```

On the client-side, we must subclass from the server message. This can be done in the implementation:

```

type My_Status is new Status with null record;
procedure Input (Obj : in My_Status);

procedure Input (Obj : in My_Status){
    // Grab the data
};

```

```
-- To subscribe, create instance of My_Status, but (important!) convert to Status
Msg : My_Status;
Status_Pkg.Input (Status (Msg)); -- Locally subscribe
```

Only one task is explicitly involved in this pattern, since in the local Agent the task itself is responsible for invoking the callback Command message.  
To subscribe the client remotely, replace the following:

```
procedure Subscribe is new Pace.Socket.Observer
  (Remote => Status_Pkg.Status, -- Needed for remote IPC
   Local  => My_Status);

Subscribe; -- Remotely subscribe
```

The communication pattern is reliable in that it uses TCP sockets when interprocess exchange occurs. Other than that, use this pattern with careful justification in that it can be prone to race conditions if used excessively. The worst-case example of this condition is where every object publishes data asynchronously but there is no central level of synchronized control. The giveaway for this predicament is the liberal use of delay statements to try to mitigate race conditions.

The publish-subscribe pattern is popular but simpler patterns exist. As an alternative, blocking of command pattern events ala the Notify pattern is an easy to code and may find more widespread usage.



### **URL Command Pattern**

Sending a text command to the PACE-friendly web server uses the tagged type semantics in a novel way to do transparent dispatching. If ASCII encoded data is also attached to the text (i.e. URL) message, the protocol pattern is essentially CGI or XML RPC (remote procedure call), REST, or SOAP (simple object access protocol) programming.

```
// Declare a subclass derived from Pace.Server.Dispatch.Action
type Msg_Stimulus is new Pace.Server.Dispatch.Action with null record;
procedure Inout (Obj : in out Msg_Stimulus);

procedure Inout (Obj : in out Msg_Stimulus) {
  Pace.Log.Put_Line ("Stimulus received : " & +Obj.Set);
};

// Register through Save_Action either in a task or at the elaboration section
Save_Action (Msg_Stimulus'(Pace.Msg with Set => +"OK"));
```

The internal web server keeps track of all registered action requests. A typical URL Action Request looks like:

```
http://wcss239:5601/PKG.MSG_STIMULUS?set=OK
```

The bolded part is the salient PACE interpreted URL. Note that whatever package that the *Msg\_Stimulus* dispatching primitive is defined in, this name has to be prepended to the URL. In so doing, the pattern becomes easy to maintain.

The URL pattern allows a backdoor interface to allow the injection of external sensor messages into the executable. In some programming circles, the *Save\_Action* call is part of a factory or a registration pattern, since these objects are created in a factory line fashion, each one stamped out prior to use. Note that the command pattern in general has no such restriction, because by definition they have been registered prior to use by the language run-time. However, when external messages are considered, the run-time has had no chance to create an object instance before the external tag arrives; thus the requirement for the registration process. Moreover, providing this registration allows us to set defaults, as in the "OK" example above.



### ***Message Lookup (Tag Hash) Pattern***

Provides a lookup (i.e. hash or set) table to *Msg* tags. The internal tag of a message is convertible to an external tag (human readable) through the signal *tag* utilities.

Although one can instance this pattern directly, it is used more often in other pattern utilities than on its own.

```
// Instance a Lookup table with hash size.

package Hash is new Pace.Lookup (479); -- Prime number is best

Obj : Msg_Name;

// set a value in Lookup table corresponding to a message tag
Hash.Table.Set (Msg_Name'Class'Tag, +Obj);

// get value and dispatch
Pace.Dispatching.Input (+Hash.Table.Get (Msg_Name'Class'Tag));
```

Note that this uses the Channel Command pattern which manages the memory automatically. Use a Mutex pattern if more than one task will access this; refer to *Pace.Server.Dispatch* for a complete example.



### ***Callback Command Pattern***

Provides a means to attach a class-wide dispatching callback to a command. The callback is contained as a Channel component to the message.

```
-- Server

type Msg_Name is new Pace.Msg with
{
    Callback : Pace.Channel_Msg;
};

procedure Input (Obj : in Msg_Name);

procedure Input (Obj : in Msg_Name) {
    Pace.Socket.Send (+Obj.Callback); // respond on callback
}
```

sg

```

Msg : Msg_Name;

Msg.Callback := Pace.To_Callback(CB);
Pace.Socket.Send (Msg);

```

Note that no data components can be attached to the callback message because the server can only see the anonymous wrapper; the Channel\_Msg "+" operator does the dereferencing on the response callback. If one wishes to add data or simplify the representation, adopt the Notify pattern and think in terms of synchronized "in out" message passing. Or use the Publish Subscribe pattern for multiple responses. This pattern essentially falls out of the command pattern and no extra library code is involved; i.e. it's a freebie.



### **Persistent Command Pattern**

Provides a means to save commands to a persistent storage device, i.e. disk.

```

procedure Input (Obj : in Msg_Name) {
    Copy : Msg_Name;

    Pace.Persistent.Put (Obj); // Save to the Data Store
    Pace.Persistent.Get (Copy); // Retrieve from the Data Store

    // Post-Condition : Copy = Obj

```

The command message is saved in a file which has the same name as the full external tag name of the command message.



### **External Stream Representation Pattern**

Sometimes we may want to override the built-in marshalling (via a Streams library) of the Proxy pattern. We can either compress the data by using block binary transfer, thus overriding the standard byte protocol. Or we can create an ASCII text representation of the stream data to interface to external programs, which may not know all the internal data representations (such as floating point representation).

```

type Status is new Pace.Msg with
{
    Data : ...;
};

procedure Input (Obj : in Status);

package Fast is new Pace.Stream.Binary (Status);
// Input is inherited.

```

Use the binary pattern with caution since it may not reconstruct controlled elements or XDR representations. The text pattern is more involved but important for language interoperability.



### **Configuration Data Pattern**

The guts of this pattern will be covered in detail in a section related to using the Pace to AMIL knowledge base, but this gives the basics.



### **Multicast Pattern**

This is based on an unreliable protocol and should only be used for transient data, such as video or graphics updates. The Receiver object contains a task that dispatches to the correct command destination. The Sender object is a protected object that guards against reentrancy.

```
// loopback example, both sender and receiver in the same context
RX : Pace.Socket.Multicast.Receiver
    := Pace.Socket.Multicast.Create (Pace.Config.Get ("multicast_address", "dvs"));
TX : Pace.Socket.Multicast.Sender
    := Pace.Socket.Multicast.Create (Pace.Config.Get ("multicast_address", "dvs"));

procedure Send_Update (Obj : in Pace.Msg'Class) {
    Pace.Socket.Multicast.Send (TX, Obj);
}
```



### **Shared Memory Pattern**

This pattern was developed for UNIX. It effectively demonstrates how a O/S specific interface can be abstracted away to look like a simple memory access.

```
type Data_Type is
{
    Int : Integer := 0;
    Flt : Float := 0.0;
};

type Data_Block is access Data_Type; // A pointer

Pool : Pace.Keyed_Shared_Memory.Block (Key => 700);
for Data_Block'Storage_Pool use Pool;

// To access memory, dereference the instance
Value : Data := new Data_Type;
```

The Key and Size values need to be identical across applications.



### **6DOF Pattern**

A concrete example of a command message used to interface to an external 3D visualization tool via the multicast pattern. The tool it was originally used for was ProE dvsMockup.

```
// Server Side
```

```

type Position is new Pace.Msg with
{
    Assembly : Str;
    X, Y, Z : Float;
    A, B, C : Float;
};

procedure Input (Obj : in Position);
// body implements the graphics update

// Client Side
TX : Pace.Socket.Multicast.Sender
:= Pace.Socket.Multicast.Create (Pace.Config.Get ("multicast_address", "dvs"));

Msg : Position;

Msg.Assembly := (Str("gyrator"), 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

Pace.Socket.Multicast.Send (TX, Msg);

```

This example gives the flavor of the pattern. Typically, we may abstract the client-side send into a library-level service call, since many concurrently executing objects will update the graphics server simultaneously.



### ***HLA command pattern.***

The FOM representation of messages includes the equivalent of tags. These tags are used to dispatch from the appropriate HLA interactions or object to the command patterns that these represent. The need for sophisticated HLA code builders is substituted for by clever use of the command pattern.



### ***Discrete Event Simulation Pattern***

The two PACE timers (relative delay and absolute delay) can be switched to a discrete event simulation mode by setting a configuration variable

```

Time : Duration = Pace.Now;

Pace.Log.Wait_Until (Time + 100.0); // Wait 100.0 s after the clock time
Pace.Log.Wait (10.0);           // Wait 10.0 s from invocation

```

The call `Pace.Now` returns the current time in seconds from the start of the simulation. For absolute wait, call `Wait_Until` with time in seconds (starting from start of sim). For relative wait, call `Wait` with time in seconds.

If the need arises to set up a timer on a thread, use the `Timer_Start` and `Timer_Expired` services. Like the `Wait` services these are guaranteed thread reentrant;

```

procedure Timer {
    Expired : Boolean;
}

```

```

Pace.Log.Agent_ID;
Pace.Log.Timer_Start (9.0);
Pace.Log.Wait (10.0);
Expired = Pace.Log.Timer_Expired;
-- Post-Condition: Expired = TRUE
}

```

**Server Instancing.** Other container abstractions that can be included in the set of patterns include lists, ring buffers, and specialized server task applications. We keep the latter separate from the other patterns, in that they are not typically used repeatedly in the code, at least not enough to warrant being called a pattern.

### **Web Server Instancing**

Instancing a multi-tasking web server in the main program is needed to use the URL Command pattern. The number of reader tasks is configurable.

```
Pace.Server.Home.Create (Number_Of_Readers => 3, Storage_Size_Per_Reader => 100_000);
```

An embedded web server works very effectively as a *stimulator*. The URL command pattern provides back-door stimulus to code behaviors. The intent is that prototyped GUI interface code to the PACE application can then be invoked through a browser.

### **Knowledgebase Instancing and Use**

Co-simulations will likely require local manipulation of the knowledgebase so that other applications will not interfere with its operation and leave it in an inconsistent state. Further, an AMIL interface needs to be re-entrant safe to allow use as a concurrent API (refer to section 7.5.1.4.1). One way to do this is to wrap the data access within a thread and then provide synchronized access to the API. This kind of server thread has flexibility in that it allows for manipulation of the knowledge locally without requiring calls to the global store. This is extended to the inference engine, as per the ESKER demo, we used the Java-friendly embedded *tuProlog* to provide a conduit to AMIL. Either query calls to Prolog or the AMIL API can then be used to retrieve information from the server thread. For example, the direct query

```
KB.Query (Name, V);
```

asks if there are any matching predicates starting with the lower case Name and followed by the variable list v. The first argument in the variable list v(1) happens to be bound to the value *rd*. The unbound argument v(2) returns the matched knowledgebase value. If a value was not found, then the exception *No\_Match* is raised.

The complement of a query is an assertion. An assertion can be formulated:

```

Fact : String = F("switch", "power"+"off"); // "switch(power, off)."
KB.Assert (Fact).

```

The *Assert* call always succeeds. Once a fact is asserted, it can be retrieved through the query:

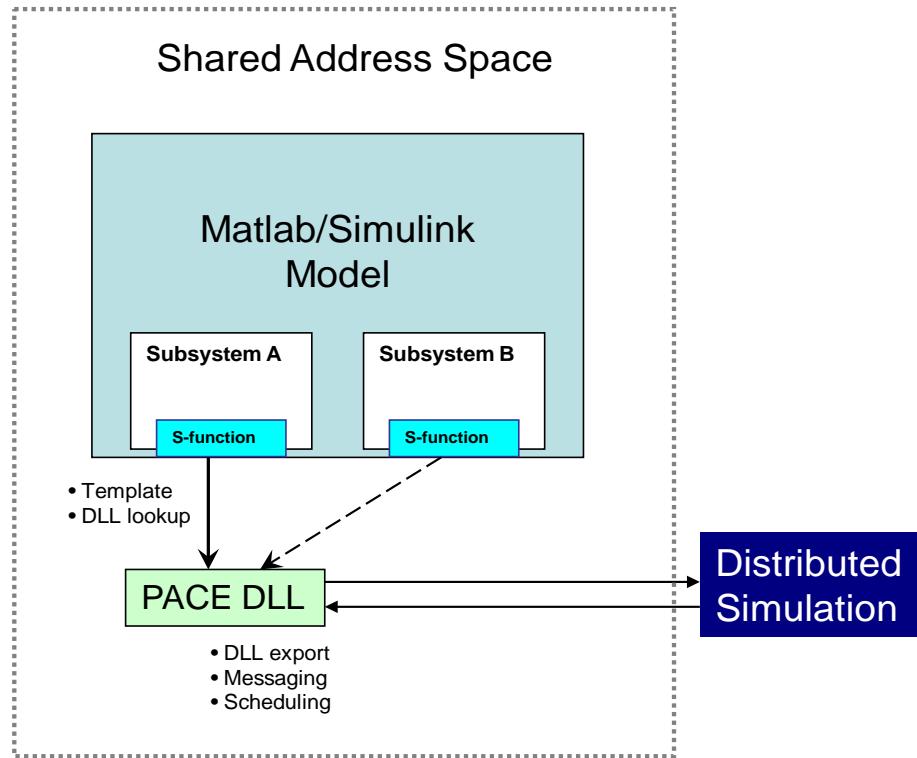
```
V : Variables [1..2];
```

```
V(1) = +"power";  
KB.Query ("switch", V); // converts to "switch (power, V2)?"  
PutLine (+V(2));
```

### **Logical Node Instance**

Each executable in a co-simulation collection needs a unique identifier so that it can be distinguished on a network. A configuration variable called PACE\_NODE takes on an integer value which is used to distinguish between executing simulations. The nodes are typically numbered starting import **PACE\_NODE**=1. The socket-based communication patterns use this node numbering scheme to determine routes. If **PACE\_NODE**=0, then the executable is stand-alone and it is routed internally. Refer to 7.5.3.4

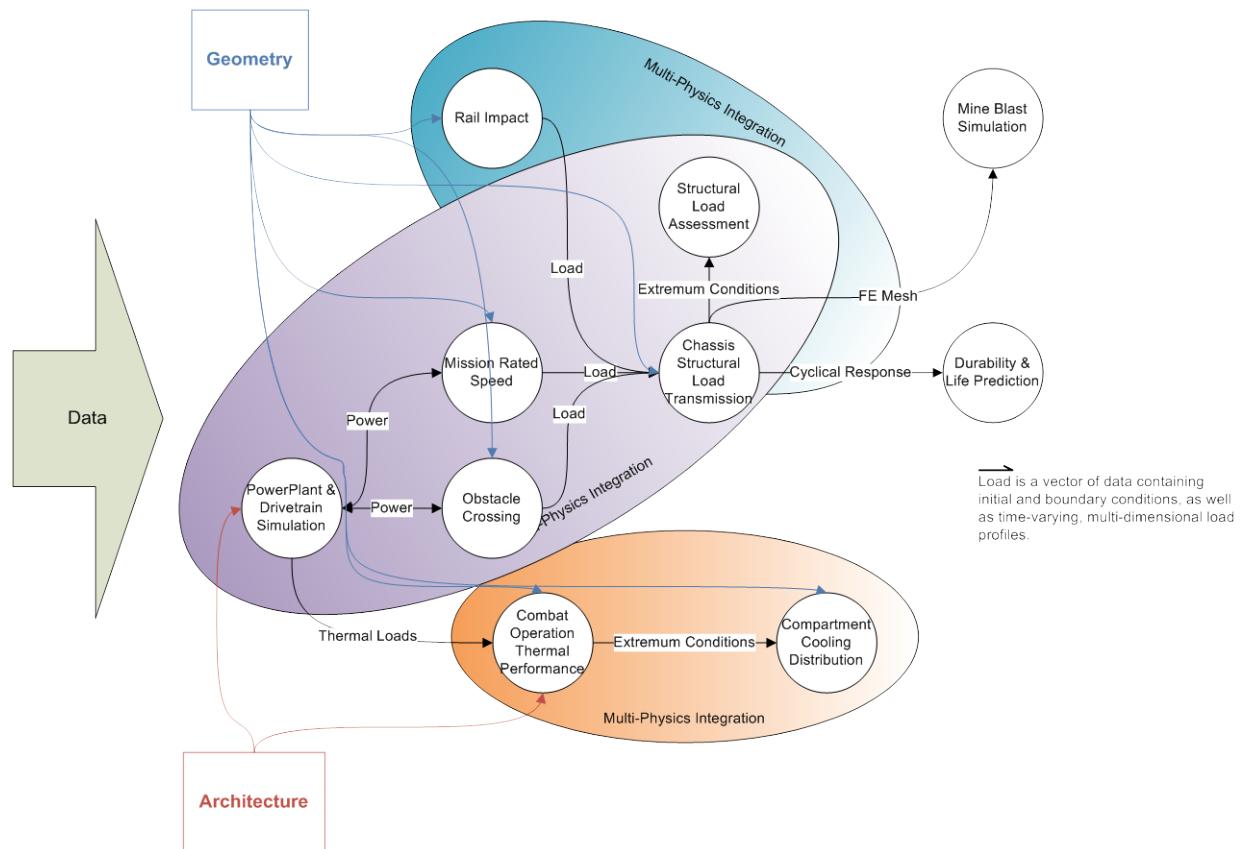
**External Application.** A typical integration of the PACE patterns with a co-simulated application is shown in Figure 7.5-83. This used the *S-function* external API of *Simulink* to link to the distributed command pattern. The rules for creating an S-function are amenable to template-based automation so a reasoner would work well for adapting such external applications



**Figure 7.6-74. Use of Distributed Pattern in Co-simulated Multi-physics Regime**

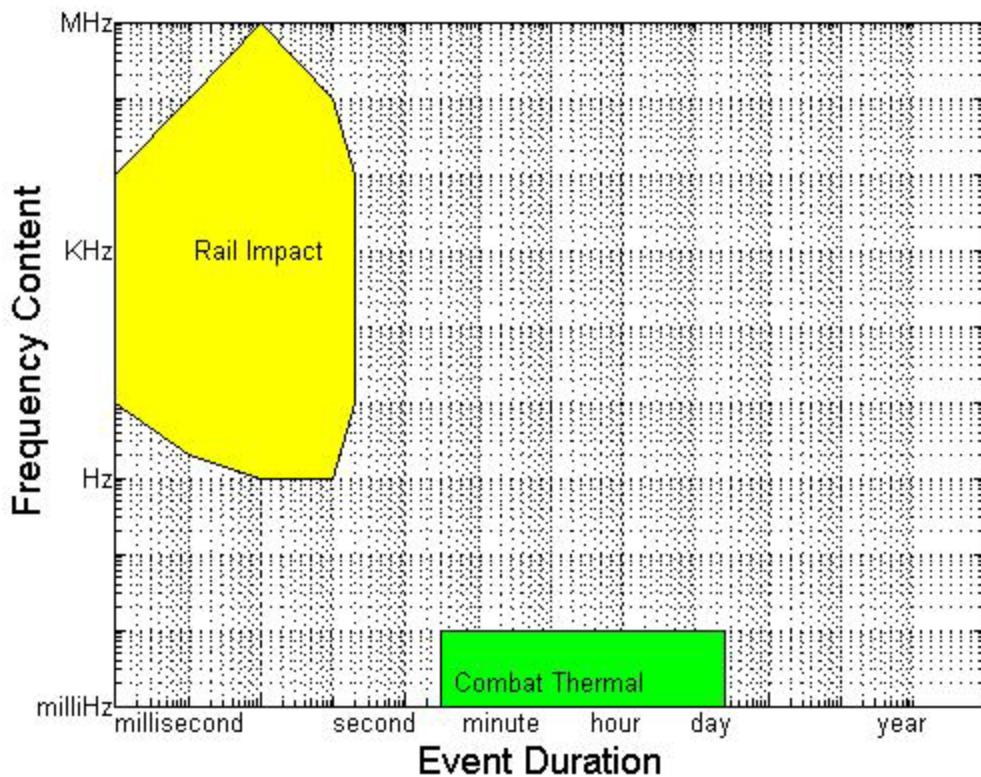
### 7.6.3.3 Multi-Physics and Compartmentalization

To have a chance of working highly-dimensional applied physics problems, a co-simulation will need to be adequately compartmentalized. Without a divide and conquer strategy the dimensionality and fine-grained nature of the problem domain will consume time and increase complexity. The well-known method of *coarse-graining* abstraction can help to improve efficient computation. One recommendation is to do more with first-order models, lumped parameter approximations, and applying principles like energy conservation and entropy maximization.



**Figure 7.6-75. Multi-Physics Data Flow and Integration**

Ideally, we should be able to argue for breaking down any multi-physics problem into separable pieces. They either separate because the actual interactions are small, or that the scales are significantly different, either in time or physical dimension, so that we can safely encapsulate the effects. For multi-scale problems we don't have many options because the computational grids will never overlap across the dynamic range (refer to Figure 7.6-76).



**Figure 7.6-76. Multi-Physics Behavior Often Occurs Over Non-overlapping Time Intervals, Allowing a Separation of Concerns**

Breaking down these processes into workflows, both concurrent and sequential, is critical. We don't necessarily have to always solve the problems collectively, but rather organize them into computational blocks. The blocks could have interfaces with minimal interactions, annotated with rationale explaining why an interaction is minimized. They form the preconditions or assumptions of the assume-guarantee contracts that we will eventually want to employ.

A good example from the META challenge problem is the Rijke tube RLC problem as a partitioned multi-physics exercise, separating out the electrical behavior from the acoustic behavior.

The other strategy is to lay out some other possible approaches that can span domains and do multi-physics. One of the potential ways of thinking about the problem is through generalized N<sub>2</sub> diagrams. The following slide is at least a start in that it categorizes the approaches where DSM and N<sub>2</sub> are used. When the information aspects get into it the mix, it then becomes a cyber-multi-physics problem (c.f. 7.6.4).

## Design Structure Matrix interaction types

Type	Interaction	Examples
Spatial	Adjacency and orientation between two elements	Finite element analysis
Energy	Energy transfer/exchange between two elements	(Hybrid) Bond graphs
Information	Data or signal exchange between two elements	$N^2$ diagram, FFT, constraint satisfaction problems
Material	Material exchange between two element	Compartment models, fluid dynamics (VTT), Markov models.

- These all fit into the approach known as *local computation*
- Possibility for generic inferencing and computational reduction
  - Distributed computing
  - Join trees
  - Dynamic programming

**Figure 7.6-77. DSM for Compartmentalizing**

AMIL is a strong candidate for a multi-physics configuration and routing description language. It matches well to the idea of adaptors and the tagged signal model, where the specifications of the data and synchronization are as important as the actual network connectivity. It then gets passed to a run-time for efficient control. AMIL is like the description logic of the Semantic Web, it describes, and is extended by an orchestration or choreography layer, which is described by the workflow archetypes.

### 7.6.3.4 AMIL Configuration and Specification

The table below indicates configurations in which AMIL can be applied across various co-simulation and co-analysis modes and contexts.

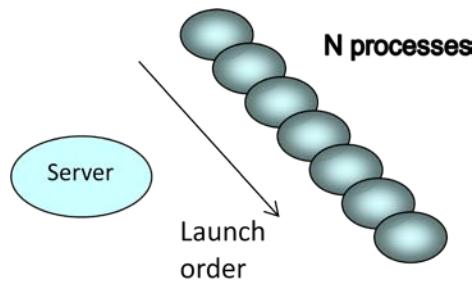
**Table 7.6-4. Categorization of Configuration Parameters for Co-simulation and Co-analysis**

<i>Configuration</i>	<i>Mode</i>	<i>AMIL parameters</i>
<i>Static data sources</i>	Co-simulation	Master model parameters Context model parameters. Items like inventories, plans, routes, etc. Ground truth information like location, heading, etc. Constraints and requirements like max speed, timing durations, etc. Debug flags, logging declarations
	Co-analysis	Decision support rules Model invocations
<i>Run-time dynamics</i>	Launching	Necessary environment variables, command line parameters. Work directories, executable names. Node declarations — Numbered and named hosts — Numbered ports Sequencing and launching order — Sequential — Parallel groups Time scale — Run in real time — Run as discrete-event simulated time model Shutdown and cleanup
	Control Flow	Routing — Direct or Point-to-point — Name-server based — HLA messaging patterns — Web services — Local routing Fail-over Execution — Large-scale simulation is choreography as the primary entities have internal dynamics which control execution — DSE is orchestration, reasoners such as ESKER make the decisions based on valuations.
	Monitoring	Health Diagnostics — Peak — Poke
<i>Automated</i>	Test	Replicated from the static and run-time configuration

<i>testing environment</i>	configuration	
	Scripting and test criteria	These were not necessarily based on the same knowledgebase because testing is an independent verification of the system-under-test.

In this table, we note that AMIL handles data configuration for co-simulations and co-analysis such as DSE.

For launching applications, the general idea of using AMIL to configure co-simulations is to associate applications with logical names and then use those as indexing into the run-time configuration requirements.



**Figure 7.6-78. Launching co-simulation apps**

The top-level triple is indicated as: (*"logical\_name"*, *represents*, *"application"*)

The logical\_name-to-application relation provided is enough to be able to index to the rest of the application information by navigating the ontological knowledgebase graph.

We have retrieved an example from previous work where we used a similar configuration knowledgebase and launching environment called P4 (P4 is similar in intent to ModelCenter and was applied to previous large scale vehicle integration efforts). The idea was to represent our distributed applications (i.e. "apps") by logical names, and then at the command line have a convenient way to specify computer host names in which the apps resided or were launched in (refer to Figure 7.6-47 where it was used to describe an ESKER launch configuration).

Then a collection of apps were simply launched by this invocation:

```
env test=localhost other_test=host2 P4
```

This is an elegant approach because the apps involved were only the ones explicitly referenced at the command line and the connection to the internal triple-store was enabled dynamically through environment variable matching. The rest of the semantic information, related to launch directories, expanded command line parameters, port routing, etc was represented in a more persistent knowledgebase (the equivalent of an AMIL graph).

The challenge is how best to organize the required app information below the top-level logical names/application representation. We ended up using a rules-based approach that would automatically construct secure shell invocations that could then be launched and monitored by the P4 environment.

The box below describes a fragment from a typical session configuration for a vehicle simulation used in an interactive environment.

```

apps :-  

    logicals("ctdb", "bin/Linux/ctdb_server"),  

    logicals("otf", "bin/Linux/otf"),  

    logicals("nabk", "bin/Linux/obj/nato_abk_server"),  

    logicals("ssom", "cannon_main_incl"),  

    logicals("test", "cannon_setup.py"),  

    logicals("crew", "cannon.py"),  

    logicals("crew", "../../../../bin/Linux/cannon").  
  

%%-----  

%% TEST LAYER  

app(Env, ".", "cannon_setup.py", [ "" ], Instance):-  

    getenv(Instance, Host),  

    Env = [ "PYTHONPATH" = "../../../../Common/ssom/pym/",  

            "LD_LIBRARY_PATH" = "../../../../Plugin/i686-linux/python-2.6.2/lib",  

            "PATH" = "../../../../Plugin/i686-linux/python-2.6.2/bin:",  

            "TEST_DEBUG" = 0,  

            "MODEL_HOST" = Host,  

            "MODEL_PORT" = 5601].  


```

**Figure 7.6-79. Logical Triples and Tuple Configuration**

The logicals represent the triple mapping from names to applications. The app rules are specified by the tuples:

$(envVars, relativeDir, application, commandLineParams, logicalInstance)$

A combination of AMIL lookups and a workflow process reasoner allows us to automate the configuration and launching environment. This approach is nicely aligned with the goals of co-simulation and T&V in general and having a semantic web information store in particular.

#### 7.6.3.4.1 Collapsing a Co-Simulation

The META goal is to rapidly create a contextual simulation of a vehicle that a team of engineers can reason about and use to work out design issues. This simulation must expand to incorporate details that will have real effects on the development direction, but can also contract to address top-level requirements constraints. This section describes a Model of Computation and Communication (MoCC) pattern to accomplish this goal.

**The concept and dilemma of distributed time.** Constructing a large simulation will always confront the developers with an interesting dilemma – does the simulation grow to serve the purpose of incorporating as many heterogeneous design elements as possible, or does it limit its extent to allow quick-turnaround experiments that better answer high-level requirements and concept of operations questions? Or is it possible to do both?

The classic case of the latter is that of a vehicle performance timeline model. A timeline model serves as a reference simulation that substantiates that a current design is operating within requirements specifications. In a realistic situation this may involve an extended period of operation, yet the purpose of the simulation is to generate quick feedback to indicate how well the concept works within the intended design. The intended design is thus referred to as the reference architecture. This idea also comes up in the EDA world, whereby a large VHDL model has to provide complete verification of the digital logic and do that efficiently.

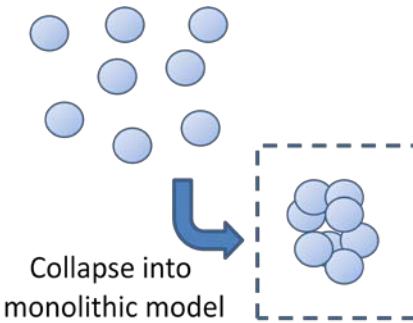
The objective then is to generate a simulation that fully addresses the complexities of the actual design, with all the attendant context models and high-fidelity component simulations, while at the same time able to compress or contract down to a level of fidelity amenable to quick turnaround studies.

We have documented a simulation approach that enables this kind of “morphability”<sup>22</sup>. This adapts well to the ideas of the ARRoW architecture we have in place, in particular with the AMIL structure, and also to the MoCC that is advocated by other META participants. For the time being, let us refer to the approach as the Node-0 model.

**The Node-0 model.** A large simulation when constructed as a heterogeneous mix of components will typically get spread around a set of computing nodes. These nodes could reside on different computers or as different processes on the same computer. The nodes are usually identified by logical entities, typically by names or numerical addresses dependent on communication mechanisms. As the simulation grows, the number of components will likely get spread amongst the nodes to relieve processing demand or for architectural reasons (say a dedicated server is situated on a certain computing node).

Unfortunately, this dispersion of computational or simulation resources makes it difficult to execute discrete-event type models that can potentially complete with a quick turnaround time. The notable issues have to do with network communication overhead and the lack of an efficient distributed simulated clock.

The solution to this problem is to collect all the distributed nodes into the base node, which we call Node-0, and then execute simulations in this more limited and monolithic context. This is straightforwardly accomplished through a few elementary architecture patterns – in the past we have applied the distributed command pattern to provide the dual simulation approaches.



**Figure 7.6-80. The Distributed Simulation is Collapsed into the Monolithic Model “Node-0”**

In the most general case, the idea is to reroute communication paths that leave the main simulation and instead redirect the destination to objects that reside locally. In the specific case

<sup>22</sup> In the past, the team used an AMIL-similar semantic layer which we called the “rule processor” or “KBase” to do model data configuration, model interconnect routing form the distributed command pattern, and model launching, monitoring, and shut-down. So it essentially could choreograph the entire distributed simulation from startup to shut-down. It could also do automated system testing and allowed for the Node-0 model.

of the distributed command pattern, the utility function *send\_message* that all messages get routed through simply has to realize that it is running on the specially-named “Node 0” and then the message gets dispatched locally. A stub or low-fidelity simulation of the destination object is all that is required to maintain the model.

The key to successfully implementing this approach is to assign enough high level system coordination logic to certain objects such that the simulation executes real behaviors and scenarios. Then when the time comes to expand the simulation to a distributed context, these objects remain and they simply exchange information with their higher fidelity representations.

This could turn into just another potential architectural approach if not for its compatibility with the AMIL semantics for defining nodes and edges. The intent of AMIL is to potentially not have to call remote nodes if valuations are available locally, either through cached values or via a simpler representation. In this case the edge points to a local destination for a resource and the distributed overhead disappears.

Whether we can use this mechanism or alternatively use the AMIL graph database as a distributed command pattern routing table, we will get the same architectural benefit, which is to straightforwardly reduce our simulation scope to a more compact context.

This application of the distributed command pattern fits well with the tagged signal model and the contract-based components championed by the IBM META-II team.

#### **7.6.3.5 Probabilistic Certificate of Correctness**

One of the primary objectives of a detailed co-simulation is to provide a basis to calculate a PCC, or how to reason about the assume-guarantee contract of a PCC. In one sense, the latter is an inverse of the PCC calculation, such that we can use environmental data to describe appropriate operating regimes. For the case of vehicle mobility and operational regimes, the obvious case is to exclude regions of ridiculous extremes, such as >70 degree slopes. The assumption then is to provide that constraint along with a probability distribution of the inner/lower range. The verification of drawing from this distribution will then guarantee results with a given PCC.

##### **7.6.3.5.1 Basis for a PCC**

Assume a probability distribution  $P(x)$  for some parameter variant  $x$ . This parameter is exogenously defined and is known to affect the vehicle or system design. To determine its impact, draw a sample from  $P(x)$ , such that  $x$  will feed into a parametric design model and thus result into a potential degradation in performance or correctness.

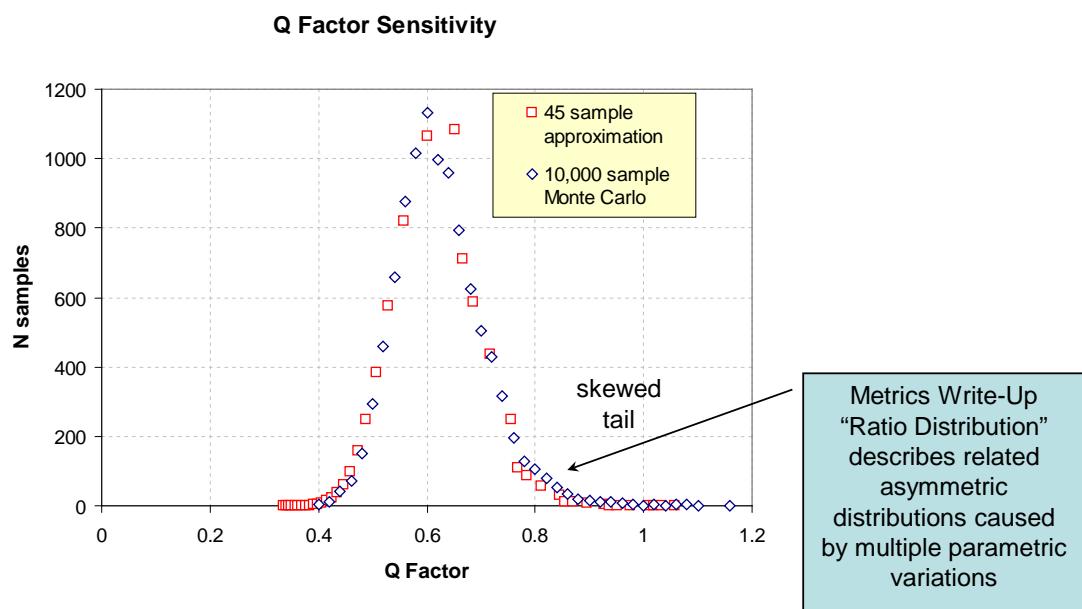
Next, choose a probability level  $P_1$  that the design should withstand under expected operating conditions and a spread in variant values. After collecting enough samples from  $P(x)$  to obtain sufficient statistical certainty, determine whether the sampled data  $P$  exceeds the threshold set by  $P_1$ .

If  $P$  does not exceed  $P_1$  then make the new assumption that the variant cannot exceed a value  $x'$ . Next redraw from the truncated distribution  $P'(x)$  which excludes parameters outside the range of  $x'$ . Iterate until the sampled data  $P'$  does not exceed  $P_1$ . This will provide a means of generating an assume-guarantee contract with  $PCC=P_1$  for the exogenous parameter of interest. This is a sufficiently general technique to apply to a number of stochastic variates with well-characterized and credibly modeled probability distributions, whether or not they follow

normal Gaussian distributions. For multi-variate problems, such as the RLC problem identified as a challenge problem, the approach is straightforwardly extended (refer to Figure 7.6-81 below).

## RLC Circuit characterized by Q Factor

*Sensitive to selection of R,L,C component values*



**Figure 7.6-81. A Multivariate PDF Drawn from 3 Normal Distributions and Applied to Solving a Quality Factor, Q**

This is similar to how the 95th percentile calculations are done for human factors designs. The specifications exclude the top 5% height soldiers as an assumption and then guarantee that the design will work with the bottom 95% soldiers with a PCC=1. Making the PCC less than 1 for this case will allow a wider range of soldier heights, but with the lower PCC as a caveat.

### 7.6.3.5.2 Generic Test Space Exploration

The general approach to including uncertainty (both aleatory and epistemic) is to sample from probability distributions which map to the system under study. For practical PCC purposes, probabilities can only arise in a limited set of ways:

1. Due to variations of design parameters that are not handled digitally by software. So this can include manufacturing variations and quality variations (i.e. why the part is cheap)
2. Due to failure mechanisms in the design which can occur randomly.
  - a. Can include wear-and-tear and spontaneous failures
  - b. Can include random failures built in by the engineer, and thus covers software flaws to a degree.

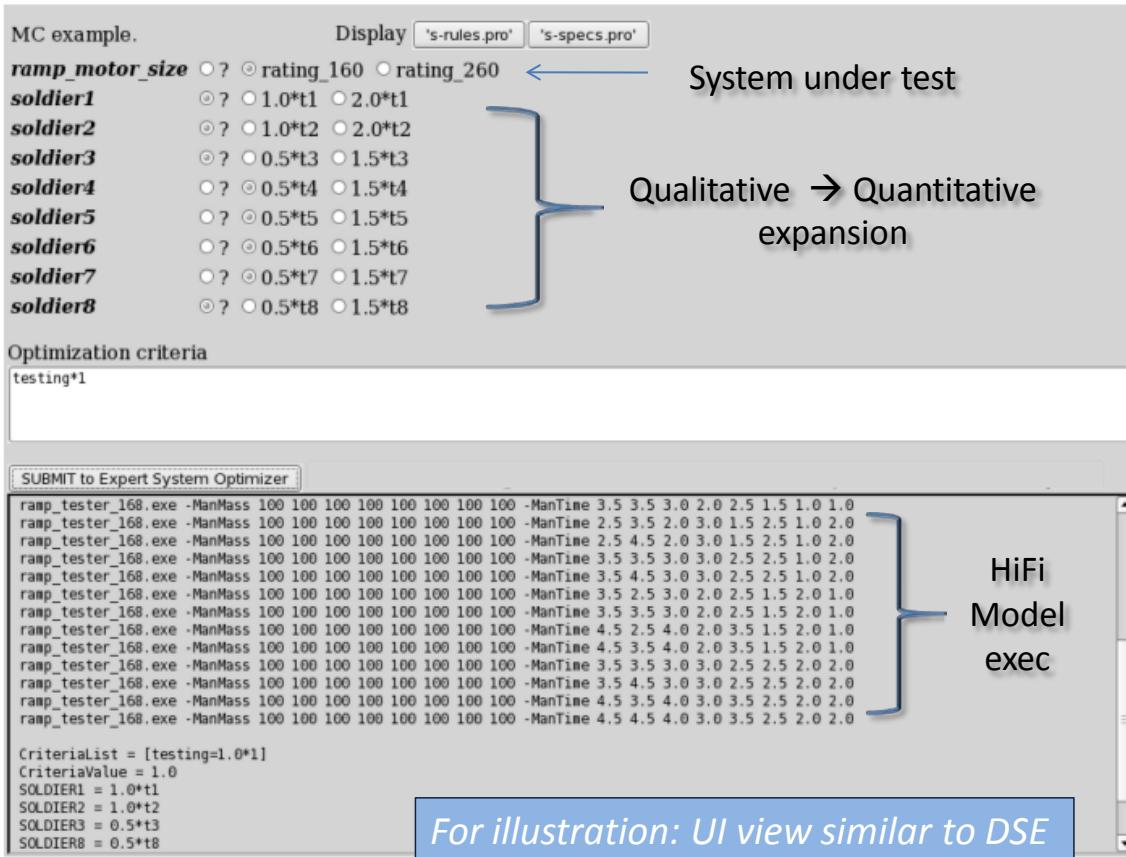
3. Due to excursions in the contextual environment that occur randomly and are outside the design envelope.
4. Due to human operational error, slow reaction times, etc. (which may be the same as #3 if the human is considered context)
5. Due to uncertainties of the epistemic variety arising from poor statistics, etc.

For the ramp challenge problem, we followed these steps:

- Set up model of environmental stimulus as a disturbance profile
- Choose samples both nominal and extreme to capture state space
- Apply model to state space points to establish importance sampling scaling
- Select Pass/Fail criteria for the subject under test
  - *Example:*
  - Full Simulink model of drive (control) and ramp (plant)
  - Torque limit on drive establishes Pass/Fail
  - Disturbance profile from state space supplants the plant model
- Single pass sweep
  - Finds worst case across the stimulus state space
  - While continuously updating the PCC

#### **7.6.3.5.3 Sampling Approaches for Verification**

The limitation of importance sampling is that it does assume a convex optimization problem and the set of situations that this intersects is not comprehensive. In many concave problem domains, which can contain many nooks and crannies, significant amounts of computational horsepower is required to verify the full ergodic state space, simply because of the combinatorics involved. For the ramp problem, we minimized the test space (refer to Figure 7.6-19) to reveal potential problems. The ESKER tool was used to sample the outcomes of a Simulink simulation, injecting different input disturbances for each trial, while at the same time keeping track of a Bayesian update of the final PCC (refer to Figure 7.6-82). The disturbances were importance sampled based on a prior likelihood of operation occurrence. This is in the spirit of the qualitative state plan (QSP) formulated by Hofmann, Robertson, and Williams. The QSP is converted to a PCC with the introduction of simulation results and quantitative priors for the context model of soldier disturbances.



**Figure 7.6-82. Test Space Evaluation Using ESKER to Map Importance Sampled Test Cases**

The algorithm contained in the importance sampled Bayesian reasoner is shown below.

```
% general constraint rule giving unity weighting if TRUE
binary_constraint(Rule, 1) :- call(Rule), !.
binary_constraint(_, 0).

bayes_update(Rule,Likelihood) :-
    binary_constraint(Rule, Value),
    current_pcc(Current, N),
    M is N + Likelihood,
    Result is (Current*N + Value*Likelihood)/M,
    retract(current_pcc(Current,N)),
    asserta(current_pcc(Result,M)).

%% example: bayes_update(X<Max,P)?
```

**Figure 7.6-83. Bayes Update of PCC Applied During Reasoning**

## 7.6.4 Distributed Computing Speed-Up Potential

We anticipate that new architectures would be required as the computational space for co-analysis and co-simulation grows. We consider two views below, with the second placing emphasis on the generic and uniform expression of the computations.

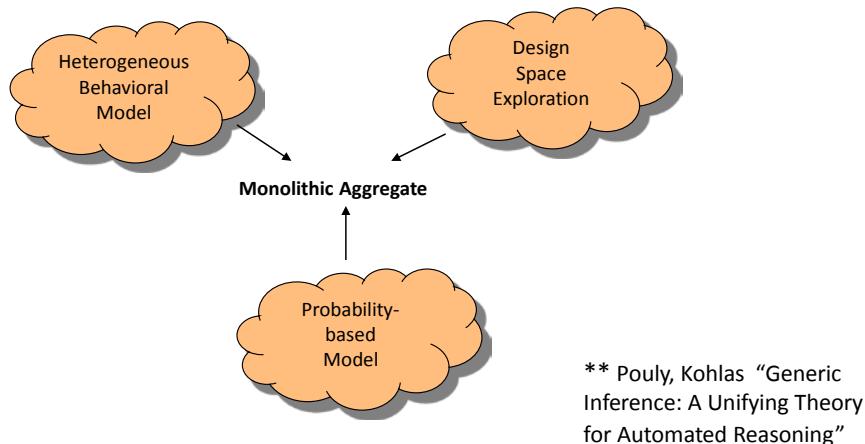
### 7.6.4.1 Spatial Computing

Spatial computing defines a network space that takes advantage of near-neighbor topology which can conceivably improve computational efficiency for co-analysis and co-simulation. It allows a large space of potential human interactions that can affect a vehicle's design and operation to be programmed as a continuous space and compiled to discrete software agents in an agent-based simulation. The description of spatial computing within the context of a prototyping tool called Proto and a visualization engine called Unity is discussed in another appendix. This becomes an alternate strategy to the co-simulation approaches.

### 7.6.4.2 Generic Inferencing

Related to the concept of spatial computing, certain problems can be expressed as a network of local computations [PK11]. The network is constructed to meet some objective such as solving a problem of near-neighbor interactions, or of computing optimization or performing inferences such as occurs in design space exploration. These are particularly well-suited for probability formulations, as that is the best way to add uncertainty to an inference problem. Figure 7.6-84 represents the ultimate goal of creating a simulation model that can solve multiple classes of design and verification problems.

- PACE belongs to a MoCC referred to as *local computation* \*\*
- Computational classes called *valuation algebras* can generically work out optimization and inferencing problems.



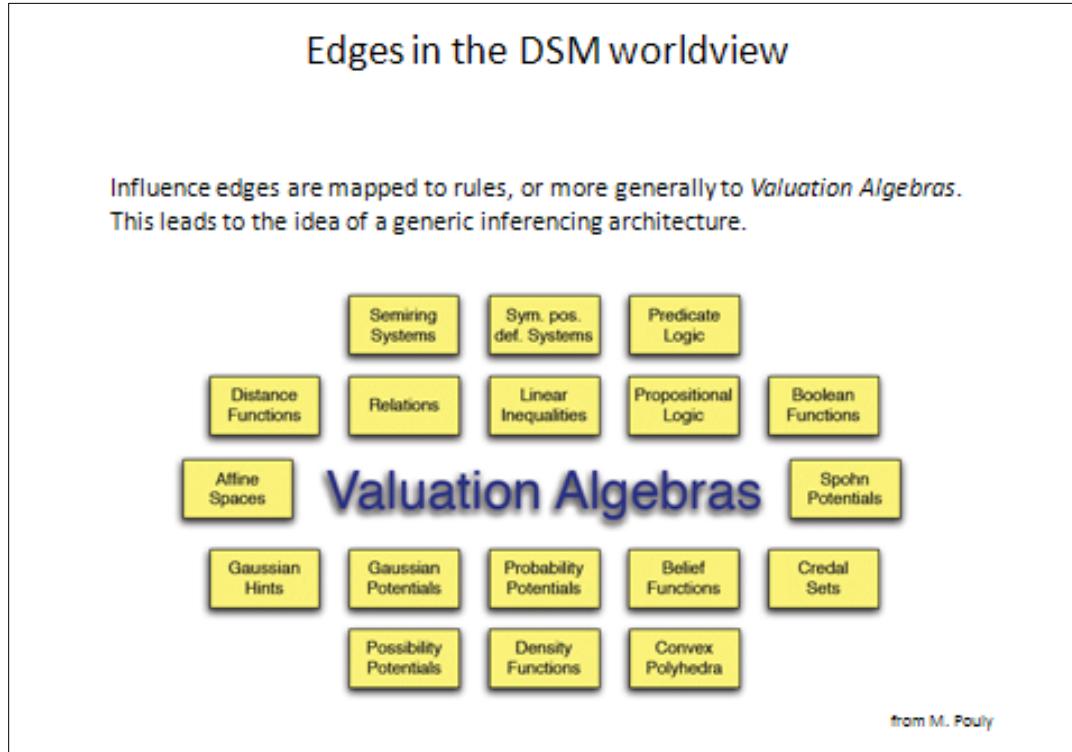
**Figure 7.6-84. Generic Reasoning Allows Several Different Approaches to Potentially Be Unified**

The unified view encompasses the concept of valuation algebras which have an associated notion of a *solution*, *best* or *preferred* value with respect to some criteria:

- In logic, whether a proposition evaluates to true

- With constraints, if it satisfies the requirements
- In optimization, solution leads to maximum or minimum values
- Under uncertainty, if sufficient margin exists from the worst case to failure

In terms of a design structure matrix, all interactions or influences take place over edges.



**Figure 7.6-85. Valuation Algebras Used in Generic Inferencing Run the Gamut of Decision Theory**

Once certain utility functions are created and normalized to create a multi-objective criteria this can support virtually any kind of inferencing, as shown in Figure 7.6-85.

Using the same approach for incorporating a range of utility functions leads to the notion of a generic framework for decision support (DSE) and PCC evaluation. The validity of this concept was demonstrated as evaluations for both DSE and PCC evaluation<sup>23</sup> were accomplished *with the same tool*: as a combination of AMIL + ESKER suggested that GEAR reasoners can provide a generic inference framework.

<sup>23</sup> The valuations considered propositions + hard constraints + continuous values + probabilities

### 7.6.5 Bibliography

- [AUK10] Aksamija, A., Ue, K., Kim, H., Grobler, F., Krishnamurti, R. (2010) "Integration of knowledge-based and generative systems for building characterization and prediction", *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, vol 24, pp. 3.16.
- [BC00] Baldwin, C., Clark, K., (2000), *Design Rules: The Power of Modularity*, MIT Press.
- [BCW97] Barley, M., Clark, P., Williamson, K., and Woods, S., (1997), "The Neutral Representation project", *AAAI Spring Symposium on Ontological Engineering*.
- [BL08] Bunus, P., Lunde, K., (2008), "Supporting Model-Based Diagnostics with Equation-Based Object Oriented Languages", *2<sup>nd</sup> Intl. Workshop on Equation-Based Object-Oriented Languages and Tools*.
- [GW06] Garber, R. Wassim, J., (2006) "Control and Collaboration: digital fabrication strategies in academia and practice", International Journal of Architectural Computing, Volume 4, Issue 2, pp 121-143.
- [HDJ08] Heinecke, H., Damm, W., Josko, B., Metzner, A., Kopetz, H., Sangiovanni-Vincentelli, A., Di Natale, M., (2008), "Software Components for Reliable Automotive Systems", *Design, Automation and Test in Europe*.
- [JP05] Pearl, J. (2005), "Influence Diagrams – Historical and Personal Perspectives", *Decision Analysis*, Vol. 2-4, pp 232-234.
- [KDG11] Kuhn, O., Dusch, T., Ghodous, P., and Collet, P. (2011), "Knowledge-Based Engineering Template Instances Update Support", Lecture Notes in Business Information Processing, Springer, Volume 3, page 151-164.
- [KKM09] Kim, A., Kang, M., Meadows, C., Ioup, E., Sample, J. (2009), "A Framework for Automatic Web Service Composition", Naval Research Lab, DTIC ADA499917.
- [LEH67] Hargrave, L.E., (1967) *Application of Redundancy Study*, Voyager Project, NASA CR-89703 (N67-40411), Jet Propulsion Laboratory
- [LS09] Lukacsy, G. and Szeredi, P., (2009), "Efficient description logic reasoning in Prolog: The DLog System", *Theory and Practice of Logic Programming*, vol.9 (3), pp. 343-414.
- [LP06] Ludwig, L. and Pukite, P., (2006). "DEGAS: discrete event Gnu advanced scheduler", Proceedings of the 2006 annual ACM international conference on SIGAda, ACM SIGAda Ada Letters, Vol XXVI Issue 3 , ISBN 1-59593-563-0.
- [MAP09] Malak, R., Aughenbaugh, J.M., and Paredis, C.J.J., (2009), "Multi-Attribute Utility Analysis in Set-Based Conceptual Design", *Computer-Aided Design*, Volume 41 Issue 3.
- [MD01] Denny, M., (2001), "Introduction to importance sampling in rare-event simulations", *European Journal of Physics*, vol.22, pp. 403-411.
- [META11] META (2011), Phase 1a Final Report: "META Adaptive, Reflective, Robust Workflow (ARRoW)", TR-2683 .
- [PE94] Pimpler, T.U., Eppinger S.D., (1994) "Design Theory and Methodology", ASME Conference, Minneapolis.

- [PK11] Pouly, M. Kohlas, J., (2011), *Generic Inference: A Unifying Theory for Automated Reasoning*, Wiley .
- [PRP94] Pukite, P.R.. (1994), “Advanced Design Tools for Evaluating Fault-Tolerant Systems”, NASA Langley Research Center, contract NAS1-20035, <http://foia.larc.nasa.gov/contracts.cgi>
- [PL07] Pukite, P., and Ludwig, L., (2007). “Generic discrete event simulations using DEGAS::application to logic design and digital signal processing”, Proceedings of the 2007 ACM international conference on SIGAda, ACM SIGAda Ada Letters, Vol XXVII Issue 3, ISBN 978-1-59593-876-3.
- [PTM03] Pukite, P. R., Thomas, B. E., Mostek, C. M., Challou, D. J., Quinn, M. A., Wentland, C. J., Sallman, W. K., (2003). “A Comprehensive M&S Strategy To Speed FCS Software Development”, submitted to *SMART Conference*, Dearborn, MI
- [QGP10] Quinton, S., Gaf, S., Passerone, S., (2010), “Contract-Based Reasoning for Components Systems with Complex Interactions”, *Verimag Research Report*.
- [RK06] Krishnamurti, R., (2006), “Explicit design space?”, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 20, 95–103.
- [RKT05] Russomano, D.J., Kothari, C.R., and Thomas, O.A., (2005), “Building a Sensor Ontology: A Practical Approach Leveraging ISO and OGC Models”, The 2005 International Conference on Artificial Intelligence, Las Vegas, NV.
- [SMV11] Sensoy, M., Mel, G., Vasconcelos, W. W., Norman, T. J.. (2011), “Ontological logic programming”. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics (WIMS '11)*, ACM
- [YGD10] Yue, P., J. Gong, L. Di, (2010), “Augmenting Geospatial Data Provenance through Metadata Tracking in Geospatial Service Chaining”, Computer and Geosciences. Volume 36, Issue 3, pages 270–281.
- [YRG10] Yang, C., Raskin, R., Goodchild, M., and Gahegan, M., (2010), “Geospatial Cyberinfrastructure: Past, present, and future”, *Computers, Environment, and Urban Systems*. Vol. 34, pp 264-277
- [ZZZ05] Zhao, Y.Z.; Zhang, J.B., Zhuang,L. and Zhang, D.H. (2005), “Service-oriented architecture and technologies for automating integration of manufacturing systems and services”, *10th IEEE Conference on Emerging Technologies and Factory Automation*

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.7 – Metrics (BBN)**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.7 BBN Resource Contention.....</b>	1
7.7.1 BBN Metrics and Modeling Language Activities.....	1
7.7.1.1 Metrics Definition.....	1
7.7.1.2 Resource Contention Complexity Metric.....	1
7.7.1.3 Process Complexity Metric .....	2
7.7.1.4 Component Model Language.....	4
7.7.1.5 Resource Sharing Concept.....	4
7.7.2 Categories and Attributes.....	5
7.7.2.1 Resource Provisioning .....	6
7.7.2.2 Resource Availability .....	7
7.7.2.3 Resource Consumption .....	7
7.7.2.4 Assessment of Resource Usage.....	7
7.7.3 Shared Resource Ontology.....	8
7.7.3.1 Shared Resource Ontology Classes.....	8
7.7.3.2 Shared Resource Ontology Object Properties .....	9
7.7.3.3 Shared Resource Ontology Data Properties.....	11
7.7.4 Extending the Shared Resource Ontology for Contention Complexity.....	12
7.7.5 Example Applications of the Shared Resource Ontology Extended to Contention Complexity.....	13
7.7.5.1 RLC Toy Problem .....	13
7.7.5.2 Preliminary Ontology for Hybrid Vehicle Battery Use and Maintenance Analysis.....	14
7.7.6 Prototype Toolchain for Metric Evaluation .....	16
7.7.6.1 GAMETE Executable Framework .....	17
7.7.6.2 Unified Data Representation.....	18
7.7.6.3 Metrics Supported by GAMETE.....	20
7.7.6.4 Metric Evaluation Demonstration.....	21
7.7.6.5 Integration of GAMETE with Toolchains.....	22
7.7.7 Conclusion.....	22

## List of Figures

Figure 7.7-1. A Tree Representation of the Ontology Class Hierarchy .....	8
Figure 7.7-2. A Tree Representation of the Ontology Object Property Hierarchy.....	10
Figure 7.7-3. Class and Object Property Relationships.....	11
Figure 7.7-4. A Tree Representation of the Ontology Data Property Hierarchy .....	12
Figure 7.7-5. RLC circuit .....	13
Figure 7.7-6. Chart Showing Contention Complexity for Multiple Battery Types and Depths of Discharge.....	16
Figure 7.7-7. GAMETE Architecture Design and Toolchain Interface .....	17
Figure 7.7-8. The Experiment/Simulation Engine feeds data from diverse sources for online or offline processing.....	18
Figure 7.7-9. Measurement storage format in the Unified Data Representation.....	19
Figure 7.7-10. Representing signals, graphs, and resource-consumption in the UDR maximizes the applicability of implemented metrics.....	20
Figure 7.7-11. Two control signals, one from a high-gain controller with periodic steady-state behavior and one from a low-gain controller with DC steady-state behavior.....	21

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
FTE	Full Time Equivalent
GAMETE	General Adaptable Metric Execution Tool and Environment
OWL	Web Ontology Language
QoS	Quality of Service
TCP	Transmission Control Protocol
UDR	Unified Data Representation
UIR	Uniform Resource Identifier

## 7.7 BBN Resource Contention

### 7.7.1 BBN Metrics and Modeling Language Activities

This portion of the report covers the BBN activities to support metrics and modeling language activities. In particular, we:

1. Review our phase 1a work on metrics for resource contention complexity and process complexity that we use as a basis for our component model language and prototype capability to host and evaluate metrics.
2. Define a component model language.
3. Develop a prototype capability to host and evaluate metrics.

The component model language and prototype capability to host and evaluate metrics are alternate approaches to the BAE metric framework. These alternative approaches were explored to investigate additional considerations for future work in these areas based on a semantic approach to utility analysis.

#### 7.7.1.1 Metrics Definition

In this section we review our Phase 1a notions of resource contention complexity and process complexity inspired and informed by prior work to define a utility metric for Quality of Service (QoS) maintenance. This work forms the basis of our component model language and our General Adaptable Metric Execution Tool and Environment (GAMETE) software we developed under Phase 1b.

#### 7.7.1.2 Resource Contention Complexity Metric

The contention complexity of a system can be decomposed based on resources. I.e., the contention of one resource does not directly impact the contention of a different resource. Hence, we define each resource to have its own resource complexity measure. We codify these and other motivating hypotheses of contention complexity as follows:

1. Entities/subsystems/components in a system use multiple resources. Entities have varying levels of criticality for using specific resources.
2. Contention complexity is a function of the potential for contention due more requests from entities to use limited resources.
3. Contention complexity can be decomposed and expressed for specific resources.  
Contention complexity is the sum of the contention complexities of resources. The contention complexity of a resource is a function of the potential for contention of that resource from entities to use that resource.
4. Contention complexity of a resource is a function of:
  - a. The number of entities that could request that resource. (A resource with more users leads to more contention complexity.)
  - b. The level of usage required for use of the resource. This is measured in terms of % usage level \* amount of time per usage. (Higher usage level means more contention, more time per usage means longer queues and more contention.) The higher this product is, the higher likelihood of contention.
  - c. More critical uses of limited resources implies more contention complexity.

We define our contention complexity metric as the sum of contention complexities for specific resources.

$$\text{ContentionComplexity} = \sum_{r \in \text{Resources}} \text{ContentionComplexity}(r)$$

We then define the contention complexity of a resource as:

$$\text{ContentionComplexity}(r) = \sum_{c \in \text{DependsOn}(r)} \frac{E[\%level(c, r)]\text{var}(\%level(c, r))}{\text{criticality}(c, r)}$$

To reason about these tradeoffs and the impact they have on contention complexity, we define the following variables which are used in the above definitions:

- $\%level$  represents the percentage commitment for a resource by a consumer. For instance, if 20 watts are needed, but entities request 30, then there is an over-commitment of 10 watts, resulting in a 50% over-commitment.
- $\text{DependsOn}(r)$  represents all entities that depend on a resource  $r$ .
- $\text{criticality}(entity)$  represents the criticality of an entity. We assume criticality is on the scale between 0 and 1.

### 7.7.1.3 Process Complexity Metric

Process complexity is important to measure because unknown delays in tasks in a process can lead to problems and perturbations in a system's design or operation. The propensity of tasks in a process to be delayed can depend on various measurable task properties including difficulty, quality requirements, schedule requirements, etc. These measures can be difficult measure until tasks are joined together to form a process and can vary for tasks between processes

We codify our hypotheses that inspire our process complexity definition as the follows:

1. A process depends on multiple, possibly repeated tasks. Some tasks cannot be started until the completion of prior tasks. We call these relationships as logical dependencies. A schematic of task dependencies for a process can be seen in Figure 7.7-1 where a process has several initial tasks, a single final task and multiple dependencies shown as arrows.
2. Each task has its own complexity, called task complexity. Task complexity is a function of several variables including difficulty, process maturity, schedule and quality, among others. Difficulty represents the amount of skill and resources required to complete a task. Process maturity represents how mature the process is to accomplish a given task. Schedule represents the likelihood of being able to accomplish a task before its deadline. Quality represents the needed quality of work required to accomplish tasks successfully for a given process.
3. Process complexity is directly proportional to the complexity of the processes' tasks and the number of dependency relationships. As the complexity of tasks increases, the complexity of a process increases. As the number of dependencies increase, the complexity of a process increases.

We define our process complexity metric by first defining a task complexity metric. We define our process complexity metric as a summation of task complexity where the form of summations depends on the structure of the dependency relationships.

Process complexity is a function of task complexity and the dependency structure of tasks within a process. Consequently, we define process complexity iteratively with respect to the weighted accumulation of task complexities in a process. More specifically, we define the process complexity at every task in a process as the sum of the task complexity and the complexity of all tasks it immediately depends on. We define process complexity as the process complexity of the final task.

To express this definition of process complexity with respect to a task, we use the expression  $P(i)$  to represent all parents of a task  $i$ . In our example, dependency schematic in Figure 7.7-1, Task 6 has Tasks 3 and 4 as parent tasks. We define the process complexity of a task  $i$  as follows:

$$PC(i) = TC_i + \sum_{j \in P(i)} PC(j)$$

We define task complexity as a function of the difficulty and maturity of processes used to complete a task and the schedule and quality requirements imposed by the process on the task. It is important to note that various processes can be used to complete a task, and process selection involves engineering tradeoffs that impact the ability to complete tasks on schedule and with a given quality.

To reason about these tradeoffs and the impact they have on process complexity, we define the following variables:

- $D(t)$ : difficulty of task  $t$  – Measured by number Full Time Equivalents(FTEs), for example
- $M(t)$ : maturity of task  $t$  – Measured by number of years, for example
- $Q(t)$ : quality requirement of task  $t$  – Measured on scale of 1-100%. This is a user-defined parameter. Could represent importance of high quality or how much functionality is provided if task fails
- $S(t)$ : schedule pressure of task  $t$  – Measured by schedule slack in days, for example. (This could also be a stochastic measure.)

The exact equation for  $TC(\cdot, \cdot, \cdot)$  will vary based on application, but a good initial candidate is:

$$TC = \frac{D(t)}{M(t)*Q(t)*S(t)}$$

Note that when our iterative definition of process complexity is expanded to be an expression of just task complexities, so tasks are weighted more heavily than others if there are many tasks that are dependent on them. This is by design as our intuition is that process complexity should be heavily weighted by the complexity of more dependent tasks. We use this approach to compute process complexity as a weighted sum of task complexities. As such, we can expand the complexity definition for implementation as follows in a tool where the weight on any task complexity is the number of sink-source paths from the task to the final task:

$$PC = \sum_j w_j TC_j$$

Our intention is to develop tools that compute task complexities during design time based on either estimates, simulations, or evaluations of the task complexity data (difficulty, maturity, quality, schedule.) The accuracy of the task complexity computations depends on the accuracy of these estimates.

An issue with this metric is that some of its useful applications can be difficult due to the need to quantify task complexity. For example, a task in a vehicle design process that depends on humans (i.e., the design of user interface aspects or human-in-the-loop operations) may be highly variable due to the skill of the humans involved. The risk mitigation is that the metric does not have to capture everything about how to measure task complexity, simply that the process complexity measure changes with task properties and requirements.

#### **7.7.1.4 Component Model Language**

In this section we introduce a component model language to capture shared resource interactions. In particular, we describe a Web Ontology Language (OWL) ontology we designed to describe resource sharing scenarios and how general metrics are assessed in these scenarios. This resource sharing ontology and its context-specific extensions are used to build composed models of systems components that interact both through directed communications and implicit and explicit resource sharing. Our intention is that this ontology should be modifiable and extensible for scenarios describing resource sharing metrics, allocation and consumption algorithms, consumption models and so forth.

After we outline general resource sharing concepts, we identify general categories of attributes of shared resource languages that should be captured in the ontology. We describe the ontology and discuss ontology extensions for specific resource sharing scenarios such as the assessment of resource sharing complexity. We provide several examples of using the ontology to model applications, such as an RLC circuit and the preliminary approach for the assessment of the complexity of maintenance resource sharing over the lifetime of a hybrid vehicle.

#### **7.7.1.5 Resource Sharing Concept**

Our general shared resource concept is informed by our perception that systems (whether cyber, physical, cyber-physical, or otherwise) are comprised of resources and system actors that interact with and through the resources. This interaction can affect the efficacy and efficiency of the system and therefore must be considered in the design, development, and testing phases of system creation. When interactions are not considered early in system creation, e.g., during the design phase, then they must be tested for during the system verification and validation

phase, which is costlier, or there is a risk that they will be discovered after system deployment, where unanticipated resource interactions and contention that affect the system operation or performance can lead to even more costly system failure or retrofitting. One of the goals of the META program, and the ARRoW project, is to incorporate aspects like resource contention earlier in the system design, to reduce the cost of later phases of system development, testing, and maintenance. In this section we describe the various entities that should be incorporated in developing a shared resource modeling language. These entities are formalized in our shared resource ontology that we describe below.

Resources may include, for example, engine power, fuel, communication bandwidth and cooling capacity. System actors include resource consumers, resource providers and resource allocators. *Resource consumers* use resources – for example an internal combustion engine consumes fuel, or network cards use bandwidth. Note that we consider that resources may be transformed and disappear due to consumption (such as fuel), or resources may be used, but not disappear due to use (such as bandwidth.) *Resource providers* generate resources that are consumed, e.g., an alternator attached to an engine generates electrical energy or a sensor may provide information about object detections. *Resource allocators* decide which resources are used by which consumers. For example, an engine control module determines the rate of fuel flow to an engine and a Transmission Control Protocol (TCP) implementation determines the use of bandwidth used by nodes communicating on an intranet.

Various measurements can be made of system behavior, including the behavior of resources and actors. Those measurements can be used to assess various aspects of system performance. These assessments may be used by the system actors to alter their behavior, or by users to assess system health. For example, a resource allocator such as an engine control module uses measurements including engine speed, temperature, and atmospheric density to assess engine performance and allocate fuel to an engine. Similarly, TCP uses measurements of packet sequence numbers to assess congestion. In the example we discuss below, we assess a metric called resource sharing complexity over various measurements to assess the propensity for resource contention.

The system actors, measurements, and assessments do not necessarily operate continuously like a centrifugal governor operating on a steam engine. Most often system actors, measurements, and assessments are implemented digitally and update either on a clock cycle or when driven by external events. As such, system interactions, measurements and assessments need to be modeled and coordinated with respect to event occurrences which may include clock ticks.

### 7.7.2 Categories and Attributes

Depending on the application, system actors (such as resource consumers, resource providers, and resource allocators) primarily interact through more than shared resources. Components could interact through directed communication, but these kinds of engineered/designed interactions are implemented through resource sharing interactions such as the use of communication buses. Our insights in this document are driven primarily by experience and published reports on developing and using resource sharing models to architect component interactions in information management systems. Based on our experience, (explicit or implicit) shared resource interactions are difficult to capture and express. We see the need for a resource sharing interaction language that would be extensible and compatible with component modeling languages.

Selecting the attributes for a component model language for shared resources requires the selection of an appropriate level of abstraction. Although users of modeling languages may sometimes want or need to understand the inner operations of components, it is often sufficient to exclusively model aspects of component interactions, even when these interactions do not occur through explicitly specified interfaces. Furthermore, it is necessary to abstract from many of the details of the inner operations in order to reduce and manage the complexity involved in system modeling, design, and development. Since components at a given level of abstraction interact explicitly and implicitly through shared resources, a component model language for shared resources needs to capture the relevant properties of resource interactions. We focus on resource interactions because many component interactions can be described as interactions by the components through shared resources.

In this section, we describe the primary functional attributes and categories of attributes that should be captured in a shared resource modeling language. These primary attributes are formalized in our shared resource ontology that we describe below.

We identify four general categories of attributes for shared resource model languages: Resource Provisioning, Resource Availability, Resource Consumption Models, and Resource Usage Assessment. In the section immediately following we motivate and provide an overview of these selections of attribute categories. For each of these categories of attributes, we discuss several attributes with examples that should be included in component model languages for shared resources.

We should note that the attributes and categories we discuss are intended to be representative, and are by no means exhaustive or exclusive. We intend for our taxonomy of attributes and the attribute language schema to be a guide that is further customized for specific applications and systems. The attribute categories we discuss are:

1. Resource provisioning attributes: these attributes cover how the resources are allocated.
2. Resource availability attributes: these attributes cover how the availability of the resources may change after provisioning.
3. Resource consumption attributes: these attributes cover how the resources are consumed by component operation.
4. Resource assessment attributes: these attributes cover how the consumption of the resources are typically evaluated.

All of the attribute categories we list include some aspect of resource constraints. In fact, most of the attributes describe some aspect of constraints on the behavior and use of the resources that need to be expressed in the resource models. These constraints limit the behaviors that need to be accounted for in composing model components through resource interactions.

### **7.7.2.1 Resource Provisioning**

The resource provisioning attribute category contains attributes that describe how resources are allocated. These attributes include:

1. Resource Shared: This attribute captures whether the resource is shared or private.
2. Allocation Decider: This attribute captures who decides on the allocation of resources. Possible values include the system designer, system user, or the system for autoconfiguring systems.

3. Allocation Mechanism: This attribute captures how the allocation is decided, whether by directed allocation from a human-in-the-loop (such as a designer or user), a supervisory controller, or a collaborative resource allocation system in peer-to-peer environments.
4. Allocation Dynamism: This attribute captures whether the allocation is performed only once, or whether reallocation occurs during runtime.
5. Reallocation Trigger: When resource allocations are dynamic, this attribute captures how reallocation is triggered – whether from a clock tick, a feedback control mechanism, or driven by an external event like changes in missions.

### **7.7.2.2 Resource Availability**

The resource availability attribute category contains attributes that describe how resource availability changes over time and how resource consumers and allocators observe those changes in availability. These attributes include:

1. Resource longevity: This attribute captures whether the resource has permanence (like physical space), it is consumed constantly (like time), or its consumption is a function of usage properties (like how fuel is consumed based on performance demands.)
2. Resource availability observability: This attribute captures whether the resource availability is directly observable, partially observable, or unobservable to various entities including consumers, providers and allocators.

### **7.7.2.3 Resource Consumption**

The resource consumption attribute category contains attributes that describe how resource consumption occurs. These attributes include:

1. Resource consumption predictability for consumer: This attribute captures how predictable resource consumption is for consumers – whether it is schedulable (such as fuel availability due to consumption), partially schedulable (such as ammunition for weapons), or un-schedulable (such as armor plating.)
2. Resource consumption predictability for allocator: This attribute captures how predictable resource consumption is for allocators.
3. Resource consumption controllability: This attribute captures whether the resource consumption is fully controlled (like fuel usage), partially controlled (like heat dissipation capability), or uncontrollable (like time.)
4. Resource usage correlation: This attribute identifies when consumption of particular subsets of resources are positively or negatively correlated.

### **7.7.2.4 Assessment of Resource Usage**

The resource assessment attribute category contains attributes that describe how resource consumers and allocators observe both the consumption and need for additional resources. These attributes include:

1. Uses for online resource usage feedback: This attribute identifies the resource consumption assessments that are useful for runtime reallocation and tuning. These assessments include performance utility, control error terms, safety margins, etc.
2. Uses for design-time assessment: This attribute identifies resource attributes to be considered during design-time. These assessments include resource complexity, etc.

### 7.7.3 Shared Resource Ontology

We have designed an ontology to capture the above attributes and categories of attributes of a shared resource language. We followed general ontology design principles to make the ontology as simple as possible so that it is as extensible as possible and widely applicable.

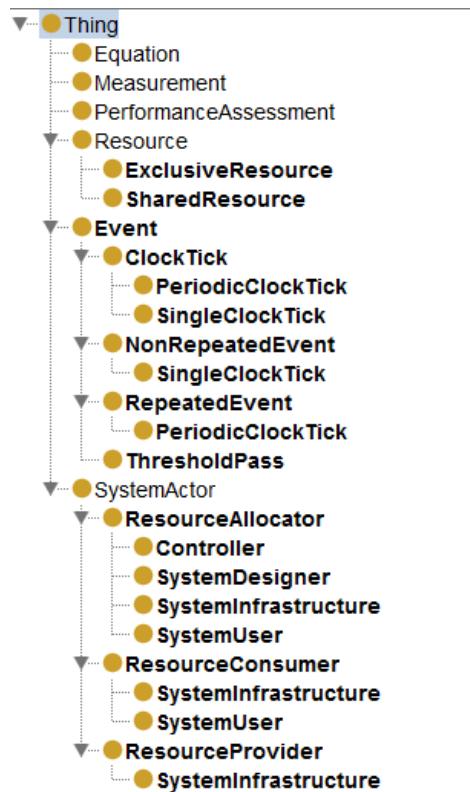
We call this ontology the *shared resource ontology* and assigned it the following Uniform Resource Identifier (URI):

<http://meta.bbn.com/ont/2011/04/shared-resource-ont.owl>

The ontology incorporates the system entities such as resources, system actors, measurements and performance assessments. These entities and several others are represented as classes in the ontology. The ontology also incorporates object and data properties that represent the attributes and attribute categories we discussed above.

#### 7.7.3.1 Shared Resource Ontology Classes

A schematic of the ontology class hierarchy can be seen in Figure 7.7-1.



**Figure 7.7-1. A Tree Representation of the Ontology Class Hierarchy**

There are several main classes of entities in the ontology – *Resource*, *SystemActor*, *Measurement*, *PerformanceAssessment*, *Equation*, and *Event*.

The *Resource* class represents resources. We define two subclasses – *ExclusiveResource* and *SharedResource* to represent resources that are used exclusively by one consumer or multiple consumers, respectively. We expect that shared resources will be prevalent in applications that use this ontology, but that representations of exclusive resources will be necessary for scenarios where some resources are exclusively assigned to safety-critical systems, such as life support in pressurized-cabin air-vehicles.

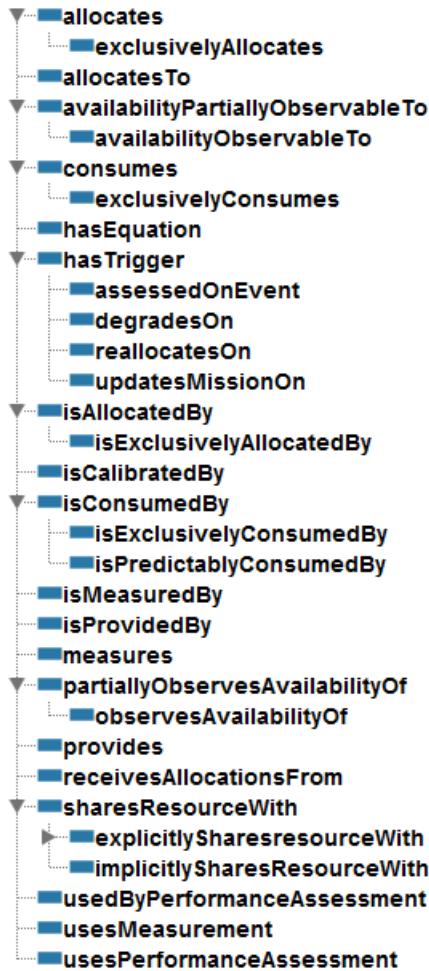
The *SystemActor* class represents the system actors. Subclasses include *ResourceAllocator*, *ResourceConsumer* and *ResourceProvider* that represent, respectively, actors that allocate, consume, and provide resources as discussed above. Subclasses of *ResourceAllocator* include *Controller*, *SystemDesigner*, *SystemInfrastructure*, and *SystemUser*. Subclasses of *ResourceConsumer* include *SystemInfrastructure* and *SystemUser*. *SystemInfrastructure* is also a sub-class of *ResourceProvider*. *Controller* represents resource allocation controllers such as the engine control module discussed above. *SystemDesigner* is the system designer and is used when the system designer allocates resources. For example, a designer may decide that some engine designs have a restricted fuel flow to limit power output and increase longevity. *SystemUser* represents possibly multiple system users which both allocate resources through command decisions and consume resources. For example, the driver of an automobile consumes engine power by accelerating the vehicle. The driver also allocates fuel resources to the engine by adjusting the throttle. *SystemInfrastructure* represents the system which inherently allocates, provides, and consumes resources. For example, modern computer motherboards allocate and provide electrical power to chip components, while also consuming electrical power.

The *Measurement*, *PerformanceAssessment*, and *Equation* classes have no subclasses. Although not introduced earlier, *Equation* represents mathematical expressions that are used by *PerformanceAssessment* objects, among others.

The *Event* class represents discrete points in time that may drive the taking of measurements, running performance assessments, changing allocations and so on. These events may be repeated or not, hence the *RepeatedEvent* and *NonRepeatedEvent* classes. Events could occur at clock ticks (repeated or not), or when a measured value or assessment passes some threshold, as is usually the case when using a performance assessment to decide when to perform a resource allocation – when the expected benefit of a reallocation surpasses the expected benefit of not changing a resource allocation, a controller should use this threshold passing to decide to perform a reallocation of resources.

### 7.7.3.2 Shared Resource Ontology Object Properties

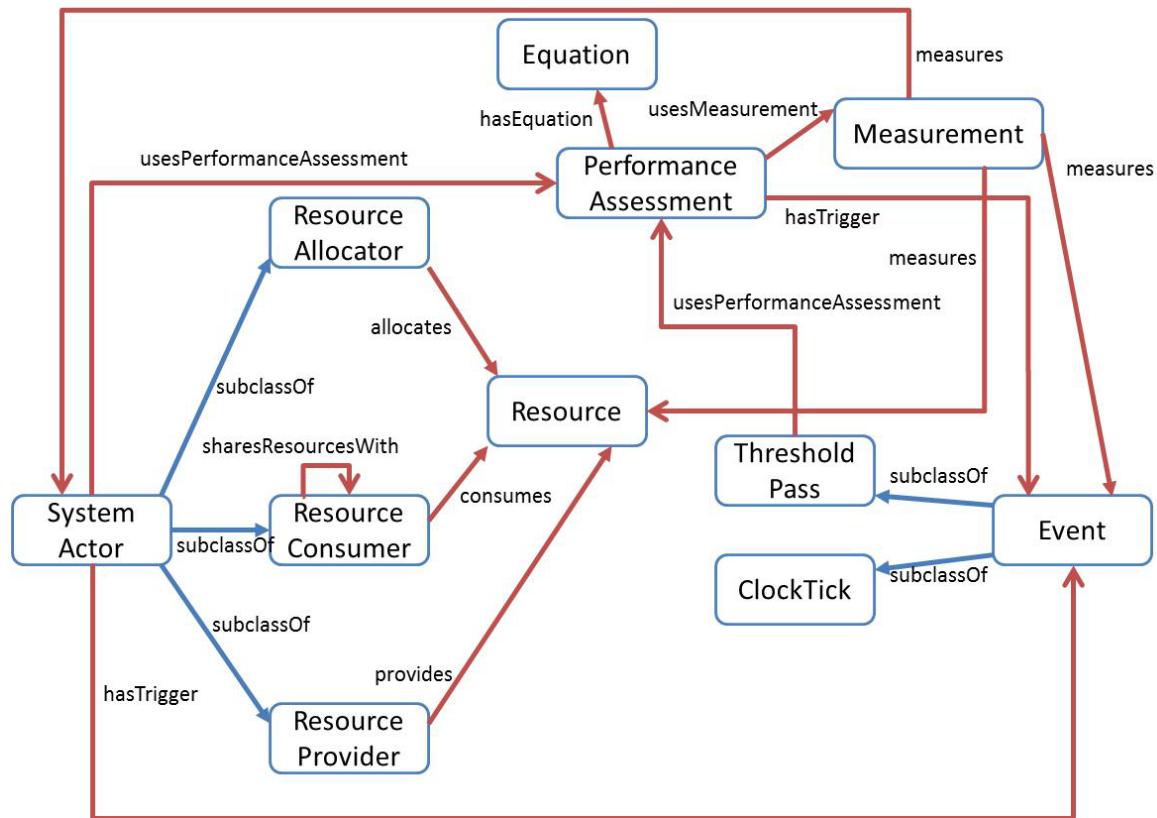
A schematic of the ontology object property hierarchy can be seen in Figure 7.7-2.



**Figure 7.7-2.** A Tree Representation of the Ontology Object Property Hierarchy.

The identification of the property relationships should be fairly self-explanatory because of the naming conventions we used. As such, we only describe a subset of these properties. Note that the object properties are generally broken into pairs to represent inverse relationships. For example, *consumes* is used to identify which resources a consumer consumes, while *isConsumedBy* is used to identify which consumers consume a resource.

A high-level sketch of object property relationship between entities can be seen in Figure 7.7-3. To simplify Figure 7.7-3 and make it more informative and less confusing, we do not show inverse properties whose use can be easily inferred from naming conventions. For example, *allocates* is an inverse property of *isAllocatedBy*. We also do not show some subclass entities whose specialized property usage can be easily inferred from naming conventions and the properties of superclasses. For example, *exclusivelyConsumes* is a subProperty of *consumes* which is used with ExclusiveResource objects instead of normal Resource objects.



**Figure 7.7-3. Class and Object Property Relationships.**

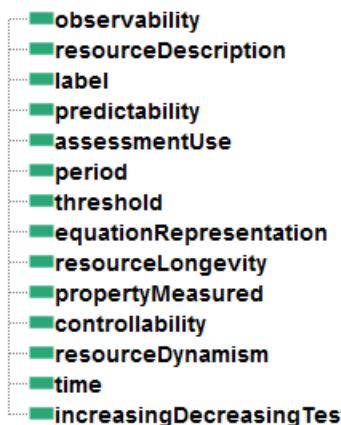
We do not have many subProperty relationships in the shared resource object property hierarchy. Most of these object properties are used to identify resource consumptions and availabilities with special exclusivity and observability properties. For example, a monitor connected to a video card frequently exclusively consumes the information flowing from the card. Similarly, in a ground vehicle such as a HMMWV, the engine control unit has the exclusive ability to observe atmospheric pressure in order to regulate the air-fuel mixture into the engine.

We defined the *sharesResourceWith* property and its subproperties to identify resource consumers that share resources. This sharing may be implicit, explicit or coordinated, so we defined subclasses to capture these scenarios.

We defined *hasTrigger* properties to identify events which trigger when events are allocated or consumed, or when performance is assessed.

### 7.7.3.3 Shared Resource Ontology Data Properties

A schematic of the ontology data property hierarchy can be seen in Figure 7.7-4.



**Figure 7.7-4. A Tree Representation of the Ontology Data Property Hierarchy**

The identification of the property relationships should be fairly self-explanatory because of the naming conventions we used. Most of the data properties represent either times, or they represent strings to describe objects. The time properties include *time* and *period* which respectively represent a time when an event occurs and the amount of time between events. The description string properties include *resourceDescription*, *resourceDynamism* and most others.

#### 7.7.4 Extending the Shared Resource Ontology for Contention Complexity

As a demonstration of the extensibility of our shared resource ontology, we now discuss an extension of this ontology for an ontology we call the *contention complexity ontology*. We gave this ontology the following URI:

<http://meta.bbn.com/ont/2011/04/contention-complexity-ont.owl#>

Contention Complexity is a metric that assesses the propensity for a resource to experience contention. We define each resource to have its own resource complexity measure. We codify these and other motivating hypotheses of contention complexity as follows:

1. Entities/subsystems/components in a system use multiple resources. Entities have varying levels of criticality for using specific resources.
2. Contention complexity is a function of the potential for contention due to more requests from entities to use resources than can be accommodated by the limited resources.
3. Contention complexity can be decomposed and expressed for specific resources. Contention complexity is the sum of the contention complexities of resources. The contention complexity of a resource is a function of the potential for contention of that resource from entities to use that resource.
4. Contention complexity of a resource is a function of:

The number of entities that could request that resource. (A resource with more consumers leads to more contention complexity.)

The level of usage required for use of the resource. This is measured in terms of % usage level \* amount of time per usage. (Higher usage level means more contention, more time per usage means longer queues and more contention.) The higher this product is, the higher likelihood of contention.

More critical uses of limited resources imply more contention complexity.

This metric is expressed as the following equation:

$$\text{ContentionComplexity}(r) = \sum_{c \in \text{DependsOn}(r)} \frac{E[\%level(c, r)] \text{var}(\%level(c, r))}{\text{criticality}(c, r)}$$

We extended the shared resource ontology by defining a subclass of PerformanceAssessment called ContentionComplexity. We then defined three subclasses of Measurement: VariancePercentResourceUsage, ExpectedPercentResourceUsage and Criticality. We defined a new object property forResource that associates ContentionComplexity objects with specific resources. Finally, we define a single individual, ContentionComplexityEquation that expresses the above equation for contention complexity.

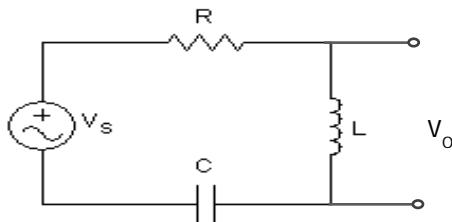
With this very small set of changes, we were able to adapt our general shared resource ontology to a much more specific and still useful ontology for a specific metric that can be further refined.

### 7.7.5 Example Applications of the Shared Resource Ontology Extended to Contention Complexity

We now explore the use and usefulness of the extended Contention Complexity ontology with two specific examples: a toy RLC circuit problem and a hybrid battery problem.

#### 7.7.5.1 RLC Toy Problem

We now consider the application of our contention complexity ontology to the RLC circuit in Figure 7.7-5.



**Figure 7.7-5. RLC circuit**

In this circuit, the voltage source ( $V_s$ ) is a resource provider and provides a resource of electrical energy. The RLC circuit (without the power source) is both a resource consumer and an allocator. The RLC circuit consumes some energy and determines how much energy at each frequency to transfer to any consumer connected at the  $V_o$  output port. Anything that connects to the output port  $V_o$  is both a resource consumer and an allocator because it can influence how the power output of  $V_s$  is allocated across the frequency domain and hence consumed by the RLC circuit. To follow normal electrical engineering convention, associated with  $V_s$  is a time-varying current  $I_s(s)$  that represents the current flowing from the voltage source to the resistor. When the  $V_o$  output port is not open, the current that flows through the output port from the resistor to the capacitor, parallel to the inductor is denoted as  $I_o(s)$ .

We designed an application-specific ontology for this example that imports the contention complexity ontology. We call this application-specific ontology the ***RLC-contention-complexity ontology*** and gave it the following URI:

<http://meta.bbn.com/ont/2011/04/RLC-contention-complexity-ont.owl#>

The amount of resource provided at a given frequency  $s$  is a function of the voltage  $V_s(s)$  and the current through the voltage source  $I_s(s)$ . This provided electrical energy is allocated by the system designer to the RLC components and an output port which provides a voltage  $V_o(s)$  and a variable current  $I_o(s)$  that is a function of the systems connected to the output port. We assume that the voltage source has some max power output  $W_{max}$ . The power output at a given frequency is  $V_o(s)*I_o(s)$  and the integral of this product over the frequencies from 0 to infinity is the total power output.

With this analysis, we designed the application-specific RLC ontology by adding just a few individuals to the imported contention complexity ontology. We added:

- The SourcePowerContention individual which is of type ContentionComplexity. This individual uses measurements of the source voltages and currents, the output voltages and currents, and the RLC circuit parameters to compute contention complexity.
- The SourcePowerOutput individual is of type Resource. It is the resource consumed by the RLC circuit and any other consumer attached to the  $V_o$  port.
- The VoltageSource individual is of type Resource Provider. It provides the SourcePowerOutput resource.
- The RLCCircuit and RLCOutput individuals which are of types ResourceAllocator and ResourceConsumer. The RLCCircuit individual has associated data on the RLC values.
- The  $V_s$ ,  $I_s$ ,  $V_o$ ,  $I_o$  measurements which are used to computed the resource consumptions.

Based on parameterizations of RLCCircuit,  $V_s$ , RLCOutput,  $V_o$ , or  $I_o$ , the RLC-Contention-Complexity ontology can be used to automate the assessment of the contention complexity in the toy circuit.

#### 7.7.5.2 Preliminary Ontology for Hybrid Vehicle Battery Use and Maintenance Analysis

Similar to our extension of the Contention Complexity ontology to the RLC application domain, we can apply similar techniques to more complicated cyber-physical systems such as an example of the complexity introduced into the hybrid vehicle lifecycle due to the contention for resources needed to maintain a particular choice of battery. For this problem, the more often a battery needs to be replaced, the more resources it will use (e.g., manpower and money) that could be used to maintain other systems. We want to use this metric to select a battery and battery configuration (i.e., the depth of discharge used in the hybrid vehicle's charge/discharge control algorithm) to minimize the maintenance resources needed by the battery over a vehicle's lifecycle for a variety of vehicles based on parameterizations of vehicle weight, battery capacity, the battery charging control, and the vehicle use patterns.

Note that this example is simplified. It does not include other factors that would be involved in the selection of a battery, such as the battery's weight, power density, disposal costs, and safety.

To compute the metric for this example, we need an expected value for the level of demand for the resource (which in this case is the cost to replace the battery) and the variability of that demand. For this example, the depth of discharge has an effect on the demand for the resource. A battery has to be replaced (i.e., demands the resource) after a certain number of charging cycles. A deeper discharge minimizes the number of charging cycles but a shallower discharge enables more cycles before the battery needs to be replaced.

For this scenario, we define two individuals which are of type Resource: PowerStoredInBattery and MaintenanceResources. These individuals respectively capture the ability of the resource to recharge over real or simulated terrains and the requirements for resources for battery maintenance.

There are two individuals of type ResourceProvider that provide the PowerStoredInBattery resources: EnginePower, Battery and RegenerativeBraking. Because maintenance resources are part of the nominal system infrastructure and must be provided for vehicle use, we consider maintenance to be provided finite and provided by the system infrastructure for the purposes of our analysis. Larger analysis might include multiple vehicles of various types to analyze maintenance requirements.

There are two individuals of ResourceConsumer type: Battery and VehicleDrive. The battery has multiple properties associated with it including the cycle capacity for a given depth of discharge which is represented as a PerformanceAssessment individual. Each battery is associated with BatteryType and PowerCapacity properties. Vehicle drive is parameterized with performance requirements.

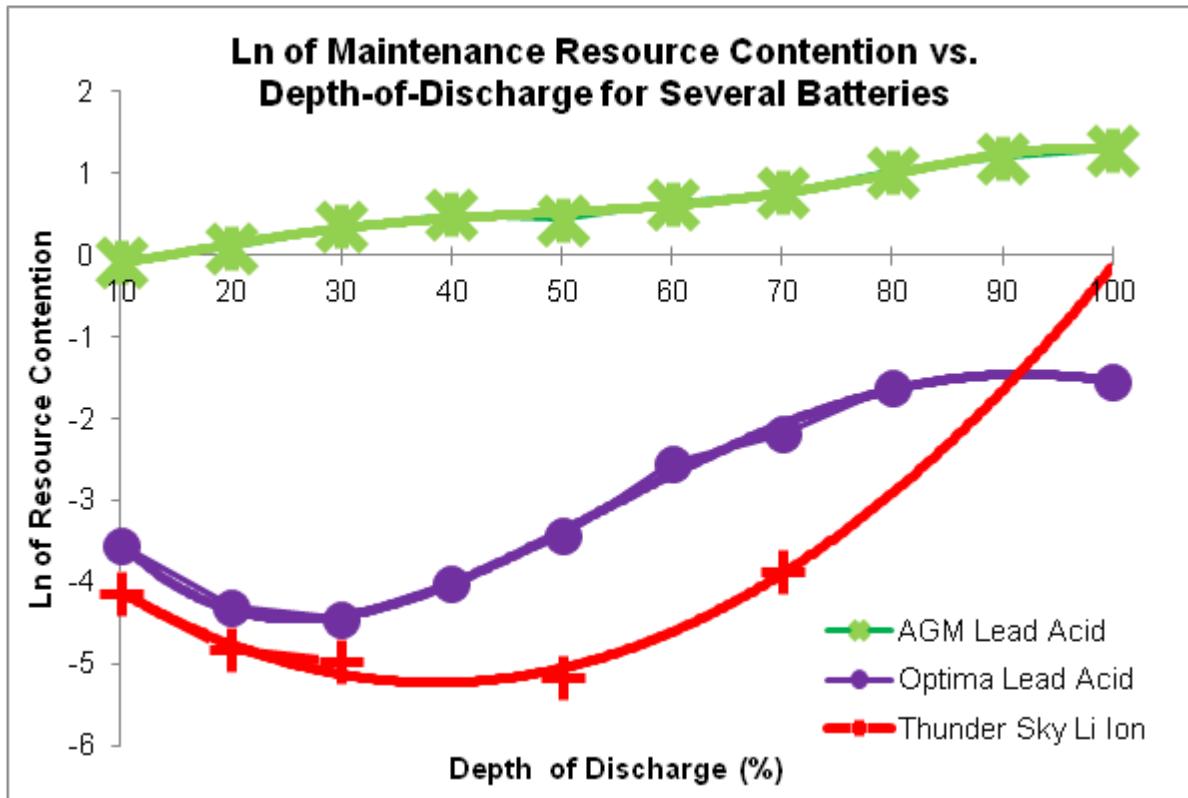
The BatteryController individual is of type ResourceAllocator and allocates power for either the discharging or recharging of the battery.

They are multiple events that are used to control the allocation of resources based on whether the engine turns on or off, whether the vehicle is drawing power from the battery, and whether the vehicle needs motive power or not. These measurements are used by the battery charger to decide when to charge the battery. The control operation is ultimately measured by a Measurement individual for the battery: the number of recharging cycles experienced by the battery.

The battery control algorithm is based on a straightforward Energy Transfer Model. Basically, going up a hill at a particular speed uses a certain amount of energy (potentially provided by discharging the battery) and going down a hill transforms potential energy into kinetic energy that can be regeneratively stored in the battery as electro-chemical energy. We designed our simple control algorithm so that the battery will be used for motive power until it is fully discharged (to its prescribed depth of discharge), then it would charge until it is fully charged – either by the engine or regenerative charging system.

Our intention is that this ontology we are sketching for hybrid vehicle battery maintenance is used to analyze the maintenance requirements for individual vehicles or fleets of vehicles. We see it being used to compute the lifecycle replacement cost for batteries' replacement. We used an early version of this ontology to run simulations for multiple batteries at 10 different depth of discharge levels (10% through 100%), with each simulation run covering 10,000 hours of vehicle operation, and kept track of how many charging cycles occurred over the 10,000 hours

of operation. The graph in Figure 7.7-6 shows preliminary results of the computed metric graphed for each battery and depth of discharge on a logarithmic scale.



**Figure 7.7-6. Chart Showing Contention Complexity for Multiple Battery Types and Depths of Discharge**

We should be careful to note that as currently designed, assessments using the ontology will always admit some error because, for reasons of tractability and abstraction to simplify analysis, the ontology does not include all the factors that should go into the battery choice. For example, our approach does not account for safety concerns. (Li-ion batteries are known to be explosive when engulfed in flames.) But in this simplified example, the computation of the metric would show that the choice of the Lithium Ion battery with a depth of discharge between 30% and 50% provides the minimal resource contention complexity (informally, the minimal maintenance cost over the battery's lifetime). Interestingly to us is that these preliminary results align well with real-world results that we discovered from automakers after performing these experiments.

### 7.7.6 Prototype Toolchain for Metric Evaluation

We now describe our General Adaptable Metric Execution Tool and Environment (GAMETE) to host and evaluate metrics. We developed GAMETE during META Phase 1 to evaluate general classes of metrics as part of the design and Verification and Validation (V&V) of complex engineered systems. GAMETE identifies system designs that are less complex, more efficient, less likely to fail, less costly, and that have higher performance, and evaluates the behavior of designs during experimentation or simulation for V&V. We have demonstrated GAMETE with complexity metrics of our own design and from several general classes of

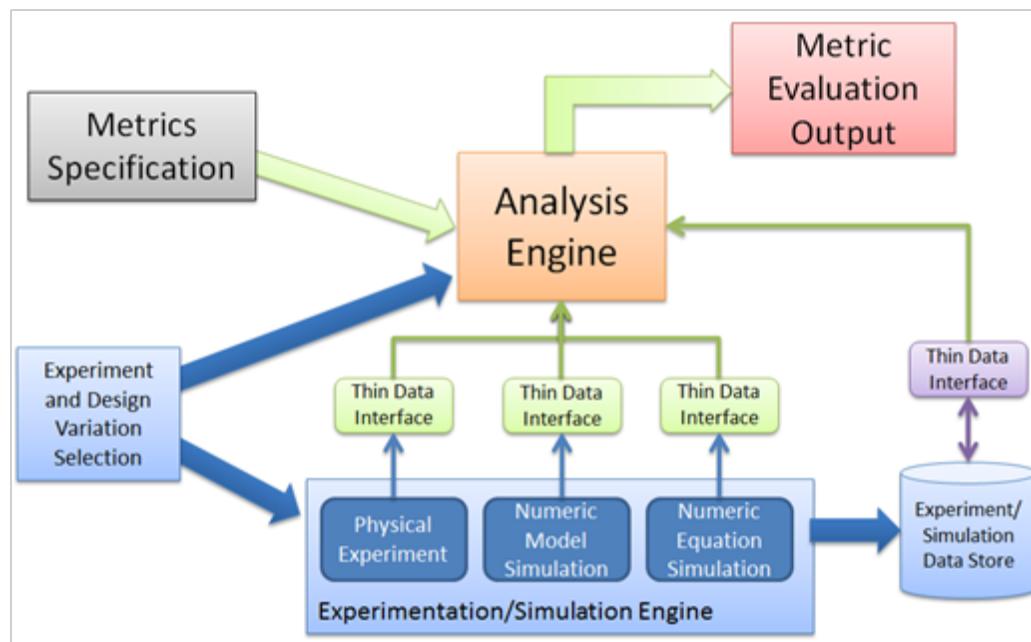
metrics provided to us by other META and META II performers. By design, GAMETE is easily integrated with larger design toolchains. We have already integrated GAMETE into the BAE Systems ARRoW toolchain as part of META. The benefits of GAMETE include:

- Great increases in the scale and breadth of metrics and experimental data sets supported through the GAMETE evaluation infrastructure.
- Improved interactive capabilities of GAMETE by supporting automated online data ingest and online metric evaluation as experiments are run.
- Aiding the identification of primary and secondary impacts of design alternatives by automating the evaluation of metrics over design alternatives.

#### 7.7.6.1 GAMETE Executable Framework

A diagram of the GAMETE architecture and design tool interface is shown in Figure 7.7-7.

GAMETE enables the META design and V&V toolchain vision by supporting the evaluation of general classes of user-selected metrics on user-selected design variations and user-selected data. GAMETE is highly-extensible and adaptable to many larger tool chains because its metrics, coverage, and experiments are specified as pluggable components so they can be added and managed orthogonally to metric evaluation and analysis. The evaluations are output to the user or other consumers in external toolchains as part of either design or V&V activities.



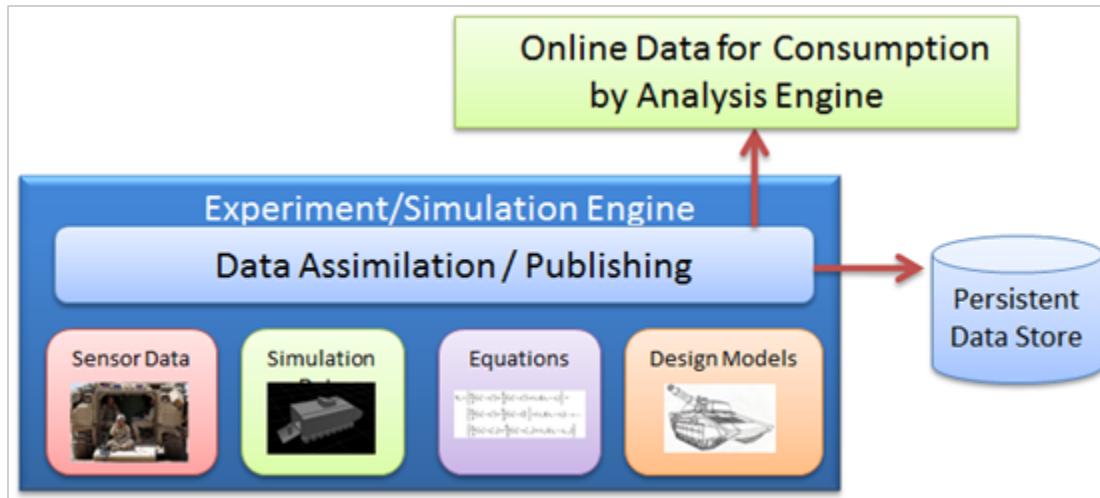
**Figure 7.7-7. GAMETE Architecture Design and Toolchain Interface**

Key benefits of the GAMETE approach not seen in other solutions for metric evaluation include (1) our *unified data representation* design to enable the evaluation of metrics over large and diverse classes of data, and (2) support in our analysis engine to evaluate similarly large and diverse classes of metrics. Although GAMETE is a prototype, we have demonstrated the unified data representation and metrics library capabilities on a variety of data from the META

and META-II programs and have integrated GAMETE with the BAE Systems ARRoW toolchain.

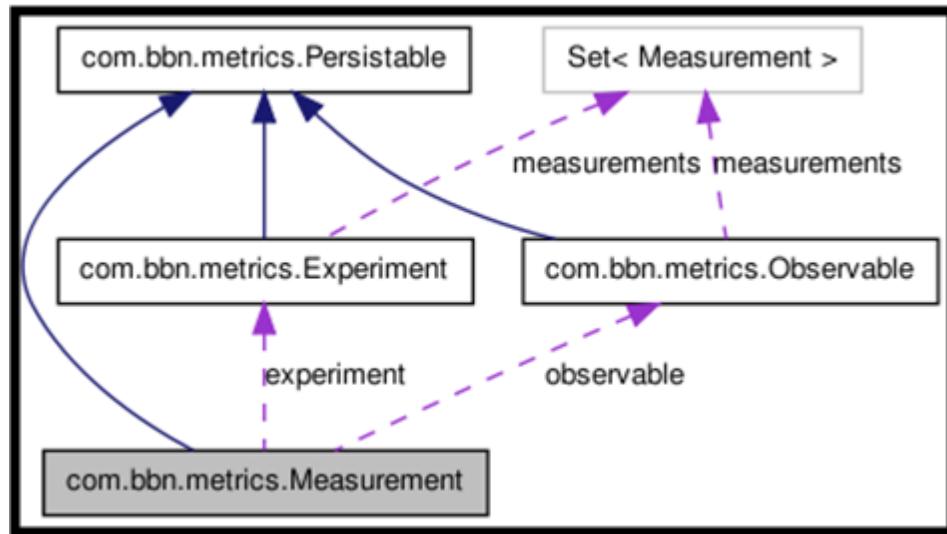
### 7.7.6.2 Unified Data Representation

A central part of GAMETE capabilities and extensibility derives from the only assumption we make on experimentation/simulation engines – that they are generic processes that can commit data directly to a consumer such as GAMETE or directly to a consumer after persisting the data in a datastore. An example experimentation/simulation engine is shown in Figure 7.7-8 that hosts these data sources. The experimentation/simulation engines supported by GAMETE include engineered systems that generate data directly from (i) *sensors* such as those that might monitor aspects of system behavior during testing or that gather information during design, (ii) *simulations* that might generate data over multiple runs, (iii) deterministic evaluations of possibly coupled *equations*, and (iv) model representations, such as graph models or equations. GAMETE is designed to semi-automate the collection of data as needed from all of these either online or offline data sources.



**Figure 7.7-8. The Experiment/Simulation Engine feeds data from diverse sources for online or offline processing.**

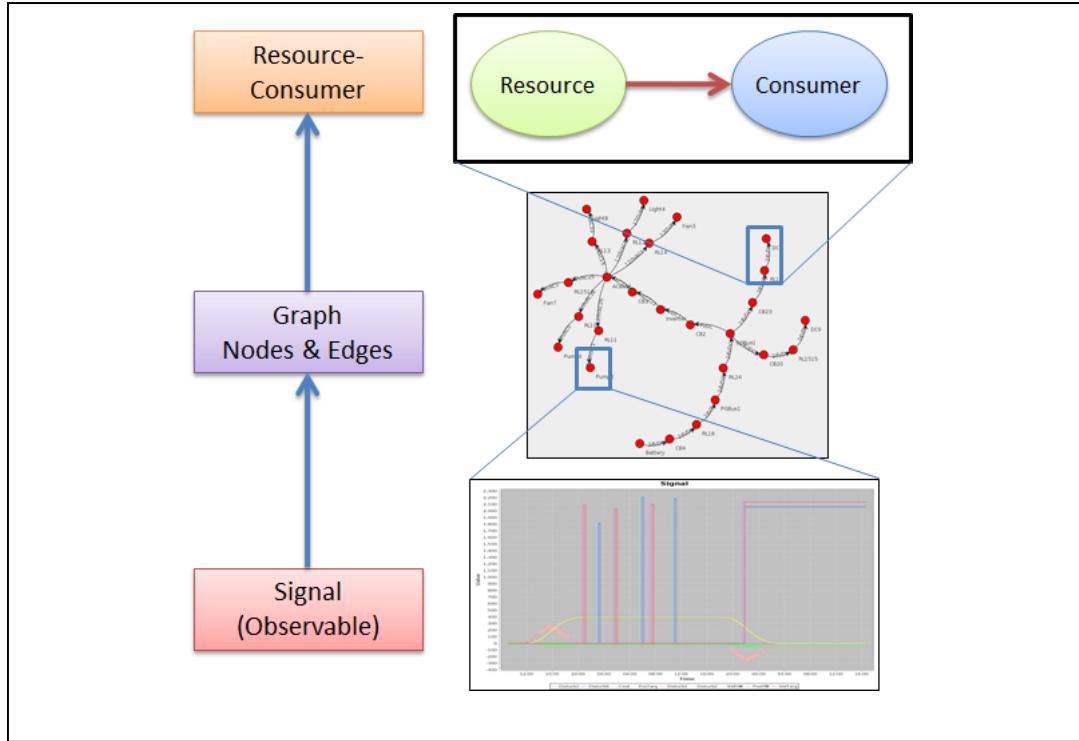
The distinguishing factor of our GAMETE approach is our Unified Data Representation (UDR). The UDR models the reporting/storing of experimental/simulation and model data that enables the pluggable metrics and experiments to be developed in the execution framework. Recognizing that experimental data is collected from a number of different methods, using different tools with varying frameworks and languages, the UDR provides an easily-supported framework to interface the analysis engine with experiments and simulations over which metrics are evaluated. Refer to Figure 7.7-9.



**Figure 7.7-9. Measurement storage format in the Unified Data Representation.**

The UDR is developed around a measurement as sketched in Figure 7.7-10 which shows the recording of a value (e.g., 9 volts) of some *Observable* component (e.g., a battery) at a given time. Solid blue arrows represent inheritance relations, dotted purple lines show composition.

Figure 7.7-10 shows the usefulness of the UDR for resource-consumer problems as graphs, and graph nodes/edges as signals, thus maximizing the applicability of implemented metrics. The key insight in the UDR is that the data types that need to be stored for the META metrics application domains are hierarchically ordered. This insight allows metrics to be applied in interesting new ways such as representing resource-consumer relationships in graphs and applying graph-based behavioral metrics to analyze the complexity of resource-consumer relationships.



**Figure 7.7-10. Representing signals, graphs, and resource-consumption in the UDR maximizes the applicability of implemented metrics.**

### 7.7.6.3 Metrics Supported by GAMETE

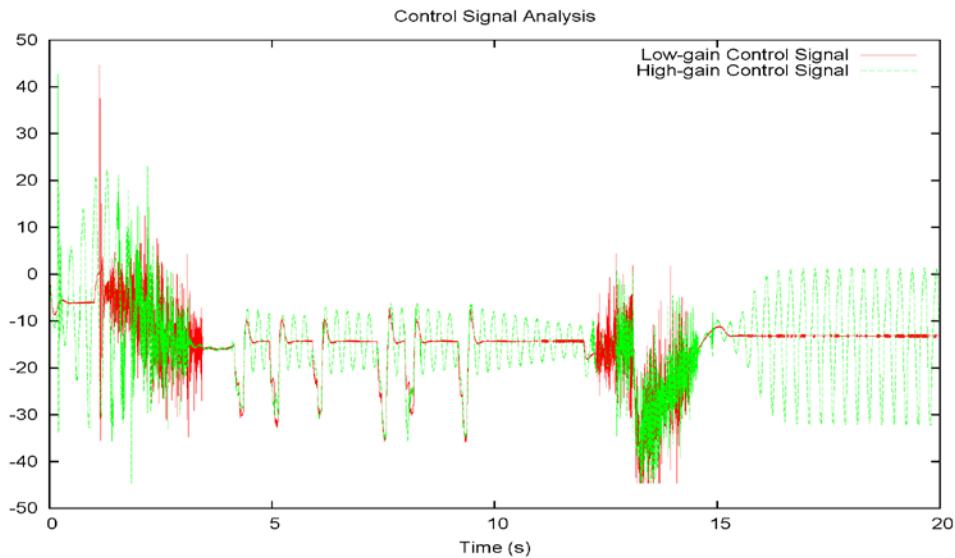
We have successfully demonstrated GAMETE with metric libraries as part of META Phase 1. We demonstrated GAMETE with complexity metrics of our own design and from several general classes of metrics provided to us by other META and META II performers.

An example metric we defined and implemented with GAMETE is *resource contention complexity*. This metric assesses the propensity of processes to become resource starved and focuses on the interaction of dynamic components in complex engineered systems over time. The intuition is that vehicle designs in which there is greater propensity for resource contention are more complex and costly over their lifecycle than designs in which there is less propensity for contention. Few designs capture all the potential for resource contention, which can include obvious contention for resources like power or processor time and less obvious ones, such as heat dissipation, maintenance resources, and physical space. We developed simulation models in Python of battery maintenance in hybrid powertrain military vehicles and used GAMETE to evaluate resource contention complexity over our custom simulation environment.

We implemented metrics designed by other META and META II performers, including signal complexity metrics from the BAE META team and behavioral metrics from the PARC META 2 team. We evaluated these metrics over Simulink and graph model outputs, respectively, of the ramp example developed by BAE for META to assess the complexity of several designs.

#### 7.7.6.4 Metric Evaluation Demonstration

As a demonstration example for using GAMETE to evaluate signal complexity, we considered the control signal output by two different control designs for the BAE ramp model. The control signal in the ramp model controls the amount of torque output by the ramp motor. One control design corresponds to a low-gain controllers and another control design corresponds to a high-gain controller. Figure 7.7.11 shows the two control signals from these two different controllers. The low-gain controllers has a nearly constant output at steady-state while the high-gain controller has a periodic output even at steady state because its high-gain control signal causes the controlled system to overshoot its desired operating point before the system can converge. We used the two control signals in Figure 7.7-11 as input to GAMETE and we evaluated the signal complexity metric for these two signals. The control signal from the low-gain controller has a signal complexity 40% lower than the control signal from the high gain controller. This makes intuitive sense because the high gain controller generates a periodic actuator input signal rather than constant steady-state signal which requires lower-cost control circuitry and maintenance.



**Figure 7.7-11. Two control signals, one from a high-gain controller with periodic steady-state behavior and one from a low-gain controller with DC steady-state behavior.**

The hierarchical relationship we impose through the UDR of resource-consumers, graphs, and signals allows for metrics to be applied in GAMETE in useful and interesting new ways. For example, since resource-consumer relationships are represented in graphs, the graph-based behavioral metrics can be computed to analyze the complexity of resource-consumer relationships. Furthermore, since the graph nodes and edges represent signals, signal metrics can be applied to each edge and node of the corresponding graph. This capability is partially why GAMETE is a best-of-breed technology that has the effectiveness, general applicability and promise for continued cost-effective improvement in the META Maturation Phase.

### 7.7.6.5 Integration of GAMETE with Toolchains

Besides the UDR, the GAMETE architecture uses thin data interfaces between the GAMETE analysis engine and the data sources, which makes GAMETE easily integrated with larger design frameworks and toolchains. As part of our META activity with the BAE META team, we designed the thin data interfaces to easily “plug-and-play” new data sources into GAMETE so that the new data can be ingested through the UDR. Besides the ARRoW activity, we have also ingested PARC META 2 electrical system data. Our testing, demonstration, and evaluations have shown that GAMETE has promise for being an important part of a vehicle and CPS design toolchain, further investment to mature and integrate the prototype system would enable this promise to become a reality. In the next section, we describe some areas for focus under the META Maturation Phase that we propose to do.

### 7.7.7 Conclusion

Taken together, our metrics definition, component language and GAMETE metric evaluation framework work provide a key aspect of an end-to-end design tool chain. The metrics definition identify aspects used for the verification and analysis of complex military system design. The component language is used to design systems from sub-systems and propagate the complexity analyses. The GAMETE evaluation framework makes these advances real and usable by design engineers.

By design, GAMETE is easily integrated with larger design toolchains. We have already integrated and incorporated GAMETE functionality into an integrated META tool-chain, providing complexity management features necessary for AVM design goals. We will expand upon this activity during the maturation phase to mature GAMETE and integrate with other providers’ toolchains.

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.8 – BBN Spatial DSE**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.8 Spatial Design Space Exploration.....</b>	<b>1</b>
7.8.1 Motivation: Design Space Exploration .....	1
7.8.2 META Accomplishments: Lightweight Agent Simulation Specifications (LASS) ..	3
7.8.3 Proto/Unity Integration .....	9
7.8.4 Deliverables .....	10
7.8.5 Bibliography .....	10

## List of Figures

Figure 7.8-1. By using LASS in design space exploration, potential environment-specific human-vehicle interaction issues can be addressed earlier in the design process.....	2
Figure 7.8-2. Proto uses the amorphous medium abstraction, in which a discrete network (right) is viewed as an approximation of a continuous space (left), allowing simpler distributed system programming, greater robustness, and increased scalability.....	4
Figure 7.8-3. Example use-case scenario for LASS: egress through an IFV rear ramp/gate (right) similar to that of a Bradley IFV (left).....	6
Figure 7.8-4. Example of design change evaluation in LASS: as the passenger bay changes size (down to 4 passengers on right, up to 16 passengers on left), the number of warfighter agents needing to egress can automatically adjust, along with the agent program controlling them ...	7
Figure 7.8-5. Example of design change evaluation in LASS: a more complex door structure (right) intended to provide better cover to egressing warfighters than the standard door structure (left). Agent-based simulation can provide a rough quantification of the benefit in safe deployment. ....	8
Figure 7.8-6. LASS combines the realistic physics and interactive terrain modeling of game engines with Proto's robust and scalable agent control in a rich simulation environment.....	9

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
API	Application Programming Interface
DSE	Design Space Exploration
IFV	Infantry Fighting Vehicle
LASS	Lightweight Agent Simulation Specifications
MRAP	Mine Resistant Ambush Protected
ODE	Ordinary Differential Equations

## 7.8 Spatial Design Space Exploration

During META, we have produced a prototype of Lightweight Agent Simulation Specifications (LASS), a design-space exploration tool that radically reduces the cost of incorporating capability requirements into the early design process. The capability requirements for every vehicle include its interactions with humans in its environment (e.g., deployment of passengers under fire, coordinated operations with dismounted infantry, navigation of populated areas). These can significantly influence the lifecycle viability, maintenance, and cost of a system, because users can interact with a system in a wide range of unanticipated ways (such as soldiers exiting a vehicle before its ramp is fully deployed), but these have traditionally been extremely hard to evaluate in early design phases, before a physical prototype exists, due to lack of cost-effective simulation tools to accurately model the large range and unpredictability of human interactions.

LASS fills this necessary gap in cyber-physical and vehicle system design using two state-of-the-art technologies. *Spatial computing* allows the large space of potential human interactions that can affect a vehicle’s design and operation to be programmed as a continuous space and compiled to discrete *software agents* in an agent-based simulation. The result is that LASS will radically reduce the cost of constructing agent-based simulations and including simulations of human-interaction in design-space exploration. These simulations are a key part of a vehicle design tool-chain, making it easier to detect potential problems early and avoid costly redesigns in the later stages of design and deployment. LASS combines our unique and open-source Proto spatial computing language for specifying scalable and adaptive agent-based simulations, with commercially available physics simulation, game-based simulation, and scenario authoring tools. Because Proto simulation specifications are scalable and adaptive, a capability test scenario can evaluate a wide range of variable vehicle designs without any further intervention from a simulation engineer.

Our key achievements during META are:

- Development of the LASS concept and use case scenarios.
- Proof-of-concept demonstration of importing a vehicle design specified in terms of rigid bodies and joints, and use of differently scaled versions of this design in a use case scenario.
- Development of a prototype LASS system that connects Proto with the Unity simulation engine, and uses the combination to create proof-of-concept scalable and adaptive capability test scenarios.

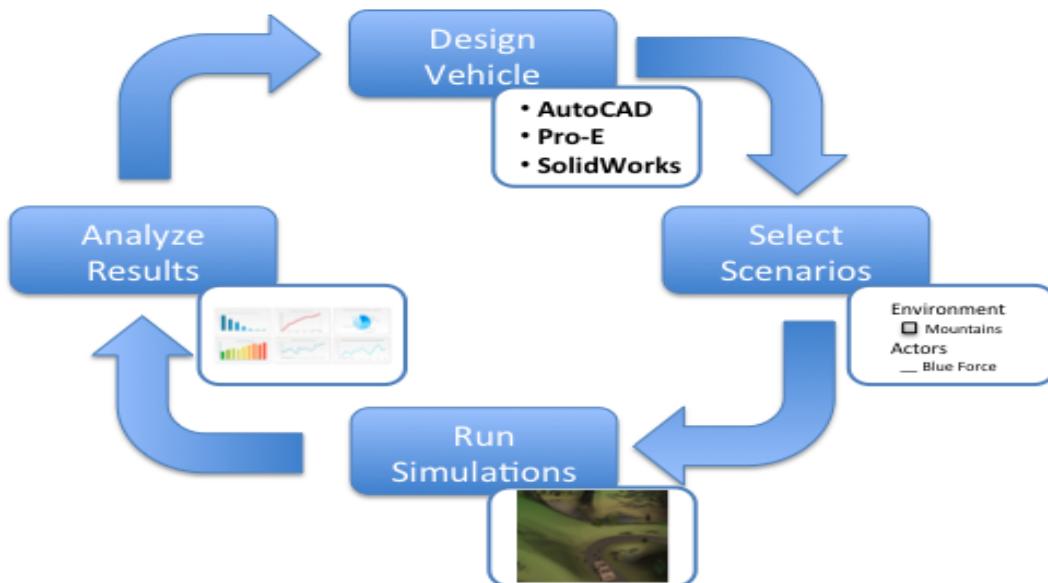
### 7.8.1 Motivation: Design Space Exploration

In response to a 2011 congressional request to evaluate the challenges confronting the fielding of ground combat vehicles [KLS11], The RAND Corporation stated, “Current efforts to keep up with rapidly changing requirements on the battlefield are struggling.” Without a method for incorporating battlefield requirement exploration into the design process, identification of potential issues is often left to soldiers in the field, resulting in a lengthy and costly redesign process.

Design changes become radically more expensive in later stages of development. Coarse-grained simulation of emerging operational scenarios is vital for achieving fast and effective design space exploration, allowing design options to be evaluated against an array of potential scenarios early in the design process. While it is now becoming common practice to include

environmental model simulations in the design cycle (e.g., various weather conditions and terrain types), these models still ignore the human-vehicle interaction that is a critical component of any successful combat vehicle. One key reason for this failure to include human-vehicle interaction is the prohibitive cost and difficulty of creating and maintaining scenarios with conventional agent-based simulation authoring tools.

**Any effective vehicle design process needs to incorporate lightweight agent simulations**, which allow human-vehicle interaction scenarios to be created inexpensively and then used to evaluate a wide range of vehicle design options without any need for human adjustment of the simulation. An evolving vehicle design can then be evaluated against the operational constraints of soldiers and civilians, identifying both potentially fatal issues and possible design tradeoffs long before the first vehicle reaches the battlefield. If these simulations are lightweight, in the sense that they are simple to construct and can self-adapt to many types of design changes, then they can be used early and throughout the process (refer to Figure 7.8-1), enabling designers to maintain a clear connection, throughout the design process, between their design decisions and the required field capabilities.



**Figure 7.8-1. By using LASS in design space exploration, potential environment-specific human-vehicle interaction issues can be addressed earlier in the design process.**

The addition of lightweight agent-based design-space exploration to a design tool-chain can:

- **Improve** the overall design process through iterative design-evaluate cycles.
- **Reduce** costly downstream changes by identifying potential human-vehicle interaction issues earlier in the design life cycle.
- **Streamline** tool-chain evaluation through automatic adaptation of interaction scenarios and parameters.

### 7.8.2 META Accomplishments: Lightweight Agent Simulation Specifications (LASS)

We view the problem of human-vehicle interactions through the lens of agent-based simulation. A potentially complex simulation can be decomposed into a network of interacting agents where each simulated human in the vehicle’s environment may be represented by an agent, which attempts to act in accordance with its orders and own interests.

There are three key ideas in our approach, which we call Lightweight Agent Simulation Specifications (LASS):

- **Refine capability requirements instead of setting physical requirements.** Historically, high-level capability requirements (e.g., “deploy quickly and safely under fire”) are refined into detailed physical requirements (e.g., ramp/gate drive specifications). These physical specifications may conflict with the design of alternate solutions. Under the LASS approach, wherever possible, capability requirements are refined to simulations with metrics independent of the physical realization of the design. This provides much more flexibility in how a design is realized, as well as the possibility of a holistic evaluation of design decisions.
- **Integration of agent-based simulations within the design framework.** Manually evaluating how agents interact within the physical design is likely to be infeasible for the human design team. By integrating the construction of capability simulations with the vehicle design process, the system will be able to provide meaningful capability feedback during each iteration of the design loop.
- **Spatial computing creates simple, adaptable simulations of agent interactions.** When a simulation is viewed as a network of agents, many interactions happen between agents that are physically near one another. For example, disembarking soldiers may physically get in one another’s way, and within a vehicle heat and vibration spread through local interactions. We may thus view such a network of interacting agents as a spatial computer. Our spatial computing technology allows agents to be programmed as a scalable aggregate, rather than as a collection of individuals, allowing simulations to be specified simply and to automatically adapt to many classes of design change.
- The LASS approach exploits these ideas to extend simulation-in-the-loop design to incorporate high-level capability requirements for the interaction of a design with humans in its environment. Fast, lightweight simulation will allow enhanced design exploration. For a human designer or automated design tools, LASS exposes trade-offs that are otherwise difficult or impossible to detect due to information lost when capability requirements are reduced to physical design constraints. At a higher level of decision-making, project managers could use LASS to evaluate the sensitivity of a design to proposed changes in capability requirements.

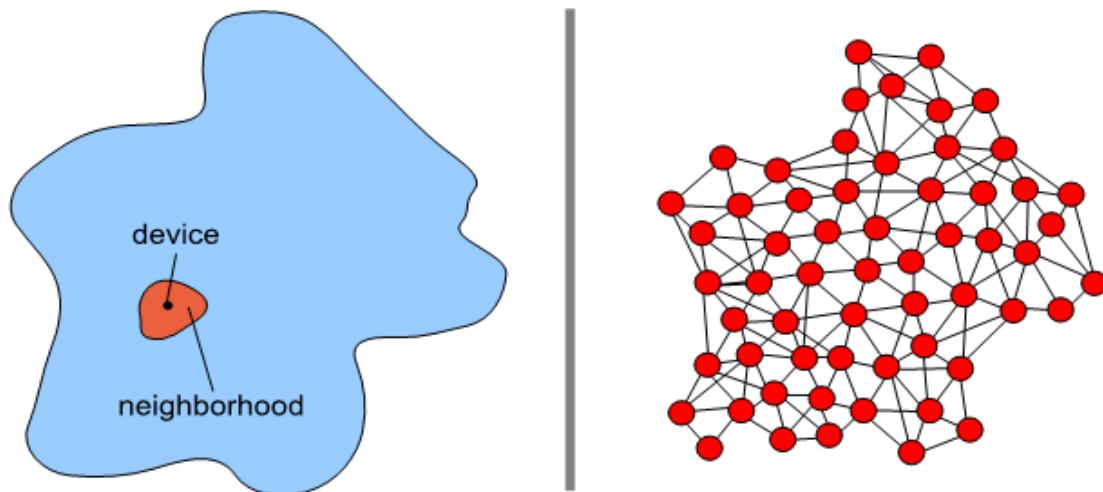
#### 7.8.2.1 Spatial Computing enables LASS

Our LASS approach is founded on spatial computing, and is enabled by the Proto spatial computing language and simulator. In this section, we present a brief introduction to spatial computing and Proto; for a more detailed introduction, see [BB06, BBM10].

When a simulation is viewed as a network of agents, it is often the case that most interactions happen only between nearby agents. We may thus view such a network of interacting agents as a spatial computer—a collection of devices embedded in a (usually physical) space such that interactions between devices are mostly local in space [BS10, DGG07].

Viewing the system in such a way allows us to apply the scalable aggregate programming techniques that have been developed in the Proto spatial computing language [BB06, BBM10]. The key insight enabling Proto's continuous spatial approach is the recognition that there are many systems where the focus is best placed not on the devices that make up the system, but rather on the space through which the devices are distributed. Sensor networks are a prototypical example: e.g., the point of a target-tracking network is to monitor the movement of entities through an area. The fact that this involves observations made by and at particular devices is only of interest so far as it contributes toward that goal. Multi-robot systems and ad-hoc mobile networking are good examples as well, e.g., the point of a robot coverage algorithm is to examine all points in a space of interest, and the point of an ad-hoc routing algorithm is to move information across space to where it is needed. These latter systems are closely related to agent-based simulation of interaction between a vehicle and humans in its environment.

If devices interact primarily over short distances, then the aggregate structure of the agent interaction network forms a discrete approximation of the structure of the space of interest. Combining these two observations allows us to view the network using an abstraction that we call the amorphous medium [JB04]. An amorphous medium is a Riemannian manifold<sup>1</sup> with a computational device at every point, where every device knows the recent past state of all other devices in a local neighborhood (Figure 7.8-2). A network of locally communicating devices can thus be viewed as a discrete approximation of an amorphous medium, with each device representing a small region of nearby space and messages sent between nearby devices implementing the information flow through neighborhoods.



**Figure 7.8-2.** Proto uses the amorphous medium abstraction, in which a discrete network (right) is viewed as an approximation of a continuous space (left), allowing simpler distributed system programming, greater robustness, and increased scalability.

---

<sup>1</sup> A manifold is a mathematical object that looks like Euclidean space locally, but globally may be different. For example, the surface of the Earth is a 2-dimensional manifold: locally it looks flat, but if you keep going in a straight line, you will return to your starting location. A Riemannian manifold also guarantees the availability of other key geometric building blocks such as angle, distance, area and volume, curvature, and gradients (generalized derivatives). For a good introductory text, see [MPC92].

With carefully chosen computational primitives and a means of combining those primitives, such as those provided by Proto [BB06, JL02], it is possible to maintain a tight relationship between an abstract computation specified for a Riemannian manifold and an actual computation being carried out on a real network that is distributed through that space, so that a program written to execute over continuous space can be approximated on real devices.

Applying this approach to capability simulations for design space exploration, we see a number of potential advantages:

- **Proto makes it simple to create agent-based simulations.** The continuous aggregate model used by Proto means that much of the details of constructing distributed software programs, such as interacting agent models in an agent-based simulation, is implicit. This means that complex agent control patterns can often be implemented with only a few simple lines of code.
- **Continuous-abstraction simulations can self-adjust for many design changes.** The amorphous medium abstraction means that Proto programs are typically formulated in terms of geometric operations and information flows through regions of space. When done correctly, this produces extremely flexible and scalable programs: changes in number or distribution of devices are reflected simply as changes in the manifold on which the program is being executed, which implicitly adjusts the result of geometric operations, automatically causing a program to self-adjust for its changed environment. This is discussed in detail in [BS10]. In the design space exploration context, this capability can be used to create simulations that adapt to changes in design without a need for human intervention.
- **Agent-based simulations are generally parallelizable.** An agent-based simulation models the world as a network of independent devices that interact through a well-defined set of messaging or physical interfaces. As a result, it is typically relatively straightforward to parallelize agent-based simulations for faster execution.

We thus see that a spatial computing approach to capability simulation, implemented with Proto, is likely to give us the lightweight agent simulations that we desire for evaluating capability specifications of a vehicle's interaction with humans in its environment.

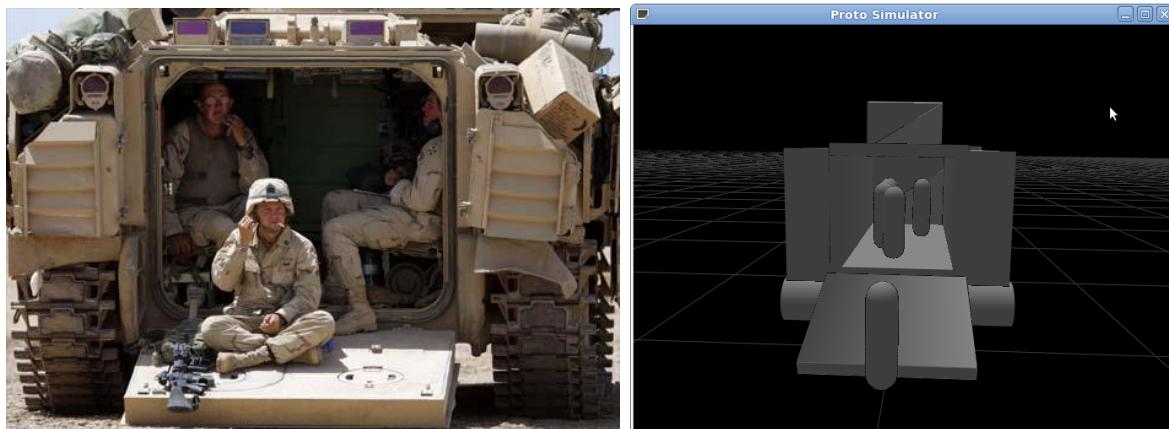
With cheap, self-adaptive, and parallelizable simulation of vehicle capabilities, we are also given the possibility for automated design look-ahead. It is valuable for a designer to know not just how good a particular design variant is, but whether the changes being made are increasing the fragility of the design to further changes. One evaluation method approach is to take the current design and its previous version, then parameterize them to create a linear function for blending from one to the other. The system could then evaluate the design at a number of intermediate points, as well as continuing beyond the current design in an extrapolation of what might happen if the design continues to be varied in the same way. In this way, the sensitivity of a capability to design change might be evaluated, with a warning given if the capability slope near the current design is significantly more negative than the slope near the previous version.

### 7.8.2.2 Illustration of Design Import into Use-Case Scenario

For our initial investigation of the LASS concept, we began by applying it to the design of an Infantry Fighting Vehicle (IFV) rear ramp/gate, with an aim to demonstrate integration of a mechanical design model with a Proto-driven use-case scenario. We chose the ramp/gate scenario, of the three main examples that were then under discussion in the ARRoW project, because it has a capability specification that engages directly with warfighters using the vehicle and other humans in the surrounding environment.

We have focused on egress of passengers through the ramp/gate (Figure 7.8-3). In particular, our mock-up scenario has:

- An IFV with a rear ramp/gate
- A number of warfighters inside waiting to disembark
- Optional enemies scattered around outside, intending to shoot disembarking warfighters.



**Figure 7.8-3. Example use-case scenario for LASS: egress through an IFV rear ramp/gate (right) similar to that of a Bradley IFV (left)**

Each simulation was specified with two components: a scenario and an agent model. The scenario is contained in an XML file that specifies the current IFV design (including the ramp/gate structure), the distribution of warfighters within the IFV, and the distribution of enemies (if any) outside of the IFV. The agent model is a Proto program that specifies a controller for the design and the agents that will interact with it. In this case, the design controller opens the ramp/gate, the warfighters try to protect themselves from enemy fire while dispersing outward, and the enemy fires suppression fire at the area of the IFV ramp/gate.

In this mock-up system, this was implemented by means of a plug-in for the MIT Proto simulator. This plug-in extended the existing free software plug-in for running Ordinary Differential Equations (ODE) simulations in MIT Proto, in order to be able to support scenario scripting, models of combat interactions, virtual sensors needed by the controller, and computation of design performance metrics. The whole system can be invoked with a set of command-line options to the MIT Proto simulator, and runs either in interactive mode for exploration and debugging or in a batch mode for distributed bulk computation of performance and design sensitivity metrics. Figure 7.8-3 shows a screen shot of a ramp/gate design in our mock-up system. Note that this system has since been superseded by the Unity LASS implementation described below.

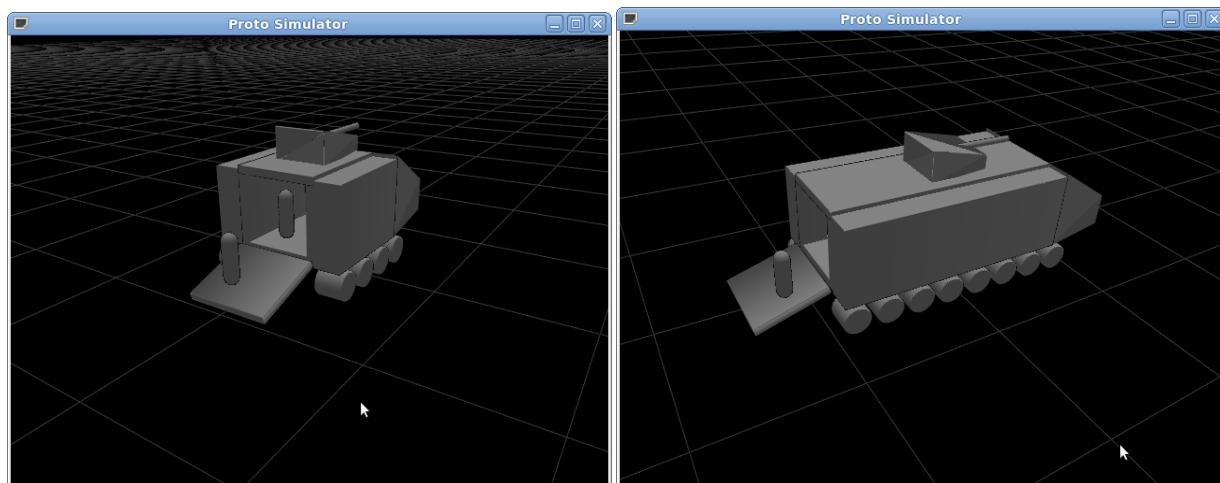
Once constructed, a scenario allows direct (simulated) measurement of the high-level mission capabilities of the design. We are currently considering two metrics in a mock-up egress scenario:

- Number of seconds for all passengers to complete egress
- Number of casualties (when under enemy fire)

Each of these metrics would be measured on many instances of the simulation, so that stochastics of the simulation do not have an undue impact. We currently envision measuring mean and variance of performance over a batch of 20 simulations. Measuring variance can also help in determining when the simulation can meaningfully aid in distinguishing between proposed designs.

Once a LASS simulation has been set up, design variants can be easily evaluated, to within the accuracy of the simulation. In our mock-up system, we have been exploring this concept by developing two examples of design change: changing the size of the IFV passenger bay and changing the design of the ramp/gate.

Changing the size of the IFV passenger bay changes the number of warfighters that can be aboard as passengers (Figure 7.8-4). Such a change might be caused by an overall change of the vehicle size, or by space reallocation between the passenger bay and other adjacent components of the IFV design. Whatever the cause, when the bay changes in size, it changes the number of warfighter agents that need to egress, and therefore may change the time it takes for egress. Changing the size of the ramp/gate may also change its size and effective profile or its opening time, and therefore the vulnerability of warfighters as they deploy.



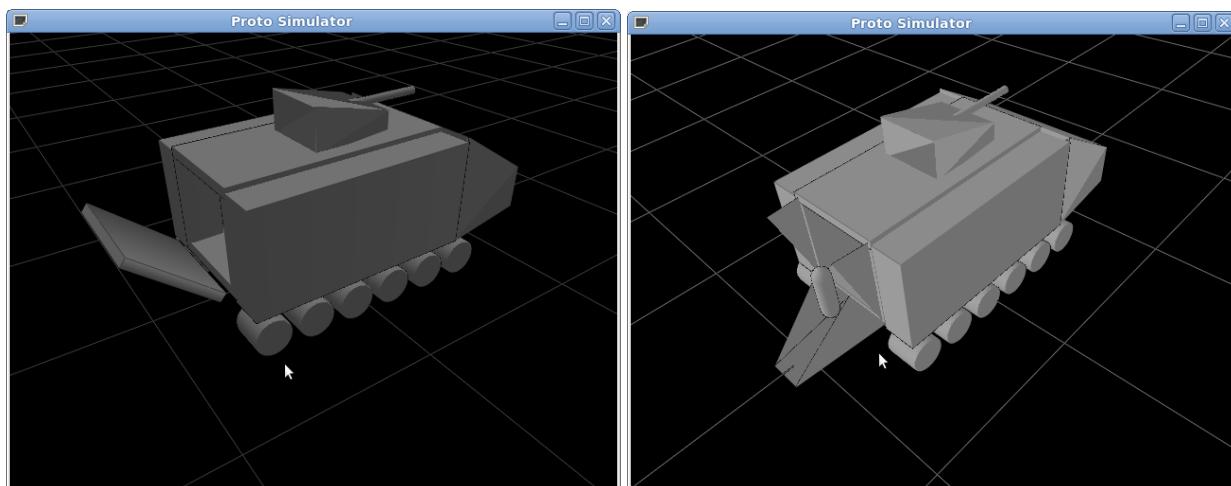
**Figure 7.8-4. Example of design change evaluation in LASS: as the passenger bay changes size (down to 4 passengers on right, up to 16 passengers on left), the number of warfighter agents needing to egress can automatically adjust, along with the agent program controlling them.**

Changing the size of the IFV passenger bay changes the number of warfighters that can be aboard as passengers (Figure 7.8-4). Such a change might be caused by an overall change of the vehicle size, or by space reallocation between the passenger bay and other adjacent components of the IFV design. Whatever the cause, when the bay changes in size, it changes the number of warfighter agents that need to egress, and therefore may change the time it takes for egress.

Changing the size of the ramp/gate may also change its size and effective profile or its opening time, and therefore the vulnerability of warfighters as they deploy.

A LASS simulation that fills the passenger bay with a given density of warfighter agents will automatically adjust to these changes in design, increasing or decreasing the number of warfighter agents as the size of the passenger bay changes, thereby allowing exploration and evaluation of design alternatives without any cost in simulator adjustment.

Some types of design change, however, will require modification of the simulation. An example would be changing the design of the ramp/gate from a single rectangular piece that swings downward to a trapezoid that swings downward and two “wings” that swing sideways instead to provide better cover for egressing warfighters (Figure 7.8-5). This possible variant trades increased ramp/gate complexity for a possible improvement in egress safety. Besides the obvious change to the IFV design model, investigating this design change may also require changes to the controller for the ramp/gate, if the “wings” are to open on a different schedule than the main ramp/gate section. The simulation then allows a rough quantification of the benefit that the additional cover would provide for the high-level capability requirement for safe egress.



**Figure 7.8-5. Example of design change evaluation in LASS: a more complex door structure (right) intended to provide better cover to egressing warfighters than the standard door structure (left). Agent-based simulation can provide a rough quantification of the benefit in safe deployment.**

### 7.8.2.3 3D Game-Engine Based Simulation

When simulating the human-vehicle interaction, the addition of physics models into the evaluation is essential to ascertain the effectiveness of the design. Without the ability to detect collisions, analyze the effects of gravity or determine how the rigid body dynamics of the vehicle alter the interaction, the results of the simulation would be meaningless.

To include high-quality physics models without the high barrier of developing a Newtonian physics model [JL02] to achieve this level of detail, we have chosen to use a 3D game-engine as part of our simulation environment. By leveraging an off-the-shelf game engine, we can reduce the complexity of the simulations, allowing the game engine to handle the low-level physics details associated with movement and interaction in a simulated world context. In addition to a robust physics model, leveraging a game engine also provides us with:

- **Geo-Typical terrain models.** Leveraging the ability to execute the simulation in terrain similar to the deployed environment, the simulator can determine the effects of the physical environment on the complete interaction. If the vehicle design results in an exit door not fully opening due to rugged terrain, a chokepoint could result, compromising the safety of disembarking soldiers. By including terrain based simulations into the design cycle, potential issues typically not observed until deployment, can be identified and addressed early in the process.
- **Robust Application Programming Interface (API) for environment interaction.** Modern 3D game environments come packaged with a set of authoring tools and programming interfaces for working with the environment. These APIs include functionality such as line of sight, and distance calculations that are essential components of the agent-based simulation. By leveraging the APIs provided, a designer can focus on analyzing the design and not on implementing the interactions between the vehicle and the environment.

### 7.8.3 Proto/Unity Integration

As part of the META project, we integrated Proto into the off-the-shelf game engine Unity 3D. By linking Proto with the game engine, we were able to blend the benefits of LASS with the rich physics model, geo-typical environment and authoring toolkit provided by the game engine.

Through a combination of game-engine scripting and a native-library bridge, we were able to develop a framework for conducting human-vehicle evaluations within a game-based simulation environment (Figure 7.8-6). With the inclusion of spatial computing to the game engine environment, we can quickly alter the troop size and tactics of the AI characters in the simulation, allowing for a range of evaluations to occur on a design with minimal modification to the tool-chain.



**Figure 7.8-6. LASS combines the realistic physics and interactive terrain modeling of game engines with Proto's robust and scalable agent control in a rich simulation environment.**

Through the use of the Unity GUI framework, we were able to quickly insert various 2D displays for controlling various aspects of the scenario, including the insertion and modification of different Proto programs in realtime.

#### 7.8.4 Deliverables

To demonstrate how Proto and Unity can work together creating a simulation environment for evaluating human-design interactions, we developed a sample application containing 30 Proto controlled agents randomly placed within a geo-typical scene. The agents are situated around an MRAP (Mine Resistant Ambush Protected) vehicle, and are controlled via the provided Proto program.

In this demonstration application, the user is free to control various aspects of the simulation such as camera angles, camera position and agent network visualizations. The demonstration tool also provides controls for redistributing the agents within the scene and also the ability to change / modify the Proto code controlling the agents.

#### 7.8.5 Bibliography

- [BB06] Beal J., Bachrach, J. “Infrastructure for engineered emergence in sensor/actuator networks” IEEE Intelligent Systems, pages 10–19, March/April 2006.
- [BBM10] Bachrach J., Beal J., McLurkin. J., “Composable continuous space programs for robotic swarms”, Neural Computing and Applications, 19(6):825–847, 2010.
- [BS10] Beal, J., Schantz, R. “A spatial computing approach to distributed algorithms”, In 45th Asilomar Conference on Signals, Systems, and Computers, November 2010.
- [DGG07] DeHon, A., Giavitto J-L, Gruau, F. 06361 abstracts collection – computing media languages for space-oriented computation. In André DeHon, Jean-Louis Giavitto, and Frédéric Gruau, editors, *Computing Media and Languages for Space-Oriented Computation*, number 06361 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [JB04] Beal, J. “Programming an amorphous computational medium”, in *Unconventional Programming Paradigms International Workshop*, September 2004.
- [JL02] Lewis, M., Jacobson J., “Game engines in scientific research”, *Communications of the ACM*, Vol. 45, No. 1, pages 27-31, 2002.
- [KLS11] Kelly, P., Landree, M., Steeb, M., *The U.S. Combat and Tactical Wheeled Vehicle Fleets, Issues and Suggestions for Congress*, The RAND Corporation, 2011.
- [MPC92] do Carmo, M. P. *Riemannian Geometry*, Birkhauser Boston, 1992.

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.9 - RMPL**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.9     Reactive Model-based Programming Language (RMPL).....</b>	<b>1</b>
7.9.1    Introduction and Overview.....	1
7.9.2    Basic Structure of an RMPL file.....	2
7.9.3    RMPL for Plant Models.....	3
7.9.3.1   Field Definitions.....	4
7.9.3.2   Value Definitions.....	6
7.9.3.3   Well Formed Formula.....	6
7.9.3.4   Method Definitions.....	7
7.9.3.5   Transition Definitions .....	8
7.9.3.6   Constructor Definitions.....	9
7.9.3.7   Examples.....	10
7.9.4    RMPL for Control Programs/Temporal Plans/QSPs and Kirk.....	11
7.9.4.1   RMPL Combinators.....	11
7.9.5    Whenever.....	15
7.9.5.1   Always .....	15
7.9.5.2   Repeat .....	16
7.9.5.3   Choose .....	16
7.9.5.4   Temporal Bound .....	17
7.9.5.5   do within { A } .....	18
7.9.5.6   slack sequence { A1; A2; ...} .....	20
7.9.5.7   slack parallel {A1; A2; ...} .....	21
7.9.5.8   optional {A} .....	22
7.9.5.9   soft parallel {A1; A2; ...} .....	23
7.9.5.10   Temporal Constraints between Subexpressions.....	23
7.9.6    Supporting Reachset Analysis with RMPL.....	25
7.9.6.1   Basic Goals for the Hybrid Language Extensions .....	25
7.9.6.2   Brief Overview of Modes in RMPL.....	26

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
DLP	Dynamic Linear Programming
DS1	Deep Space One
ESL	Executive Support Language
ME	Mode Estimation
MILP	Mixed Integer Linear Programming Solver
MPL	Model-based Programming Language
ODE	Ordinary Differential Equations
POMDPs	partially observable Markov decision processes
QSP	Qualitative State Plan
RA	Remote Agent
RMPL	Reactive Model-based Programming Language
WFF	Well Formed Formula

## 7.9 Reactive Model-based Programming Language (RMPL)

In this appendix we begin with an overview of the Reactive Model-based Programming Language (RMPL) modeling language including a historical perspective. We then describe the specific changes that were made to the RMPL language in support of Meta including the interface with the reachset analysis system.

RMPL is a general modeling language that permits the definition of automata, the connections between components in a complex system and the constraints placed on the system both in terms of the state variables of the models of the components and temporal constraints.

Additionally, as a result of the extensions added for Meta we are able to specify constraints on the (hybrid) dynamics of the modeled components as well.

RMPL models can be broadly divided into two categories, the plant models that describe the capabilities of a plant and the control program that models the way in which the plant is driven through its states.

RMPL works with a number of back-end solvers that include technology for using the models described in RMPL to perform useful functions. Some of these solvers include temporal planners (Kirk), some involve Mixed Integer Linear Programming Solver (MILP)/ Dynamic Linear Programming (DLP) solvers (Sulu), some involve diagnosing the state of a plant and generating sequences of commands that will get the plant into a desired state and keep it there for a period of time (Titan) and the list continues to grow. In support of these uses of RMPL as a modeling language there are a variety of internal automata representations that are brought into play -- some of these solvers and representations are mentioned in this appendix.

For Meta the key changes have been the extension of the RMPL modeling language to include dynamics equations, the ability to represent hybrid constraints (instead of discrete state constraints), and the connection of RMPL to the reachset analysis capability and later to the OPSAT solver in support of design space exploration.

### 7.9.1 Introduction and Overview

Numerous highly-autonomous aerospace systems, such as NASA's Deep Space One (DS1) spacecraft, the next generation of space telescopes, and various Mars Rover prototypes, are being deployed that leverage many of the fruits of Artificial Intelligence research in automated reasoning: planning and scheduling, task decomposition execution, model-based reasoning and constraint satisfaction. Yet a likely show stopper to widely deploying this level of autonomy is the myriad of languages employed for the numerous software tasks running on a spacecraft processor.

These tasks include sequencing, system monitoring, system reconfiguration, planning, and low-level control.

As a solution to this problem, we introduce the Reactive Model-based Programming Language (RMPL), which combines probabilistic, constraint-based modeling with reactive programming constructs, and offers a simple semantics in terms of partially observable Markov decision processes (POMDPs). RMPL can express a rich set of mixed hardware and software behaviors, and thus will provide a foundation for developing a unified model-based framework for autonomous robot/spacecraft control, providing integrated sequencing, monitoring and fault protection capabilities.

RMPL represents a significant evolution from the Model-based Programming Language (MPL) used in the Livingstone mode estimation and reconfiguration system flown as part of the Remote Agent (RA) on DS-1. RMPL is an object-oriented language that, like MPL, describes co-temporal interactions between subsystems (constraints), and the evolution of these interactions over time. Like MPL, it provides a language for specifying, at a commonsense level, the behavior of complex embedded systems that react to external and internal stimuli. Using RMPL, plant models can be described both in terms of their nominal behavior, and their behavior under failure.

In addition to allowing specification of plant models, RMPL provides a suite of reactive programming constructs, similar to those in Esterel, or the Executive Support Language (ESL) used to develop the RA Smart Executive. These constructs can be used to write control programs, which are specifications of the desired system behavior, and which operate directly on the hidden plant states described in the plant model. A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the control program.

### 7.9.2 Basic Structure of an RMPL file

An RMPL file is a sequence of class definitions rather like a Java program is.

```
class Name1 {
...
}
class Name2 {
...
}
...
class Namen {
...
}
```

A class has a domain of **values** over which it ranges, a set of callable **methods** that may change the value of the class instance and related/connected instances, and a set of un-commanded **transitions**. The body of a class definition consists of, in no particular order, field, mode, method, constructor, and transition, definitions. Value definitions define values that instances of the class can take on. Fields provide the ability for the component to have state and to refer to other components. Constructor definitions allow components to be connected together by assignment, transitions define how a method can autonomously transition to new values and methods define primitive commands supported by the component and non-primitive methods for controlling the process of achieving a desired new value. The start definition supports to describe general probability distribution of start modes.

```
class MyClass {
    value definitions
    field definitions
    method definitions
}
```

```

transition definitions
constructor definitions
start definitions
}

```

By convention class names start with an upper case letter. Field and Value names begin with a lower case letter and constructors must be the same name as the class (and thus begin with an upper case letter). These are just conventions to aid in readability and the language does not enforce them.

### 7.9.3 RMPL for Plant Models

A plant is a collection of components connected together to make the system which is the plant. In RMPL each component is represented by an instance of a class. An RMPL class therefore represents a class of components that can be instantiated and used in a plant definition. The class instances can be connected together by using a constructor method. A class can define fields that can refer to other class instances. A class defines what values the component can have and how it can transition between those values. A switch component could for example have values ‘On’ and ‘Off’. We would represent such a switch object as follows:

```

1: class DigitalValues { value high; value low; }

2:

3: class Switch {
4:   DigitalValues output;
5:   initial value off = (output==low);
6:   value on =(output==high);
7: }

```

The first class defines a set of values (high and low in this case). These values can be assigned to a field of type DigitalValues. The second class Switch introduces a component with state. It has a single field called output whose type is DigitalValues. It in turn defines a value ‘off’ that is true if the ‘output’ field is ‘low’. The prefix ‘initial’ in the value definition says that the switch is initially in the ‘off’ state in which the field ‘output’ has the value ‘low’.

A second value definition defines a value ‘on,’ which is true if the output field has the value ‘high’. The switch therefore defines two values that it can take on, ‘off’ and ‘on’. These values are defined by constraints over the values of fields in the component.

The above example introduces class definitions, value definitions, and field declarations. A field can take on any of the values that are defined in that class so in the above example fields of type DigitalValues can take on ‘high’ or ‘low’ and fields of type Switch can take on values of ‘off’ and ‘on’. The DigitalValues class defines a simple set of values whereas the Switch class also defines two values (‘on’ and ‘off’) but in the case of the switch these values constrain the state of fields within Switch by virtue of the constraints specified in the value definitions. Value definitions can take modifiers. So far we have seen the ‘initial’ modifier. The ‘initial’ modifier indicates the starting value of the Switch. A class can have at most one value modified by ‘initial’.

So far our switch is not very useful because it is either on or off and there is no way for it to change its value. Given that its initial value is off, it will always be off because we have specified no way for it to change its value.

A component can change its value in two ways. It may be commanded to change its value by invoking some primitive method on that class or it may simply change state autonomously, that is, without commanding. Imagine a switch that sometimes falls into the off state for no particular reason.

We can specify methods and transitions that govern how the switch changes state. In the following example we expand the definition of switch to support explicit ‘turnOn’ and ‘turnOff’ methods, whereby the switch can be commanded to change its value. We also introduce a new failure value for the Switch. Our switch will operate normally when commanded, except occasionally, when it will take on the ‘fail’ value (autonomously).

```

1: class Switch           // The Switch component type
2: {
3:   DigitalValues output;    // Output pin value - field
4:                      // representing output value
5:   // The OFF value. If the Switch is OFF, the output pin is low.
6:   initial value off = (output==low) {
7:     primitive method turnOn () => on;
8:   }
9:   // The ON value. If the Switch is ON, the output pin is high.
10:  value on = (output==high);
11:  primitive method turnOff () on => off;
12:  // This is the fault value that occurs when the observations do
13:  // not satisfy the new state constraints anticipated from the
14:  // commanded transition.
15:  failure value broken = True; // Unconstrained
16:
17:  transition fail True => broken with probability: 0.005;
18:
19:
20: }
```

Line 8 adds a method for turning on the switch when it is off. The method definition is preceded by the modifier ‘primitive,’ which indicates that the method is a primitive capability of the switch component and is not defined by a method body. The trunOn method takes no arguments and causes the switch to take on the value ‘on’. The syntax ‘=> on’ says that the result of executing the turnOn method is that the switch takes on the value ‘on’. The starting value of switch when the turnOn method is invoked must be ‘off’. This is indicated by placing the turnOn method in the body of the value ‘off’ definition. This says that the ‘from’ value of the switch must be ‘off’ in order for the turnOn() method to cause the switch to take on the value ‘on’.

In the following subsections we discuss each major element of a class definition in detail.

### 7.9.3.1 Field Definitions

From inside class Camera, power\_in can be referred to as power\_in. As an example of external reference, consider a variable my\_camera of type Camera, which has been initialized to an instance of Camera. The power\_in field of my\_camera can be accessed as my\_camera.power\_in, as long as power\_in has been declared to be public.

```
class Power {
...
}
```

```

    }
    class Camera {
        Power power_in;
        Shutter shutter;
        ...
    }
}

```

If a field is declared as private, then it can't be externally accessed. If the access (public or private) is not specified for a field, then it defaults to private.

```

private Aclass aField;      [default]
Aclass aField;             [same as above]
public Aclass aField;

```

#### 7.9.3.1.1 Variables

When a field is declared in a class, it represents an object instance of the declared class. Each field, which is either input by the constructor or created new, is a **variable** used in state constraints or transition guards.

#### 7.9.3.1.2 Flexible Templates

For convenience, a special type of fields is introduced. Using “String” field and constructor, it is allowed to declare the constraints dynamically, in other words, the state constraints or transition guards in the same class can be different depending on input of its constructor. For example, “zone” is a special field which is connected to the input argument of “Worker”. In “Main” class, we can initial two objects with different state constraint by inputting different arguments “zone1” or “zone2”.

```

class WorkerLocation { value zone1; value zone2; }

class Worker {
    WorkerLocation location;
    String zone;

    Worker(String zone) {
        zone = zone;
    }

    value off = (location != zone);
    value on = (location == zone);

    transition t1 off => on;
    transition t2 on => off;
}

class Main{
    Worker worker1;
    Worker worker2;

    observable{
        worker1.location;
        worker2.location;
    }

    Main(){}
}

```

```

        worker1 = new Worker("zone1");
        worker2 = new Worker("zone2");
    }
}

```

### 7.9.3.2 Value Definitions

In order for Mode Estimation (ME) to estimate the likelihood that a component has a given value, a value must have a **constraint** associated with it. The example below associates a constraint with the value ccc. The constraint is expressed as a logical formula,, called a well-formed formula (*WFF*); its syntax is described below.

```

value ccc = ((camera == off) &&
              (radar == off) &&
              (brake != failed));

```

#### 7.9.3.2.1 Hierarchical

In PCCA, a value definition in a class represents a mode of a component. Each component contains several modes, and all modes are only a symbol indicating the status of the component. However in PHCA, hierarchical structure is allowed. A component contains several modes, and one of these modes may be also a component containing its modes. The following example shows how to encode hierarchical plant models. Specially, “power” is a mode of the component “Camera”, but “power” is also a component “Power” with its modes “off” and “on”.

```

class Power {
    value off;
    value on;

    ...
}

class Camera {
    ...

    value off = True {
        primitive method turnOn () => power;
    }
    value broken = True;
    initial value unknown = True;
    value Power power = (switch == on);
}

```

### 7.9.3.3 Well Formed Formula

Well formed formulae allow is to build propositions that must be true for different modes of a component. Simple well-formed formulae are simple identifiers indicating a value, True, False, and an equality formula:

#### 7.9.3.3.1 Simple WFF's

```

True
False
fred
foo==bar

```

More complex propositions can be constructed by composing simple WFFs using conjunction (`&&`) disjunction (`||`) negation (not of `!`) and parenthesized expressions.

#### 7.9.3.3.2 Composed WFF's

```
not wff, !wff, foo != bar
(wff && wff && ...)
(wff || wff || ...)
(wff -> wff)
```

#### 7.9.3.3.3 Enhanced (variable == variable)

A general WFF equality in RMPL allows a variable as the left side and a value as the right side. For convenience, current RMPL allows the equality with variables on both left side and right side. The PHCA compiler will translate it into corresponding constraints.

```
class Valve {
    Flow outflow;
    Flow inflow;

    value open = (inflow == outflow);
    ...
}
```

#### 7.9.3.4 Method Definitions

The methods of a class describe the commanded transitions that instances of the class can make.

The signature of a method describes when the method is applicable, as well as the effect that the method has on the class instance and related instances. Method signatures have the following attributes:

1. Precondition: state that must be true for the method to be applicable. (default: True)
2. Postcondition: state that will be accomplished by the end of the method's execution. (default: True)
3. Invariant: state that must hold throughout the execution of the method. The precondition must imply the invariant. (default: True)
4. Arguments: the types of arguments that are passed to the method, and the classes that the arguments must be instances of.

A method definition may be preceded by a qualifier that modifies the method definition.

The qualifier if present may be one of the following:

```
primitive
public private protected
controllable uncontrollable
```

Notes: Public, private and protected are mutually exclusive (default is private). Controllable and uncontrollable are mutually exclusive (default is controllable).

A primitive method has no body and is implemented by the component primitively. Primitive methods are associated with commands. Public, private, and protected methods may be referred to, respectively, from outside the class, from inside the class only, and from within subclasses of the class, as well as the class.

A user-defined method supplies a method body, in addition to the method signature. The results of a successful execution of the method body should imply the method's postcondition.

The reason to write a user-defined method is to enforce a particular strategy to achieve the postcondition, rather than letting mode reconfiguration / planning attempt to achieve the postcondition unfettered. This ability to override mode reconfiguration is called adjustable autonomy, it allows the modeler to spell out specific steps in some cases, while leaving other, potentially unanticipated cases to be solved by mode reconfiguration.

There is a small number of control constructs that make up a user-defined method body; these are described in the section on control programs.

#### 7.9.3.4.1 Control Variables

In RMPL, control variables are encoded as primitive methods.

```
primitive method turnOn();

initial value off = True;
value on = True;

transition t1 off => on with guard: turnOn();
```

There is another alternative style:

```
initial value off = True {
    primitive method turnOn () => on;
}
value on = True;
```

#### 7.9.3.5 Transition Definitions

##### 7.9.3.5.1 Simple Transitions

Uncommanded transitions have a name, a precondition, and a postcondition, similar to those of methods. The probability given for the postcondition is the likelihood that the transition is invoked at each time step. Uncommanded transitions are introduced with the transition keyword. The following declaration specifies that the transition named “fail” may occur with .1% probability at each time step from “on” to “failed” mode.

```
transition fail on => failed with guard: True,
probability: .001;
```

There are two styles to specify transitions. The 1<sup>st</sup> style of transition specifies a source with a guard, multiple branches from the source to a set of targets with a probability respectively. It is called “And-Or Tree”. The 2<sup>nd</sup> style of transition specifies a source with a probability, multiple branches from the source to a set of targets with a guard respectively. It is called “Or-And Tree”. The simple transition can be compiled as an And-Or Tree.

### 7.9.3.5.2 And-Or Tree

And-Or Tree specifies a source with a guard, multiple branches from the source to a set of targets with a probability respectively. It is mainly used in PCCA.

```
transition t1 broken => {
    off with probability: 0.8;
    power with probability: 0.2;
} with guard: (switch == off);
```

### 7.9.3.5.3 Or-And Tree

Or-And Tree specifies a source with a probability, multiple branches from the source to a set of targets with a guard respectively. It is mainly used in PHCA.

```
transition t1 off => {
    broken with guard: (switch == off);
    power with guard: (switch == on);
} with probability: 0.8;
```

### 7.9.3.6 Constructor Definitions

A constructor method is a method that is invoked once, only when an instance of the class is created. The constructor may only contain assignments to fields of the class and instantiations of other classes. A constructor may not contain method invocations. Constructors are the way in which components are connected together, in order to make a system. The constructor name must be the same name as the class that it is a constructor for.

The example below implements a simple two input OR gate. Each input and output can be a DigitalValue. The class supports two values, nominal, in which the OR gate operates in accordance with OR gate semantics, and broken. With a low probability the gate may fail autonomously. Most of the time the gate can be returned to nominal operation by issuing a reset command.

### OrGate Example

```
class OrGate
{
    // Inputs
    DigitalValues input1; // input pin 1
    DigitalValues input2; // input pin 2

    // Outputs
    DigitalValues output; // Observed output pin
    // The nominal value. If both input pins are low,
    // the or-gate output pin is low and if either
    // input pin is high, the output pin is high.
    initial value nominal =
        (((input1==high) || (input2==high))
         && (output==high)) ||
        (output==low));
    // This is the fault value that occurs when the
    // observations do not satisfy the state
    // constraints of the or-gate.
    failure value broken = True;

    primitive method reset() broken => off
        with probability .99;
```

```

transition fail True => broken
    with probability: 0.001;
}

```

The example below builds a simple system using two switches and an Or gate. A constructor is used to connect together the components of the systems.

### Simple Circuit

```

class SimpleCircuit
{
    Switch switch1;
    Switch switch2;
    OrGate orGate;
    SimpleCircuit ()
    {
        switch1 = new Switch();
        switch2 = new Switch();
        orGate = new OrGate(switch1, switch2);
    }
}
class Main // The main class specifies the root.
{
    SimpleCircuit ourCircuit;
    Main ()
    {
        ourCircuit = new SimpleCircuit();
    }
}

```

#### 7.9.3.7 Examples

```

class Switch {
    value off;
    value on;
}

class Power {
    value off;
    value on;

    transition t1 off => on with guard: True, probability: 0.7;
    transition t2 on => off;
}

class Camera {
    Switch switch;

    initial {off} with probability: 0.9;
    initial {broken} with probability: 0.1;

    value off = True {
        primitive method turnOn () => power;
    }
    value broken = True;
    initial value unknown = True;
    value Power power = (switch == on);
}

```

```

}

class Main{
    Camera camera;

    Main(){
        camera = new Camera();
    }
}

```

#### 7.9.4 RMPL for Control Programs/Temporal Plans/QSPs and Kirk

The simplest statement is a method call. A call to a method that is defined in the same component is performed simply by using the name of the method, followed by parameters, if any, separated by commas in parentheses. This is just like Java:

```
method1(1, foo);
```

If the method is defined in another component, the method can be invoked using dotted notation, just like in Java:

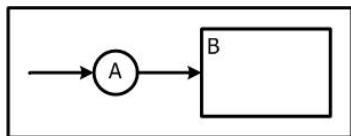
```
otherComponent.method2(3, bar);
```

Statements can be composed using one of the combinators described below.

##### 7.9.4.1 RMPL Combinators

The following RMPL combinators are based on the paper - B. C. Williams, et al., "Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior,"

###### 7.9.4.1.1 Sequence



The simplest way to compose statements is to combine them into a sequence. There are two syntactic ways of constructing a sequence:

```
sequence {A1; A2; ... }
```

creates a sequence that consists of A1, A2, etc. Since a sequence is such a common expression, the following abbreviation is supported:

```
{ A1; A2; ... }
```

Below is an example of a sequence that is contained in a method:

```

class Main {
    FOO foo;

    method run () {
        sequence {
            foo.action1();
            foo.action2();
        }
    }
}

```

```

        foo.action3() ;
    }
}

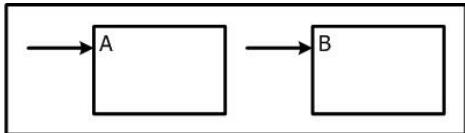
class FOO {
    method action1() {
    }

    method action2() {
    }

    method action3() {
    }
}

```

#### 7.9.4.1.2 Parallel



A parallel composition follows a similar syntactic pattern:

```
parallel { A1; A2; ... }
```

Parallel also has an abbreviated syntactic form:

```
|{A1; A2; ... }|
```

Below is an example of a parallel construct, contained in a method:

```

class Main {
    FOO foo;

    method run () {
        parallel {
            foo.action1();
            foo.action2();
            foo.action3();
        }
    }
}

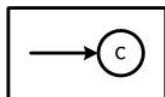
class FOO {
    method action1() {
    }

    method action2() {
    }

    method action3() {
    }
}

```

#### 7.9.4.1.3 Achieve



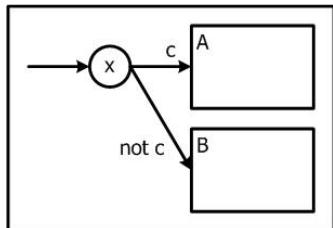
The program asserts that constraint is true at the initial instant of time. For example:

```
class Status {
    value On;
    value Off;
}

class Main {
    Status status;

    method run () {
        (status == On);
    }
}
```

#### 7.9.4.1.4 If (...else...)



If the wff evaluates to true, the process statement will be executed. For example:

```
class Status
{
    value On;
    value Off;
}

class Main {
    Status status;
    FOO foo;

    method run () {
        if (status == On) {foo.action1();} else {foo.action2();}
    }
}

class FOO {
    method action1() {
        go1();
    }

    method action2() {
        go2();
    }
}
```

#### 7.9.4.1.5 Unless

If the wff evaluates to false, the process statement will be executed. For example:

```
class Status
{
    value On;
    value Off;
}

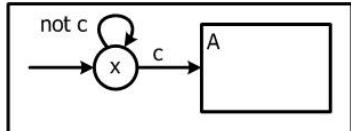
class Main {
    Status status;
    FOO foo;

    method run () {
        unless (status == On) {foo.action1();}
    }
}

class FOO {
    method action1() {
        go1();
    }

    method action2() {
        go2();
    }
}
```

#### 7.9.4.1.6 When



At the first time step in which the wff evaluates to True, the given process statement is executed in parallel with the enclosing process. The when clause is activated at most once (i.e. the first time the wff evaluates to true) during the lifetime of the enclosing process statement, for example:

```
class Status
{
    value On;
    value Off;
}

class Main {
    Status status;
    FOO foo;

    method run () {
        when (status == On) {foo.action();}
    }
}

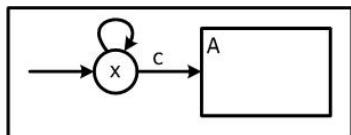
class FOO {
    method action() {
```

```

        go();
    }
}

```

### 7.9.5 Whenever



The wff is evaluated at every time step, and whenever it evaluates to true, the given process-statement is spawned in parallel, if it is not already active. For example:

```

class Status
{
    value On;
    value Off;
}

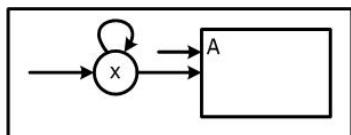
class Main {
    Status status;
    FOO foo;

    method run () {
        whenever (status == On) {foo.action();}
    }
}

class FOO {
    method action() {
        go();
    }
}

```

#### 7.9.5.1 Always



For example:

```

class Status
{
    value On;
    value Off;
}

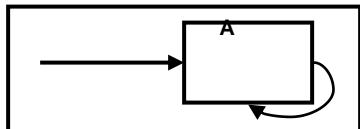
class Main {
    Status status;
    FOO foo;

    method run () {
        repeat {foo.action();}
    }
}

```

```
class FOO {
    method action() {
        go();
    }
}
```

### 7.9.5.2 Repeat



Repeat is like ‘Always’ except that ‘A’ is only executed sequentially and no multiple markings can occur as a result of using ‘repeat’. ‘Always’ spawns ‘A’ on every clock cycle and causes multiple markings.

For example:

```
class Status
{
    value On;
    value Off;
}

class Main {
    Status status;
    FOO foo;

    method run () {
        repeat {foo.action();}
    }
}

class FOO {
    method action() {
        go();
    }
}
```

### 7.9.5.3 Choose

Sometimes we want a statement to be selected by the program executive so as to maximize reward or minimize cost. Sometimes choices are made autonomously, for example, by the physical environment, based upon a probability assignment. The Choose construct implements this.

Choose allows costs, rewards, and probabilities to be specified (but at most one). This is a bug, it should allow any combination of cost, reward, and probability to be specified.

Below is a mixed example that illustrates the ways in which a choice can be made:

```
choose {
    with reward: 45 { ... }; // specify a reward
    with cost: 34 { ... }; // specify a cost
    with probability: 0.3 { ... }; // specify prob.
    with choice: { ... }; // no guidance.
```

```
}
```

For example (specifying costs):

```
class Main {
    FOO foo;

    method run () {
        choose {
            with cost: 10 [0,1] foo.action1();
            with cost: 15 [1,2] foo.action2();
            with cost: 20 [2,2] foo.action3();
        }
    }
}

class FOO {
    method action1() {
    }

    method action2() {
    }

    method action3() {
    }
}
```

For example (specifying probabilities):

```
class Main {
    FOO foo;

    method run () {
        choose {
            with probability: 0.1 [0,1] foo.action1();
            with probability: 0.3 [1,2] foo.action2();
            with probability: 0.6 [2,2] foo.action3();
        }
    }
}

class FOO {
    method action1() {
    }

    method action2() {
    }

    method action3() {
    }
}
```

#### 7.9.5.4 Temporal Bound

Any statement, primitive or composed, can be given an optional temporal constraint, by preceding it with a temporal bound, specified within square brackets, like this:

```
[0, 7] method1(1, foo);
```

A temporal bound specifies the lower and upper bound on the time for method1 to complete. In the above example, method1 can take between 0 and 7 time units to complete.

A bound without a following statement, means do nothing; for example:

```
[0, 7];
```

#### 7.9.5.4.1 Derived Constructs

Each of the following constructs can be considered “syntactic sugar” for combinations of existing RMPL constructs.

1. do within {A<sub>1</sub>; A<sub>2</sub>; ... A<sub>n</sub>}
2. slack sequence {A<sub>1</sub>; A<sub>2</sub>; ... A<sub>n</sub>}
3. slack parallel {A<sub>1</sub>; A<sub>2</sub>; ... A<sub>n</sub>}
4. optional { A }
5. soft sequence { A<sub>1</sub>; A<sub>2</sub>; ... A<sub>n</sub> }
6. soft parallel {A<sub>1</sub>; A<sub>2</sub>; ... A<sub>n</sub>}

We will consider each of the above in turn, by looking at their expansion. We group them into two classes: slack constructs (1-3) and soft constructs (4-6).

#### Execution of Expressions with Slack in Execution Time

In the expressions discussed thus far, one subexpression begins as soon as the preceding expression ends. For many applications there exists temporal slack in the execution time of activities; the program executive can then choose to execute these activities at the time with greatest utility. For example, a time window may be specified in which an expression must be executed, but the executive may choose to sit idle before starting the expression, as long as the time to completion is satisfied. Likewise it might be best to idle after the expression, or between successive expressions, as long as the time to completion is satisfied. To model this behavior we introduce three constructs: **do within**, **slack sequence** and **slack parallel**.

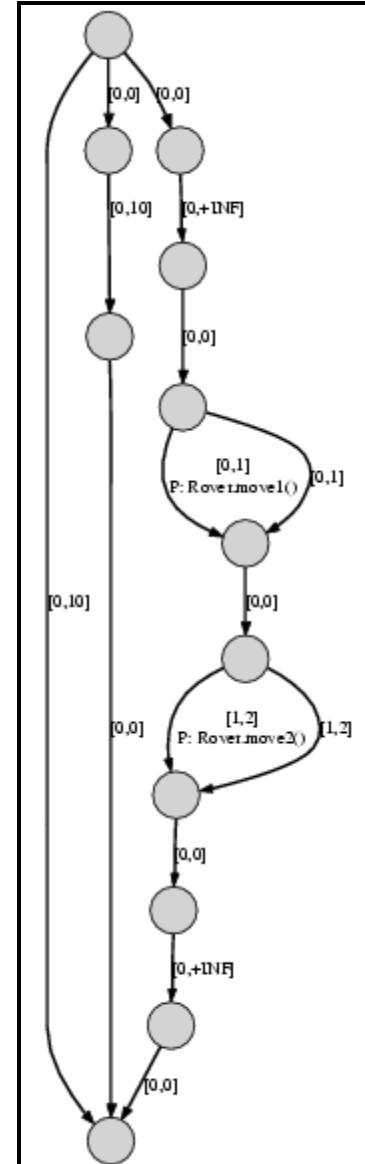
#### 7.9.5.5 do within { A }

The statement **do within** allocates a time window, and specifies that an expression A must be started and completed sometime within that window. The do within expression completes at the end of the window. In particular, the expression:

```
[lb, ub] do within { A }
```

is equivalent to:

```
parallel {
```



```
[lb,ub] noop();
sequence {
  [0, inf] noop();
  A;
  [0, inf] noop();
}
}
```

An example of its use is as follows, which mean do A somewhere within the temporal bounds  $[lb, ub]$  allowing slack on either side of A. Consider the following example:

```
class main {
  Rover foto;

  method run () {
    [0,10] do within {
      [0,1] foto.move1();
      [1,2] foto.move2();
    }
  ...
}
```

In this example, foto.move1 and foto.move2 can occur anywhere within the interval  $[0, 10]$ . Note, however, that move1 and move2 cannot have a gap between them.

### 7.9.5.6 slack sequence { A1; A2; ... }

Executes a sequence of activities, while permitting optional gaps between successive activities. The expression:

```
[l, u] slack sequence { A1; A2; ... }
```

is equivalent to:

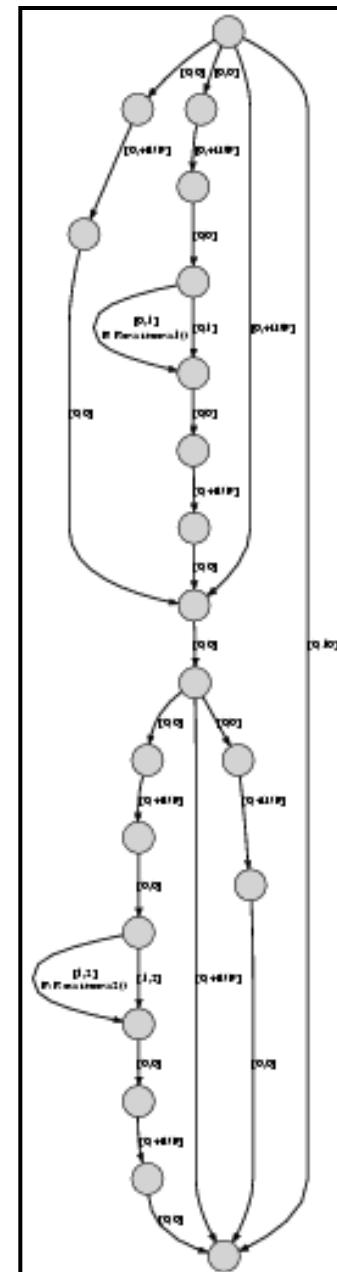
```
[l, u] sequence {
    [0, inf] do within {A1};
    [0, inf] do within {A2};
    ...
}
```

This expression means that successive activities  $A_i$  and  $A_{i+1}$  can be separated by an arbitrary gap within  $[l, u]$ .

Below is an example, along with its graphical depiction to the right.

```
class main {
    Rover foto;

    method run () {
        [0,10] slack sequence {
            [0,1] foto.move1();
            [1,2] foto.move2();
        }
    }
}
```



### 7.9.5.7 slack parallel {A1; A2; ...}

Slack parallel is the analogue of slack sequence, but for parallel composition. In particular, slack parallel is similar to parallel, but permits idle time both before and after each of the parallel activities, while enforcing the constraint that all activities complete within the time bound of the slack parallel expression.

In particular, an expression of the form:

```
[lb, ub] slack parallel {A1; A2; ...}
```

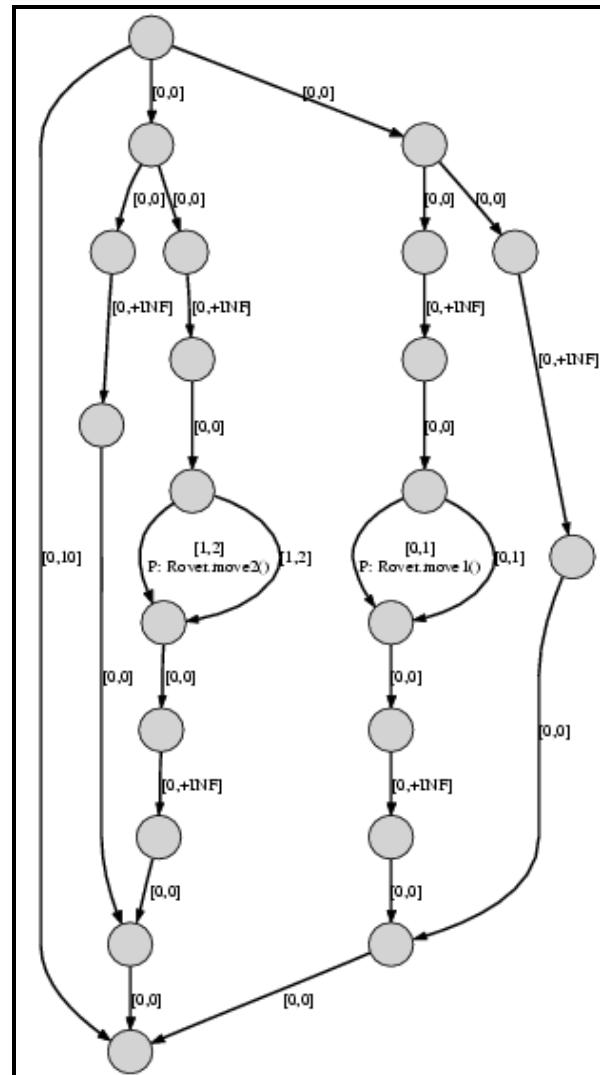
is equivalent to:

```
parallel {
    [lb, ub] do within { A1 };
    [lb, ub] do within { A2 }; ...
}
```

An example application of slack parallel is shown below, together with its graphical depiction, which means that A1 and A2 which are executed in parallel are each allowed to vary within their specified temporal window.

```
class main {
    Rover foto;

    method run () {
        [0,10] slack parallel {
            [0,1] foto.move1();
            [1,2] foto.move2();
        }
    }
}
```



## Execution of “Soft” Expressions, Containing Optional Activities

Above we introduced constructs that exploit temporal slack within a system. Conversely, for many applications, the set of activities specified to be performed is overly optimistic; however, completion of many of these activities may not be essential to mission success. In this case we would like the executive to drop activities, in order to ensure overall mission success; we refer to these as *soft activities*. To model this behavior we introduce three constructs: **optional**, **soft sequence** and **soft parallel**.

### 7.9.5.8 optional {A}

specifies that the execution of A is optional. This is equivalent to:

```
choose {
    with choice {[0,0]};
    with choice{ A };
}
```

and means to either do A or continue on immediately.

The following is an example of the use of optional.

```
class main {
    Rover foto;

    method run () {
        optional {
            [0, 1] foto.move1()
        }
    }
    ...
}
```

#### 7.9.5.8.1 soft sequence {A1; A2; ...}

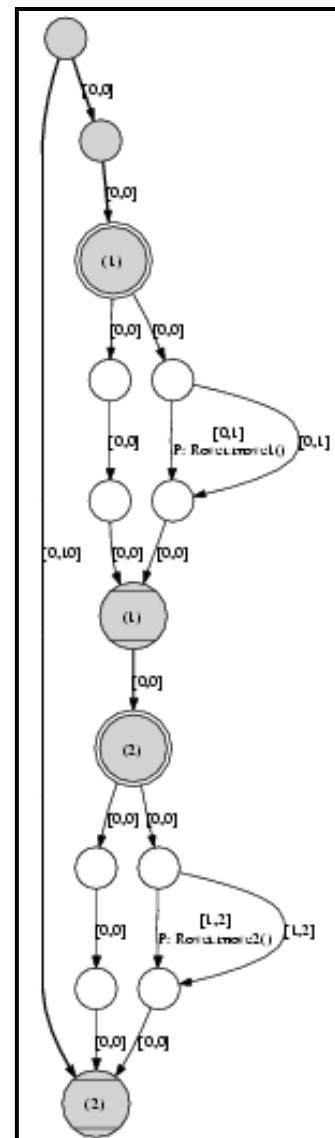
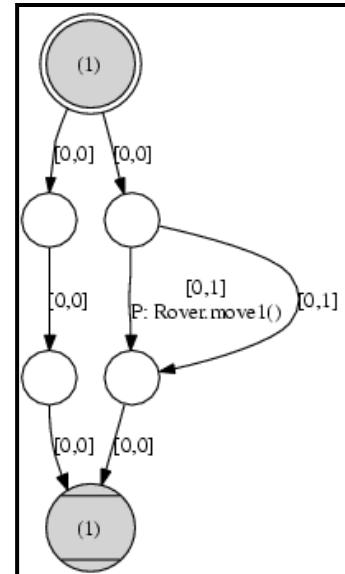
Soft sequence is similar to sequence, but permits any number of activities in the sequence to be dropped; for example, in order to ensure that the sequence terminates within its time-bound. An expression of the form:

```
[l, u] soft sequence {A1; A2; ...}
```

is equivalent to:

```
[l, u] sequence {
    optional {A1};
    optional {A2};
    ...
}
```

The following is an example of the application of soft sequence, together with its graphical depiction to the right, which means that either or both of A1 and A2 may be dropped (goal shedding) in



order to meet the temporal constraints. If rewards are provided for the activities, the planner will select what to drop on the basis of total reward.

```
class main {
    Rover foto;

    method run () {
        [0,10] soft sequence {
            [0,1] foto.move1();
            [1,2] foto.move2();
        }
    }
}
```

### 7.9.5.9 soft parallel {A1; A2; ...}

Soft parallel is the analogue of soft sequence, but for parallel composition. In particular, soft parallel is similar to parallel, but permits any number of activities in the parallel to be dropped; for example, in order to ensure that each parallel activity terminates within the time-bound of the slack parallel.

In particular, the expression:

```
[lb,ub] soft parallel { A1; A2; ... }
```

is equivalent to:

```
[lb,ub] soft parallel {
    optional { A1 };
    optional { A2 };
    ...
}
```

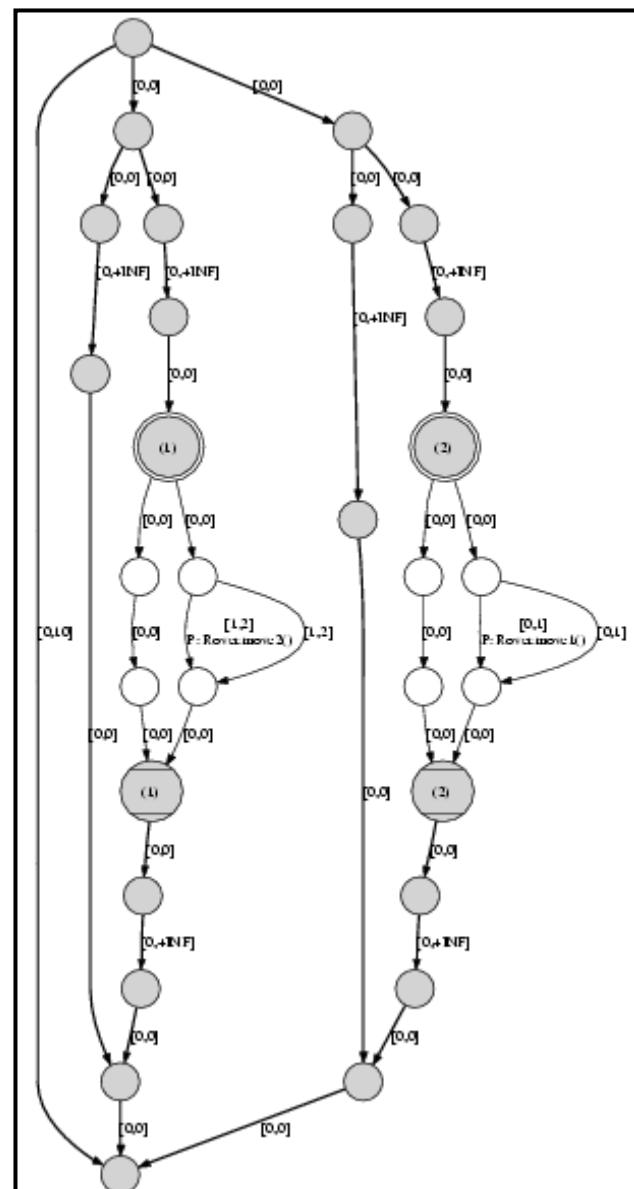
The following is an example of the use of soft parallel, together with its graphical depiction to the right, which means that the activities are executed in parallel but either or both of them may be dropped in order to meet temporal constraints.

```
class main {
    Rover foto;

    method run () {
        [0,10] soft parallel {
            [0,1] foto.move1();
            [1,2] foto.move2();
        }
    }
}
```

### 7.9.5.10 Temporal Constraints between Subexpressions

It is often sufficient to specify temporal constraints in terms of bounds on the execution time of an expression. However, in many important applications, we need to



specify temporal constraints between the start and end times of the subexpressions that appear within some expression. We accommodate this need through the introduction of “constrained” expressions.

A constrained expression augments the expressions, defined above, with a set of temporal constraints. A constrained expression allows the start and end events of its subexpressions to be labeled. A set of temporal constraints between the start and end times of these subexpressions is then specified, in terms of these labels.

In particular, two special operators, **startof** and **endof**, are introduced to refer to the start and end events of any labeled subexpression. Constraints are then specified as inequalities between any two labeled events within the constrained expression.

For a given constrained expression, all labeled subexpressions whose labels appear in temporal constraints of the constrained expression must appear within the constrained expression. Note, however, that the label may appear at any nesting level. In addition, any subexpression may be labeled.

A special syntax allows gaps to be constrained. “startof(name3) - endof(name2) in [lb, ub]” means that the activity name2 must end no less than “lb” after activity name3 begins and no longer than “ub” after name3 begins.

### Examples:

```

constrained parallel {
    name2: [ 0, 5 ] parallel {
        lab1: foo();
        lab2: baz();
    }
    name3: [ 0, 10 ] sequence {
        lab3: bar();
    }
}
(startof(lab1)<endof(lab3),
 endof(name3)>endof(name2),
 startof(name3)-endof(name2) in [lb, ub])

```

## 7.9.6 Supporting Reachset Analysis with RMPL

In order to support the use of RMPL models by the reachset analysis code it was necessary to do two things. First, it was necessary to build a new back end for the RMPL compiler that would target (emit) the models in Matlab, and second it was necessary to extend the RMPL modeling language to support hybrid models.

The extension of RMPL to support hybrid was done in a manner consistent with the overall goals of the language as a general modeling language. This permits models to be exported to a variety of solver back ends beyond the immediate need of Reachset Analysis. In particular it has allowed us to integrate RMPL with OPSAT in order to experiment with design space exploration.

### 7.9.6.1 Basic Goals for the Hybrid Language Extensions

We wished to extend the RMPL language to support hybrid models. Our goals included some that were specific to Reachset Analysis and some that were more general.

Goals:

1. We wished to enable the encoding within a full plant/control model all parameters and values that were handled through a separate file. for example, in Sulu we augment the RMPL model with a separate file that contains parameters. We wished to be able to support all such needs within a common linguistic framework. There is still a role for parameter files because one doesn't always want to encode the values in the model – but often we do.
2. The Matlab code implementing reachset analysis used external tools that required the dynamics equations to be represented in the form  $Ax+By+C$  where A, B, and C are matrices. While this is a requirement for the reachset analysis code it is also a common requirement. Since RMPL is not a matrix based language it would have been a strange departure from regular RMPL syntax to directly incorporate the matrix notation into RMPL. Instead, as shown below, we opted to extend RMPL to represent dynamics equations in a normal RMPL-like equation syntax and have the RMPL compiler construct the matrix reformulation of the equations.
3. It has been the practice of RMPL users to build a single file that contains the plant model as well as the control program, or Qualitative State Plan (QSP). this has long been an annoyance because ideally one would like to have a collection of separate RMPL plant models representing a library of parts and to have the control program/QSP include the parts that it wanted. This would enable new QSP's to be generated automatically that simply referred to the plant models that it used and it would facilitate the interchange of plant models such as when replacing one part, say a solenoid, with another part with slightly different properties such as we might want to do with design exploration. To support this need we introduced the simple notion of an include file such as is familiar to C++ programmers.
4. Model variables can be divided into several categories, broadly we can characterize the variables and endogenous variables having no connection outside the model and exogenous variables that represent the inputs and outputs to the model. Some earlier versions of RMPL had maintained the distinction explicate in the syntax whereas the current incarnation of RMPL has offered no support in the syntax to distinguish them.

We found the need to re-introduce syntax into RMPL to allow state variables to be annotated as exogenous or endogenous so that the compiler could use this information in generating the Ax+By+C formulation of the dynamics equations for the reachset analysis.

5. Reachset analysis as implemented by the MIT code base requires that the dynamics models be linear. While we wouldn't want to impose that restriction on RMPL in general it is important that RMPL be able to impose that constraint so as to provide useful feedback to the RMPL user that his model has strayed out of the bounds of what the solver will support.

#### 7.9.6.2 Brief Overview of Modes in RMPL

A plant in RMPL has modes and each mode specifies a proposition that is true in that mode. A plant has state variables that may be either boolean or a defined discrete value type.

For example:

```

1: class DigitalValues { value high; value low; }

2:

3: class Switch {
4:   DigitalValues output;
5:   initial value off =(output==low);
6:   value on =(output==high);
7: }
```

On line 1 we define a discrete type `DigitalValues` that can take on the values 'high' or 'low'. The switch, defined from line 3 to line 7, has an initial value of 'off' defined as `output==low` and a value on defined as `output==high`.

For hybrid models, we extended the value types that RMPL supports as follows:

1. Built-in support for numeric types.
2. Numeric inequalities.
3. Full equation syntax (same as Java).
4. Full support for computing the value of constant expressions.
5. Support for representing ODE's
6. Support for representing linear programming (LP) models.
7. A compile time arithmetic expression interpreter to fold constant expressions.

For numeric datatypes, initially we have implemented Real, and Integer. At some later time we may need to add Complex.

A location contains a set of differential and algebraic equations and one invariant that defines continuous behaviors while the automaton is executing in the location. One goal is to support the use of LP solvers such as in Sulu. For an LP solver we need to be able to express the model in terms of linear equations and constraints. An LP solution requires that the model be compiled into a set of variables representing values at different time points. A recurrence relation representation is ideal for such compilation.

A differential equation has the form:

$$\dot{x} = f(x_1; \dots; x_n)$$

$\dot{x}$  is understood as the first derivative of  $x$  with respect to time.

Our new syntax uses the `differential(x) = <formula>` to represent the differential equation.

An algebraic equation has the form:

$$x = f(x_1; \dots; x_n)$$

Here the simple syntax `x = <formula>` is sufficient.

The invariant is a proposition, as is presently the case, but with the addition of a full inequality syntax over real and integer values.

The main addition here is the ability to specify an equation that models the evolution of the variable value over time. If such an equation is not provided, the value is assumed to be constant while the automaton is at that location. That is, the equation  $\dot{x} = 0$  is assumed for every such variable. For each location, we have to ensure that the set of algebraic equations has a uniquely well-defined solution. This is guaranteed by requiring that the variable dependency relation for algebraic equations is not circular.

Instead of representing system dynamics as an Ordinary Differential Equations (ODE) it is often convenient to express them as recurrence relations. We added this syntax to support recurrence relations as follows:

Example: **recurrence position[+] = position[.] + dt\*velocity[.]**

The keyword **recurrence** introduces a recurrence relation over state variables. Within a recurrence relation state variables are followed by either **[+]** or **[.]**. **[+]** means the value at the next time step and **[.]** means the value at the current time step. Additionally a pseudo variable **dt** is added to quantify the time step.

Here is an example taken from one of our recent demos of this new capability the new syntax is shown in bold.

State variables are introduced as before but they may now be qualified with either **endogenous**, **exogenous**, **input**, or **output**. Unqualified state variables are assumed to be endogenous. **input** and **output** qualifiers both indicate exogenous.

The RMPL version of the current demo would look like this:

```
// Plant Model

class Ramp
{
    // Constants
    constant real g=9.81;
    constant real m=1000;
    constant real rcg=3;
    constant real Ir = 1;
    constant real Kr = 1; // what value is required here?

    // State variables
    state real position; // angular position
    state real velocity; // angular velocity

    // Input Variables
    input real torque;

    // Output Variables
    output real outp;

    real initPosMin;
```

```

real initPosMax;
real initVelMin;
real initVelMax;
real goalPosMin;
real goalPosMax;
real goalVelMin;
real goalVelMax;
real torqueMin;
real torqueMax;

// Initializing the Ramp
// This involves calculating and setting the input constraints
Ramp (real theta_nom)
{
    torqueMin = -1000;
    torqueMax = 1000;
}

run (real initMinPos, real initMaxPos, real initMinVel, real initMaxVel,
      real goalMinPos, real goalMaxPos, real goalMinVel, real goalMaxVel)
{
    initPosMin=initMinPos;
    initPosMax=initMaxPos;
    initVelMin=initMinVel;
    initVelMax=initMaxVel;
    goalPosMin=goalMinPos;
    goalPosMax=goalMaxPos;
    goalVelMin=goalMinVel;
    goalVelMax=goalMaxVel;
}

// Ramp modes
value stopped=(velocity==0);

initial value moving={
    recurrence position[+] = position[.] + dt*velocity[.],
    recurrence velocity[+] = -dt*g*m*rcg*sin(theta_nom)*position[.]/Ir -
}

```

```

        Kr*velocity[.] + dt*torque[.]/Ir-
        dt*g*m*rcg*cos(theta_nom)/Ir +
        dt*g*m*rcg*sin(theta_nom)*theta_nom/Ir,
        velocity>0
    } ;
}

```

In order to support the separation of plant models from QSP's we introduced the include syntax that works much as it does in C++. Here is an example of a QSP that refers to the plant model by using an include:

```

// Control Plan (QSP)

#include "C:/meta/RMPLRampMeta/ExampleRampGeneratePlantv1.rmpl2"

class Main
{
    Ramp Activity79;
    Ramp Activity215;
    Ramp Activity261;

    Main ()
    {
        Activity79 = new Ramp(0.0);
        Activity215 = new Ramp(0.25);
        Activity261 = new Ramp(1.5);
    }

    method run ()
    {
        sequence {
            [4.0, 6.0] Activity79.run(-5.0, 5.0, -5.0, 5.0, 0.15, 0.35, 0.07,
0.13);
            [9.0, 11.0] Activity215.run(-4.75, 5.25, -4.9, 5.1, 1.15, 1.35, 0.07,
0.13);
            [4.0, 6.0] Activity261.run(-3.75, 6.25, -4.9, 5.1, 1.4, 1.6, -0.03,
0.03);
        }
    }
}

```

```

    }
}

}

// End of generated QSP

```

The include statement includes the plant model shown before. The QSP shown above was generated automatically from an envisionment from Parc along with a magic number file that provided all of the necessary numbers that are not present in the Parc generated envisionment.

The compilation of the above generates the following output files for use with the matlab based reachset-analysis described elsewhere:

Here is the command line output from the run:

```

Including file: C:/meta/RMPLRampMeta/ExampleRampGeneratePlantv1.rmpl2
Exiting include file
Compiling RMPL 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1.rmpl2'.
Generating XML 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1.rmpl2.xml'.
RMPL read in as XML...
Compiling RMPL 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1.rmpl2' to Matlab...
Done!
Generating Matlab file 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1_events.m
Generating Matlab file
'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1_plantmodels.m
Generating Matlab file 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1_pm.m
Generating Matlab file 'C:/meta/RMPLRampMeta/ExampleRampGenerateQSPv1_qsp.m

```

We will look at the last two generated files here (shown in bold above).

First the QSP file:

```
%%%%%%%
% Automatically Generated and emitted from RMPL Compiler
% Source file: C:\meta\RMPLRampMeta\ExampleRampGenerateQSPv1.rmpl2
%%%%%%%

function [qsp_set] = ExampleRampGenerateQSPv1_qsp()

Ramp_g = 9.81;
Ramp_m = 1000;
Ramp_rcg = 3;
Ramp_Ir = 1;
Ramp_Kr = 1;

% State variables for this model:
% [Ramp_position, Ramp_velocity]

% Input variables for this model:
% [Ramp_torque]

% Output variables for this model:
% [Ramp_outp]

% Activity 1 parameters:
parameters.theta_nom = 0.0;
activity_1 = ExampleRampGenerateQSPv1_Activity([-4.0, 6.0], [-5.0, 5.0], [-5.0, 5.0], [0.15, 0.35], [0.07, 0.13], parameters);

% Activity 2 parameters:
parameters.theta_nom = 0.25;
activity_2 = ExampleRampGenerateQSPv1_Activity([9.0, 11.0], [-4.75, 5.25], [-4.9, 5.1], [1.15, 1.35], [0.07, 0.13], parameters);
```

```
% Activity 3 parameters:
parameters.theta_nom = 1.5;
activity_3 = ExampleRampGenerateQSPv1_Activity([-4.0, 6.0], [-3.75, 6.25], [-4.9, 5.1], [1.4, 1.6], [-0.03, 0.03], parameters);

% Insert activities into QSP
qsp.plant_vector(1).activities(1) = activity_1;
qsp.plant_vector(2).activities(1) = activity_2;
qsp.plant_vector(3).activities(1) = activity_3;

% Events
[qsp.events, qsp.temporal_constraints] = ExampleRampGenerateQSPv1_events(3);
```

`qsp_set = [qsp];`

Next the plant model file:

```
%%%%%%
% Automatically Generated and emitted from RMPL Compiler
% Source file: C:\meta\RMPLRampMeta\ExampleRampGenerateQSPv1.rmpl2
%%%%%%

function [sysStruct, probStruct] =
    ExampleRampGenerateQSPv1_pm(delta_t,      inputbounds,      probbound,      qsp,
index_plant)

Ramp_g = 9.81;
Ramp_m = 1000;
Ramp_rcg = 3;
Ramp_Ir = 1;
Ramp_Kr = 1;

%%%%%

% State variables for this model:
% [Ramp_position, Ramp_velocity]

% Input variables for this model:
```

```

% [Ramp_torque]

% Output variables for this model:
% [Ramp_outp]

% Modes and Dynamics Equations:

Ramp_stopped_Mode = 0
Ramp_moving_Mode = 1

Ramp_Mode = Ramp_moving_Mode;

parameters = qsp.plant_vector(index_plant).activities(1).parameters;
Ramp_theta_nom = parameters.theta_nom;

Ramp_stopped_DYNAMICSA = [1, 0; 0, 1];
Ramp_stopped_DYNAMICSB = [0; 0];
Ramp_stopped_DYNAMICSf = [0; 0];

Ramp_moving_DYNAMICSA =
    [1,                                     delta_t;
delta_t*Ramp_g*Ramp_m*Ramp_rcg*sin(Ramp_theta_nom)/Ramp_Ir, -Ramp_Kr];
Ramp_moving_DYNAMICSB = [0; delta_t/Ramp_Ir];
Ramp_moving_DYNAMICSf =
    [0;
delta_t*Ramp_g*Ramp_m*Ramp_rcg*sin(Ramp_theta_nom)*Ramp_theta_nom/Ramp_Ir-
        delta_t*Ramp_g*Ramp_m*Ramp_rcg*cos(Ramp_theta_nom)/Ramp_Ir];

sysStruct.A = Ramp_moving_DYNAMICSA;
sysStruct.B = Ramp_moving_DYNAMICSB;
sysStruct.C = eye(2);
sysStruct.D = zeros(2, 1);
sysStruct.f = Ramp_moving_DYNAMICSf;

% WHERE DOES THIS COME FROM

inputbounds.umax = inputbounds.umax-probbound;
inputbounds.umin = -inputbounds.umax;

```

```
sysStruct.ymin = [-1000000; -1000000];
sysStruct.ymax = [1000000; 1000000];
sysStruct.umin = [inputbounds.umin];
sysStruct.umax = [inputbounds.umax];

% WHERE DOES THIS COME FROM

input_cost_weight = 0.01;
probStruct.Q = zeros(2, 2);
probStruct.R = eye(1) * input_cost_weight;
probStruct.norm = 2;
probStruct.subopt_lev = 0;
probStruct.tracking = 0;
```

The dynamics equations are shown in bold for the stopped and moving modes of the system in  $Ax+By+c$  form as matrices.

Several other files are also generated that involve hooking the QSP and plant models into the reachset analysis system but we have not included them here because they are only intelligible by someone with a deep understanding of the reachset analysis code.

# META Adaptive, Reflective, Robust Workflow (ARRoW)

## Phase 1b Final Report TR-2742

### Appendix 7.10 – Verification using Hybrid Models with State and Temporal Flexibility

13 October 2011

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency

3701 North Fairfax Drive

Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)

4800 East River Road

Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

<b>7.10 Verification using Hybrid Models with State and Temporal Flexibility.....</b>	<b>1</b>
7.10.1 Executive Summary .....	1
7.10.2 Motivating Examples.....	2
7.10.2.1 Automatic Transmission Vehicle .....	3
7.10.2.2 Hill Climbing Vehicle.....	5
7.10.2.3 Elevator.....	6
7.10.2.4 Infantry Fighting Vehicle Ramp .....	7
7.10.3 Verification Process Overview.....	7
7.10.3.1 Use of Hybrid Model in Verification Process.....	8
7.10.4 Simple Example of Hybrid Model.....	9
7.10.5 Formal Specification of Hybrid Timed PCCA .....	13
7.10.5.1 PCCA .....	13
7.10.5.2 Extension to Hybrid.....	14
7.10.5.3 Extension to Timed.....	15
7.10.6 Mode Specifications.....	15
7.10.7 Verification through Qualitative Simulation using Hybrid Timed PCCA .....	15
7.10.7.1 Qualitative Simulation Problem Definition .....	17
7.10.7.2 Qualitative Simulation Examples.....	18
7.10.7.3 Compilation of Hybrid Timed PCCA.....	27
7.10.7.4 Qualitative Simulation of a Compiled Hybrid Timed PCCA.....	30
7.10.8 Bibliography.....	39

## List of Figures

Figure 7.10-1. Mode transition diagram for vehicle with automatic transmission. ....	3
Figure 7.10-2. Use cases for automatic transmission example.....	4
Figure 7.10-3. Hybrid model representing vehicle that goes up and down hills.....	5
Figure 7.10-4. Use case representing vehicle that goes up and down hills. ....	6
Figure 7.10-5. Simple elevator door example with upper temporal bound on Door Open mode.	6
Figure 7.10-6. Verification Process.....	9
Figure 7.10-7. Simple elevator door example. ....	10
Figure 7.10-8. Flow tube for door open.....	11
Figure 7.10-9. Flow tube for door open, with door closed initial constraint.....	12
Figure 7.10-10. Model of elevator movement between three floors. ....	13
Figure 7.10-11. Simple elevator door example with upper temporal bound on Door Open mode. ....	16
Figure 7.10-12. Simple velocity limited system for getting from one position to another. ....	19
Figure 7.10-13. Simple velocity limited system for getting from one position to another. ....	23
Figure 7.10-14. Simple velocity limited system for getting from one position to another. ....	25
Figure 7.10-15. Set bounds cover a percentage of the noise distribution. ....	35
Figure 7.10-16. Set bounds on noise are translated to safety bounds on the input. ....	36
Figure 7.10-17. Flow tube set for two successive activities, assuming no disturbances. ....	37
Figure 7.10-18. Flow tube set for two successive activities, with input disturbances.....	38

## List of Symbols, Abbreviations, and Acronyms

Symbol, Abbreviation, Acronym	Definition
HTPCCA	Hybrid Timed PCCA
OPSAT	Optimization Solver
PCCA	Probabilistic Concurrent Constraint Automata
QSP	Qualitative State Plan
RMPL	Reactive Model-based Programming Language
STN	Simple Temporal Networks

## 7.10 Verification using Hybrid Models with State and Temporal Flexibility

### 7.10.1 Executive Summary

For this project, MIT has developed capabilities in two areas: design verification, and design optimization. Additionally, we have advanced and adapted the RMPL (Reactive Model-based Programming Language) for use in these capabilities.

#### Design Verification

We have developed a verification capability for the design of dynamic electro-mechanical systems, such as vehicles. This capability combines recent advances in qualitative simulation, reach set analysis, optimization, and hybrid systems modeling. In particular, we have developed hybrid models that appropriately capture the state and temporal limits of the mechanism being designed, as well as the flexibility limits on tasks they are required to perform. This provides a powerful foundation for analysis of a design's performance.

Our system is intended primarily for use during the requirements analysis/conceptual design phases of a project. This is where the most important decisions are made. Also these phases often take more time and effort than the detailed design and implementation; the Bradley fighting vehicle project is one example of this. Detailed design typically requires specialized tools, such as CAD systems and detailed nonlinear simulations, which are beyond the scope of our system. Further, good tools for detailed design already exist, but tools for requirements analysis are less well developed. Therefore, most of the opportunity for improvement is in the upstream phases of a project. Our focus is in line with the overall META goal of resolving important decisions early in the design process, thereby dramatically shortening project time.

A key feature of our approach is the ability to evaluate a design of an electro-mechanical system based on the physical design limits, independently of design of a control policy. This is in contrast with standard existing techniques that use randomized forward simulations, which require implementation of both a model of the design, as well as a control policy. This is problematic for two reasons. First, designers may want to defer control policy implementation decisions and focus on the electro-mechanical design. Second, by introducing the control aspect, the design optimization problem becomes more complicated, as both electro-mechanical design and control policy are being optimized simultaneously. By using novel, recently developed reach set analysis technology, we are able to decouple electro-mechanical design optimization from control policy design optimization.

A second key feature of our approach is a specialized abstraction of quantitative model information into qualitative models. This abstraction is crucial in that it allows for fast verification, but maintains sufficient quantitative information, represented as qualitative regions, to allow for meaningful analysis. It is our experience that qualitative simulations that are devoid of quantitative information are not useful for design verification. In particular, due to the complex interplay between discrete and continuous constraints in typical designs, the omission of quantitative information results in an in-sensitivity to design parameters. The result is that the qualitative simulations are always the same; they have the same outcome for different design parameter settings. Our system avoids this by keeping sufficient quantitative information, abstracted into qualitative regions.

A third key feature of our approach is minimizing the information that users have to specify in test cases for the design. For example, the user shouldn't have to specify the details of plant models and detailed task goals for each test case. By maintaining much of this information in

the hybrid model, which is shared by all test cases, the need for repetitious and needless specification of "magic numbers" and other information in test cases is minimized.

## Design Space Analysis and Optimization

When building a design, one is frequently faced with the challenge that the problem is over constrained and that as stated there is no way to achieve the requirements. To resolve this impasse it is necessary to relax some of the constraints (requirements). Our goal with design space exploration is to provide a tool that enables designers to explore the space of constraint relaxations and the impact that they have upon requirements. The approach hinges upon two key capabilities: (a) an explanation capability; and (b) a means of representing conditional preferences for the requirements.

The explanation capability produces an abstraction of a diagnosis of the failure to produce a solution as a set of diagnosis kernels that implicate a collection of conflicts that in turn suggest relaxations of the requirements. In order for the tool to be useful to a team of design engineers it is necessary to be able to suggest, in order of preference, relaxations that would yield a design solution. To support this, the designers provide a conditional preference representation.

Our work in this area leverages previous work in our lab on design optimization, combinatorial optimization, and mixed logic linear programming. We have previously developed an optimization solver called OPSAT, which we have used to optimize hybrid discrete/continuous designs. The focus of our work in this area for this project was integration of RMPL with OPSAT, to provide a more convenient method for specifying problems.

## Language Support

We have developed two back-ends for the RMPL language that produce output compatible with the Matlab reachSet analysis code and the other that produces models in the OPSAT solver format for use in design space exploration. The RMPL backend for OPSAT is a first step towards the Design Space Analysis capability by bringing RMPL modeling to the OPSAT solver and explanation capabilities.

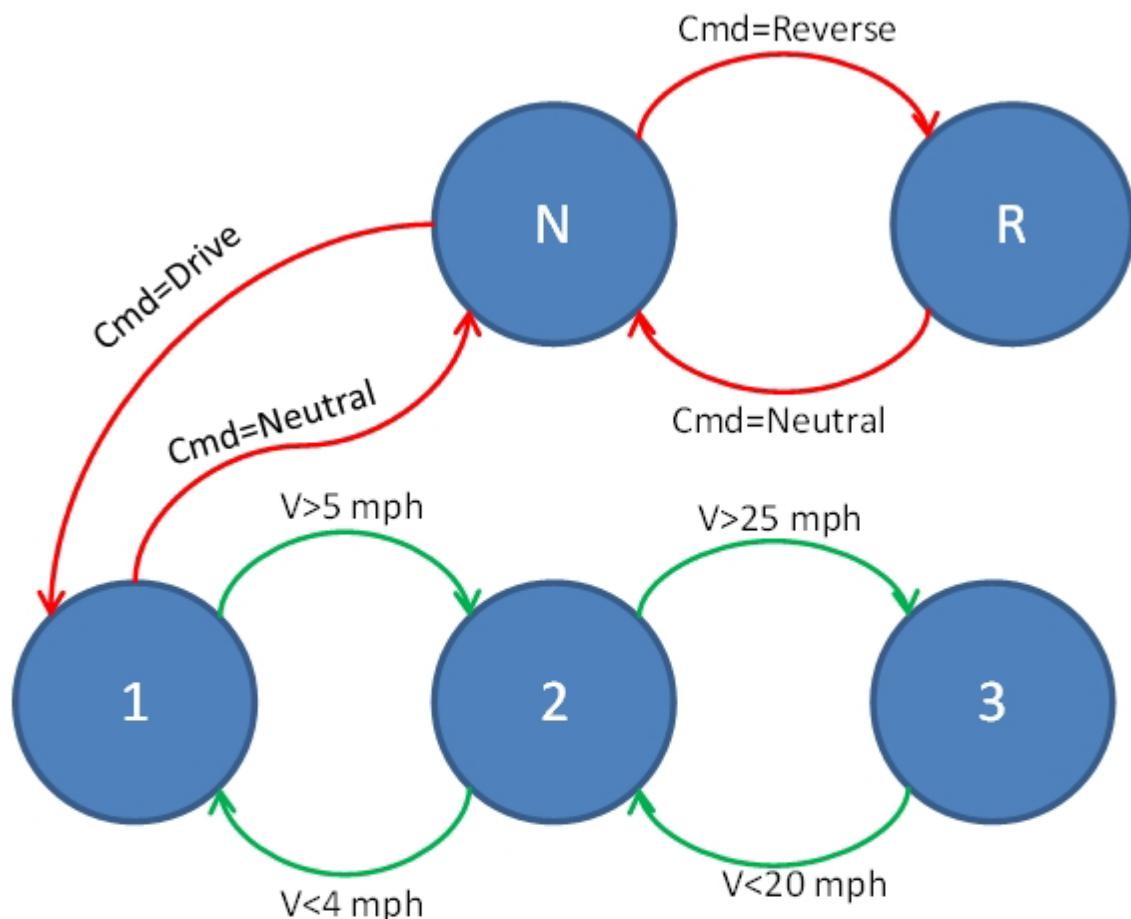
The following sections provide more details on our work in all of these areas.

### 7.10.2 Motivating Examples

Before going into the details of the verification system, we introduce a set of motivating examples, some of which will be used later to illustrate concepts.

### 7.10.2.1 Automatic Transmission Vehicle

Figure 7.10-1 shows a mode transition diagram for a vehicle with an automatic transmission. The modes correspond to the gear that the vehicle is in. Transitions between modes are achieved through a combination of input commands, and satisfaction of guard conditions. In this case, transitions between modes 1, N, and R are determined solely by input commands. Transitions between modes 1, 2, and 3 are determined by the guard conditions on vehicle velocity,  $v$ .



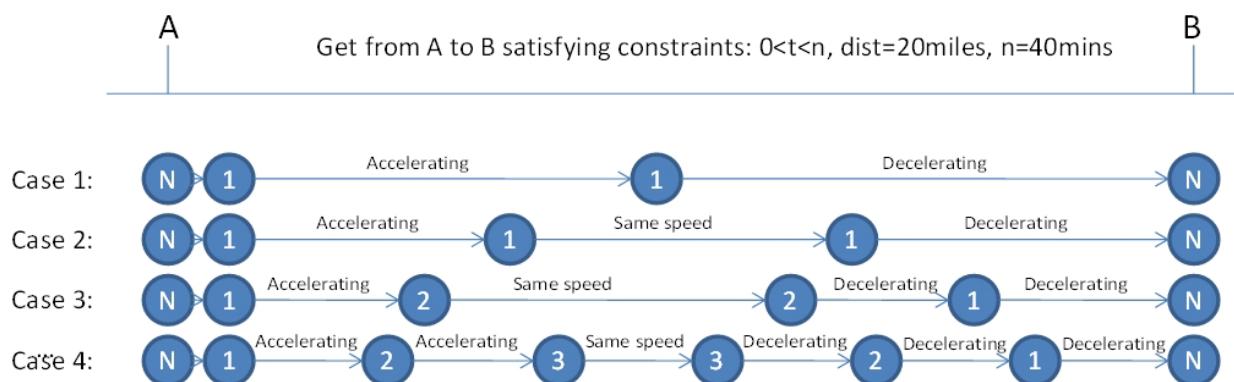
Modes 1, 2, 3, and R have (different) dynamics equations governing velocity( $V$ ) acceleration( $A$ ) Friction(etc), and throttle( $U$ ) (not shown). Modes 1 and 3 have one guarded transition out whereas mode 2 has two.



Figure 7.10-1. Mode transition diagram for vehicle with automatic transmission.

For example, suppose that the vehicle is in mode 2 (2nd gear). Transition to mode 3 occurs when  $v > 25$ , and transition to mode 1 occurs when  $v < 4$ . Thus, allowable speeds in mode 2 range from 4 to 25. The evolution of position and velocity state in mode 2 is determined by the dynamics associated with the mode, and by limits on the acceleration control input. These constraints impose limits on how a guard can be achieved, and they implicitly define the valid state trajectories for achieving the guard. As we will see, reach set analysis techniques can be used to determine valid state trajectories that end in a guard state being satisfied. The result is an explicit representation of the valid state trajectories, called a *flow tube*. These flow tubes only have to be computed once, for the model, rather than for each of possibly many use cases, because they don't depend upon the user provided use case. These flow tubes compile out the limits of what the plant with its constraints is capable of doing (independent of any particular use case). Models like this can also incorporate probabilistic transitions and noise models.

In order to verify that a design will work, a set of use cases must be specified against which the design model will be tested. Figure 7.10-2 shows how a form of qualitative simulation can generate detailed use cases given a user specified high level case. We can enumerate these in an interesting way such as in order of simplicity (number of mode changes) and we can abandon a use case as soon as we know during its elaboration that it has no solutions.



**Figure 7.10-2. Use cases for automatic transmission example.**

At the top of Figure 7.10-2 is a requirement that specifies in just enough detail what the user is trying to achieve including constraints on time, speeds, etc. This is the high level use case. Many detailed use cases can be enumerated from the model, but this should be done automatically. In this example, out of the 4 cases enumerated only one is viable given the user requirements.

Case 1 uses a single gear in a saw tooth approach. Case 2 involves a single gear for a trapezoidal model. Case 3 uses two gears, and case 4 uses three gears. Case 1 is not feasible, based on the model constraints, because the top speed will be 5 mph (in gear 1), so the vehicle will not reach 20 miles in the allotted 40 minutes. Case 2 fails for the same reasons. Case 3 fails too because even at the maximum speed of 25 mph for the entire route we can only get a 16 2/3 miles in 40 minutes. Case 4 can work as long as we can get to third gear long enough and keep it there long enough to get our average speed up to 30 mph or higher. The temporal bounds of the modes and the bounds on inputs for the modes can be extracted from bounding computations on the composition of the flow tubes.

#### **7.10.2.2 Hill Climbing Vehicle**

We can extend the automatic transmission vehicle model to include dynamics associated with going up and down hills. Hybrid discrete and continuous model aspects for this are shown in Figure 7.10-3.

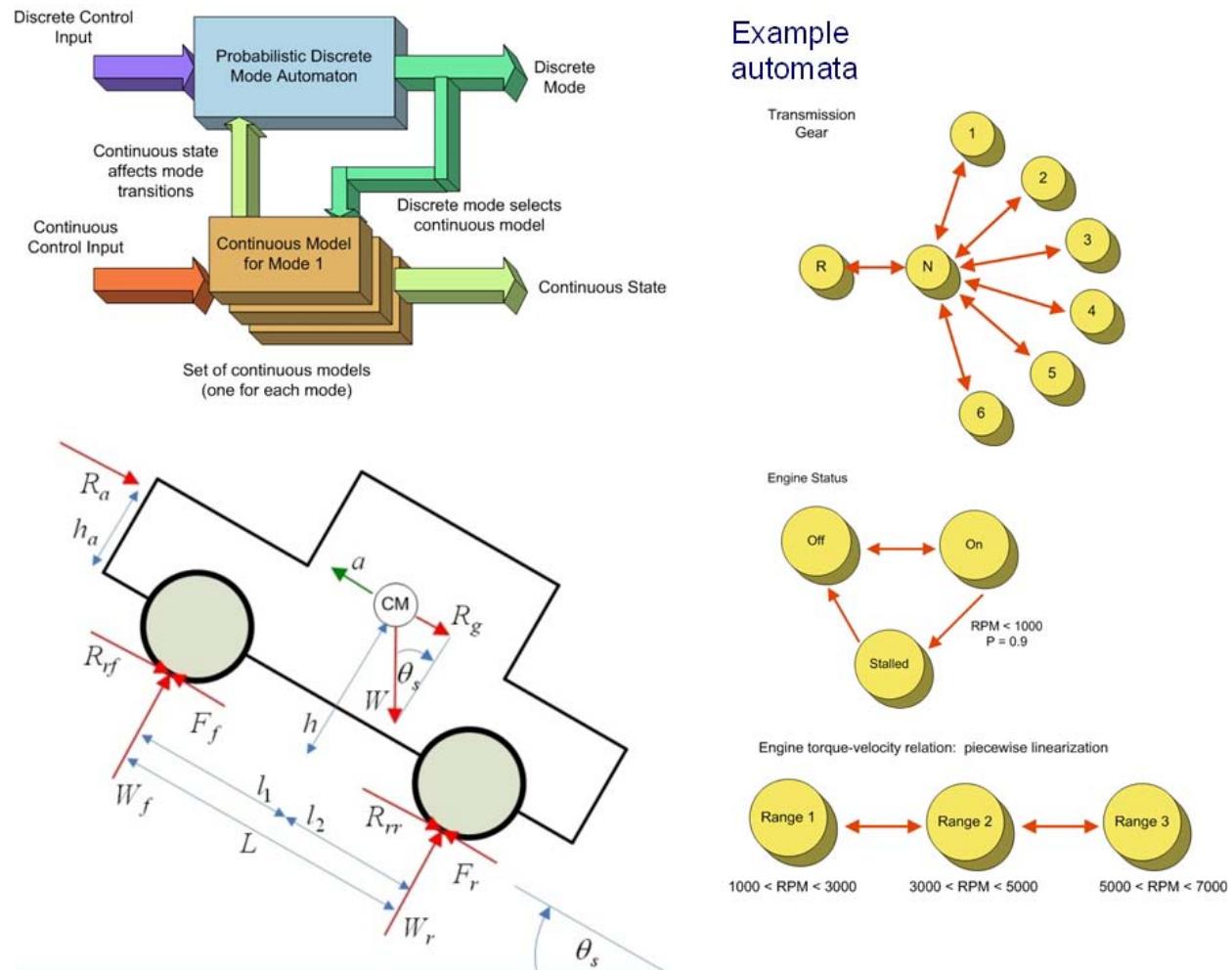
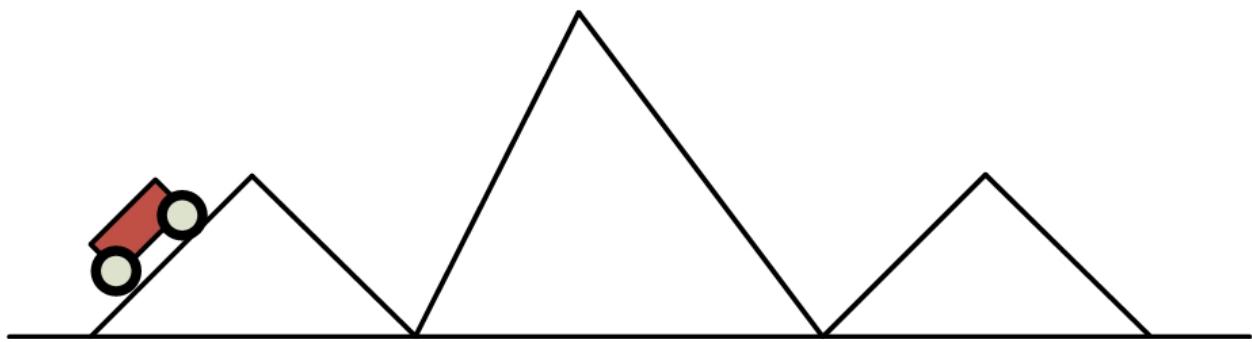


Figure 7.10-3. Hybrid model representing vehicle that goes up and down hills.

An example user-specified use case is shown in Figure 7.10-4. This use case specifies a set of intervals with different hill grades. Temporal constraints on overall completion time, or on times for individual intervals, could also be specified.

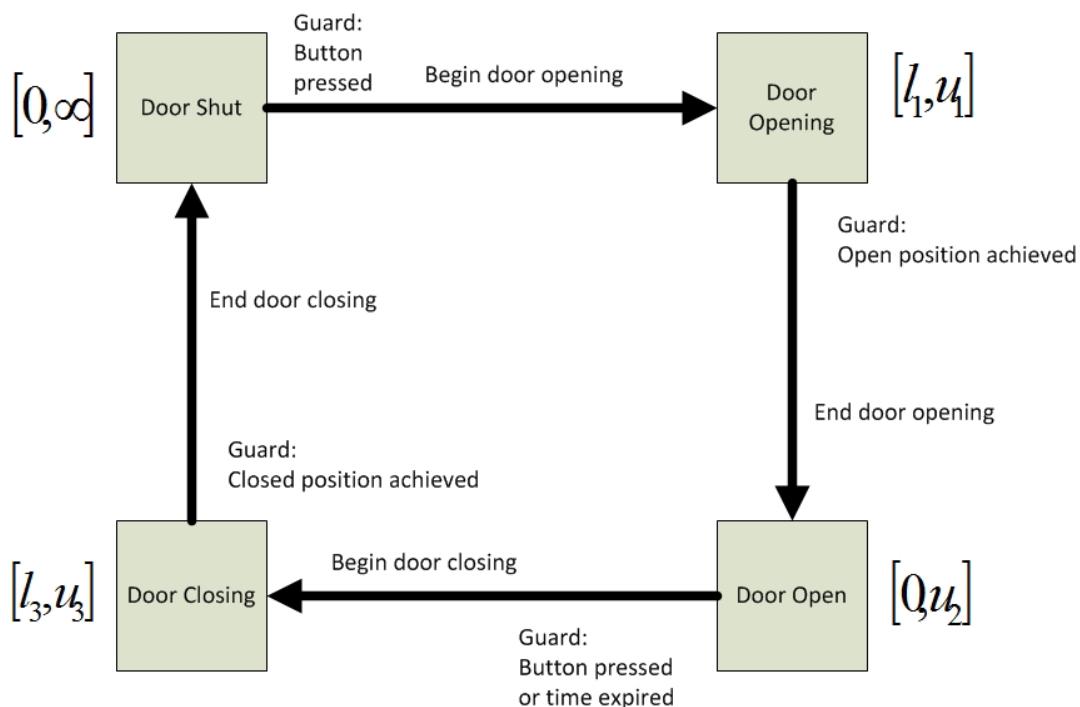


**Figure 7.10-4. Use case representing vehicle that goes up and down hills.**

### 7.10.2.3 Elevator

Consider the operation of an elevator door. This can be modeled using four discrete modes: door shut, door open, door shutting, door opening, as shown in Figure 7.10-5.

Blocks represent modes, which are analogous to activities in a QSP.  
Arrows represent transition events, which are analogous to events in a QSP.



Temporal limits  $l_1, u_1, l_3, u_3$  are based on velocity limits, and are computed from these limits using flow tubes. The temporal limit  $u_2$  is the maximum time the door is allowed to remain open.

**Figure 7.10-5. Simple elevator door example with upper temporal bound on Door Open mode.**

In general, there are interesting interactions between actuation limits, plant dynamics, and temporal constraints. Consider the door open mode. For simplicity, we assume a first-order model, where the state,  $x$ , is door position, the input,  $v$ , is door velocity, and there are constraints on the input (there is a minimum and maximum velocity).

Most real elevator doors close after a particular, specified duration. This can be accomplished by adding a temporal upper bound,  $u_2$ , to the mode specification for the Door Open mode. This is a constraint imposed by the designer, which is combined with constraints imposed by the plant.

#### **7.10.2.4 Infantry Fighting Vehicle Ramp**

The Bradley Infantry Fighting Vehicle has a door at the back that opens to allow entry and exit of troops. When the door opens, it folds down, forming a ramp down which the soldiers can run. A simple, linear equation of motion for the ramp is given by the following torque balance equation:

$$I_R \ddot{\Theta} = -g m_R r_{cg} + \tau_R \quad (1)$$

where  $I_R$  is the (scalar) inertia of the ramp,  $\Theta$  is the ramp angle,  $g$  is gravitational acceleration,  $m_R$  is the mass of the ramp,  $r_{cg}$  is the center of mass of the ramp, and  $\tau_R$  is the actuator torque exerted on the ramp about its hinge. A key assumption here, made to make the system linear, is that  $\Theta$  will be small. If this is not true, then a more complex model, with piecewise linearization and more QSP activities should be used (see subsequent sections). Force disturbance terms, due to loading from footsteps, can be added to the above formulation.

A use case for opening the ramp would consist of three activities. The first corresponds to acceleration of the ramp angular velocity to its maximum value. The goal region for this activity should be some range around the target velocity, and possibly, also some goal region for angular position. The second activity corresponds to movement at a steady angular velocity. A goal region about the reference angular velocity could also be included. The third activity corresponds to deceleration of the ramp angular velocity to zero. The goal region for this activity should be some (small) range about 0 for angular velocity, and some (small) range about the final target position.

#### **7.10.3 Verification Process Overview**

The MERS MIT component of the DARPA META project involves development of a verification capability for the requirements and preliminary designs of electro-mechanical systems. Current approaches to this type of verification involve the development of custom spreadsheets. Thus, the goal of this project is to develop a superior capability that is focused on requirements and preliminary design verification, but not on verification of detailed designs. This focus is important because requirements analysis phases of large projects typically consume a significant portion of the time and resources. Therefore, capabilities that improve the requirements analysis process and make it more efficient have high impact. Furthermore, mistakes made during requirements analysis (for example, not recognizing that the requirements pose an infeasible problem) result in expensive wasted design and development time, until the mistake is recognized and addressed.

A key requirement for this capability is that it be able to verify designs based on the physical limits of the electro-mechanical design, without requiring the user to also design and develop a control policy. This rules out the use of a monte-carlo forward simulation methods, since these require both a plant model and a control policy to run a simulation. For this reason, we use reach set analysis to compute families of state trajectories that the electro-mechanical design can achieve, based on a plant model alone. These are then tested against use cases representing the requirements.

In order to verify a design, we use the plant model, and associated reach set analysis, as a basis for *Qualitative Simulations* [WK91]. Each Qualitative Simulation produces a trajectory of *Qualitative States*, which are an abstraction of the full quantitative state trajectory, but which nevertheless have sufficient information content to allow for checking whether a use case requirement is satisfied. This approach supports fast verification of a design with respect to a potentially large and diverse set of use cases.

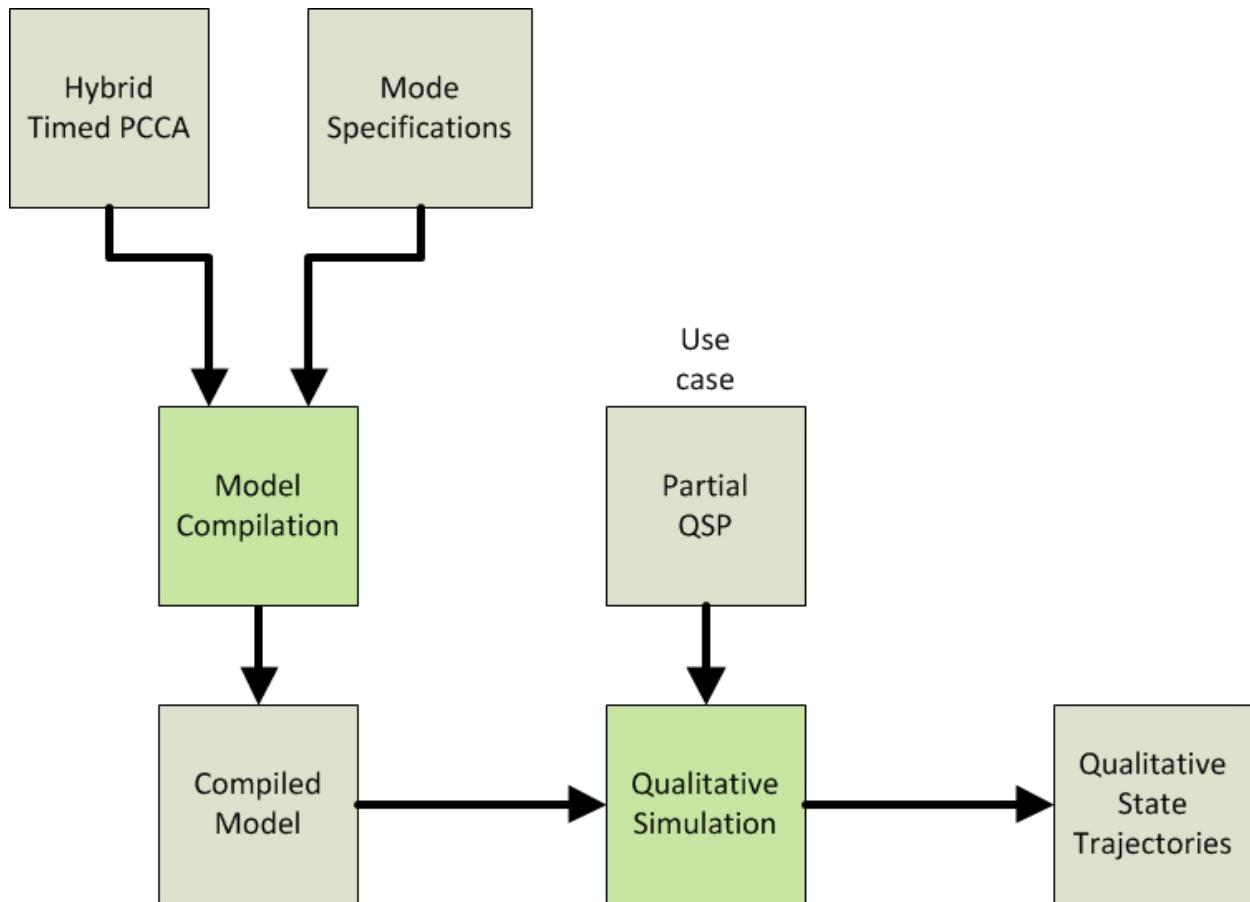
The following discussion first gives an overview of the verification process. After this, we describe the design of a Hybrid Model, based on *Probabilistic Concurrent Constraint Automata* (PCCAs) [MWI05], for use in verification applications. Key distinguishing features of this model are that it incorporates concepts of temporal flexibility into transitions, and that it uses flow tubes to represent the relation between state, input, and temporal constraints. Note that the focus here is on a model. The model information is used by the qualitative simulation to generate trajectories consistent with the use case requirements.

After definition of the plant model, we describe use case representation, as a partially specified *Qualitative State Plan* (QSP) [HW06]. Subsequently, the process of Qualitative Simulation based on the plant model, and satisfying the use cases, is described.

### 7.10.3.1 Use of Hybrid Model in Verification Process

Figure 7.10-6 shows the data structures and operations of the verification process. There are three inputs: 1) the Hybrid Timed PCCA (Hybrid Model), 2) the Mode Specification, and 3) the Partial QSP. There is one output: a set of Qualitative State Trajectories.

The Hybrid Model represents the capabilities and limitations of the *plant*. The plant is the electro-mechanical device being controlled. Limitations may include temporal constraints, and also dynamic constraints on state evolution due to saturation limits on control inputs like velocity or acceleration. The Mode Specification contains additional temporal and state space constraints that are not inherent in the plant, but rather, are imposed by the designer on the model. These specifications are general to the model; they are not specific to particular use cases. The Partial QSP (Partial Qualitative State Plan) represents a use case specified by the designer. It must include at least a specification of the initial and goal qualitative states for valid Qualitative State Trajectories. It may also include temporal and state space constraints, as well as required partial trajectories.



**Figure 7.10-6. Verification Process**

The verification process consists of two distinct operations: model compilation, and qualitative simulation. The model compilation process combines the constraints from the HTPCCA and the Mode Specification, and generates qualitative abstractions of the quantitative dynamics information, resulting in a *compiled model*. These qualitative abstractions are based on reach set analysis, which is used to construct *flow tubes* [HW06] representing families of valid trajectories. The compilation process also combines and compiles temporal constraints, so that the compiled model becomes an efficient basis for qualitative simulation.

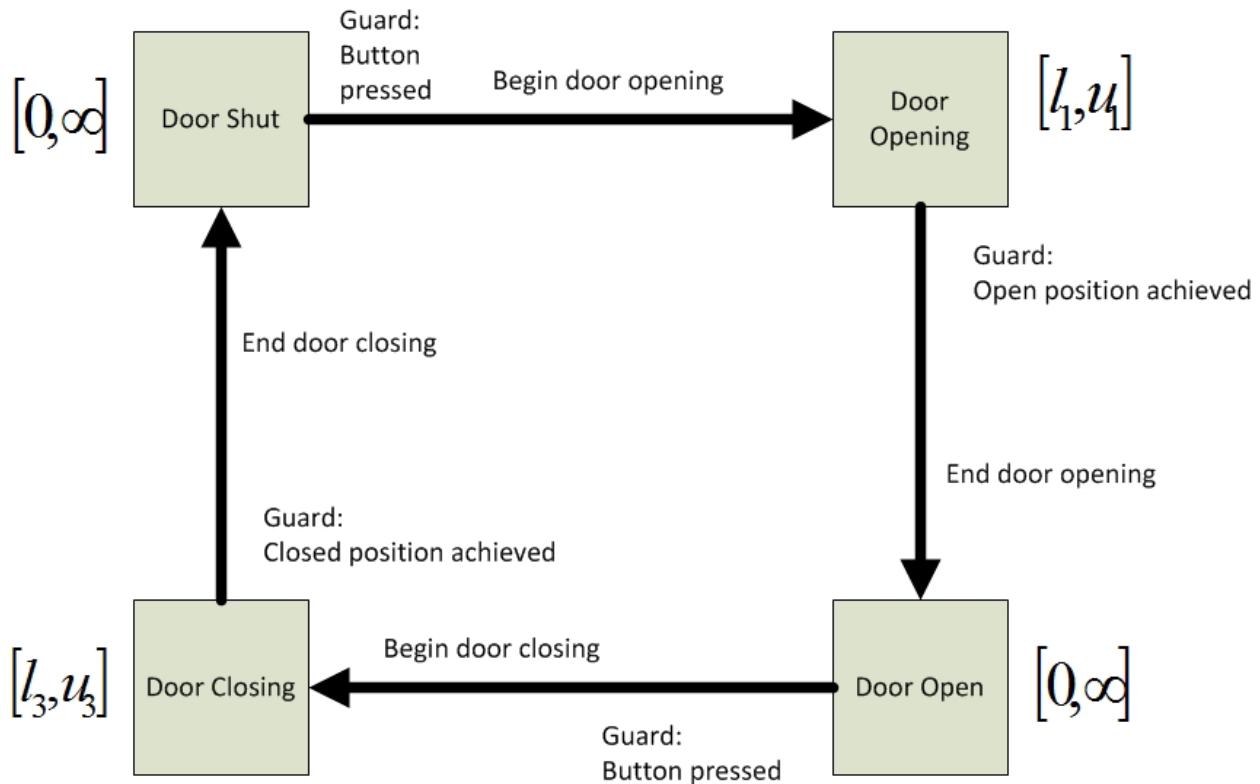
The qualitative simulation uses the compiled model to generate trajectories. It performs an efficient search to find trajectories that satisfy the Partial QSP use case requirements, and to quickly prune out ones that do not.

The next section presents a simple example of a Hybrid Model, which will be used throughout the discussion. Subsequent sections describe the data structures and operations of the verification process in more detail.

#### 7.10.4 Simple Example of Hybrid Model

Consider the operation of an elevator door. This can be modeled using four discrete modes: door shut, door open, door shutting, door opening, as shown in Figure 7.10-7.

Blocks represent modes, which are analogous to activities in a QSP.  
 Arrows represent transition events, which are analogous to events in a QSP.



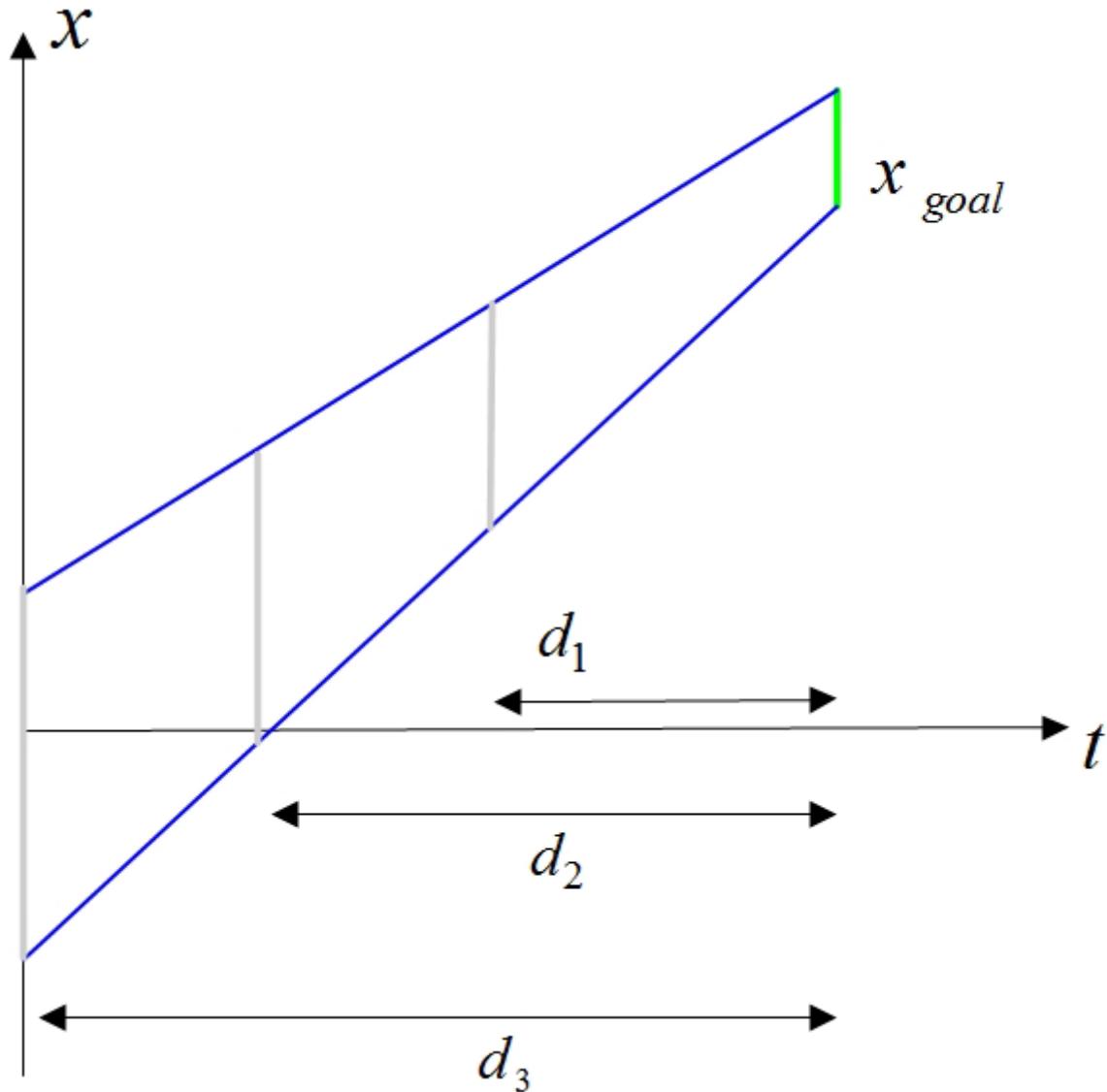
Transitions to door stuck states are not modeled here, but they could easily be added. Additionally, it would be interesting to model disturbances of various kinds to door opening and closing.

Temporal limits  $l_1, u_1, l_3, u_3$  are based on velocity limits, and are computed from these limits using flow tubes.

**Figure 7.10-7. Simple Elevator Door Example**

In general, there are interesting interactions between actuation limits, plant dynamics, and temporal constraints. Consider the door open mode in the figure. For simplicity, we assume a first-order model, where the state,  $x$ , is door position, the input,  $v$ , is door velocity, and there are constraints on the input (there is a minimum and maximum velocity).

A flow tube for this mode is shown in Figure 7.10-8.



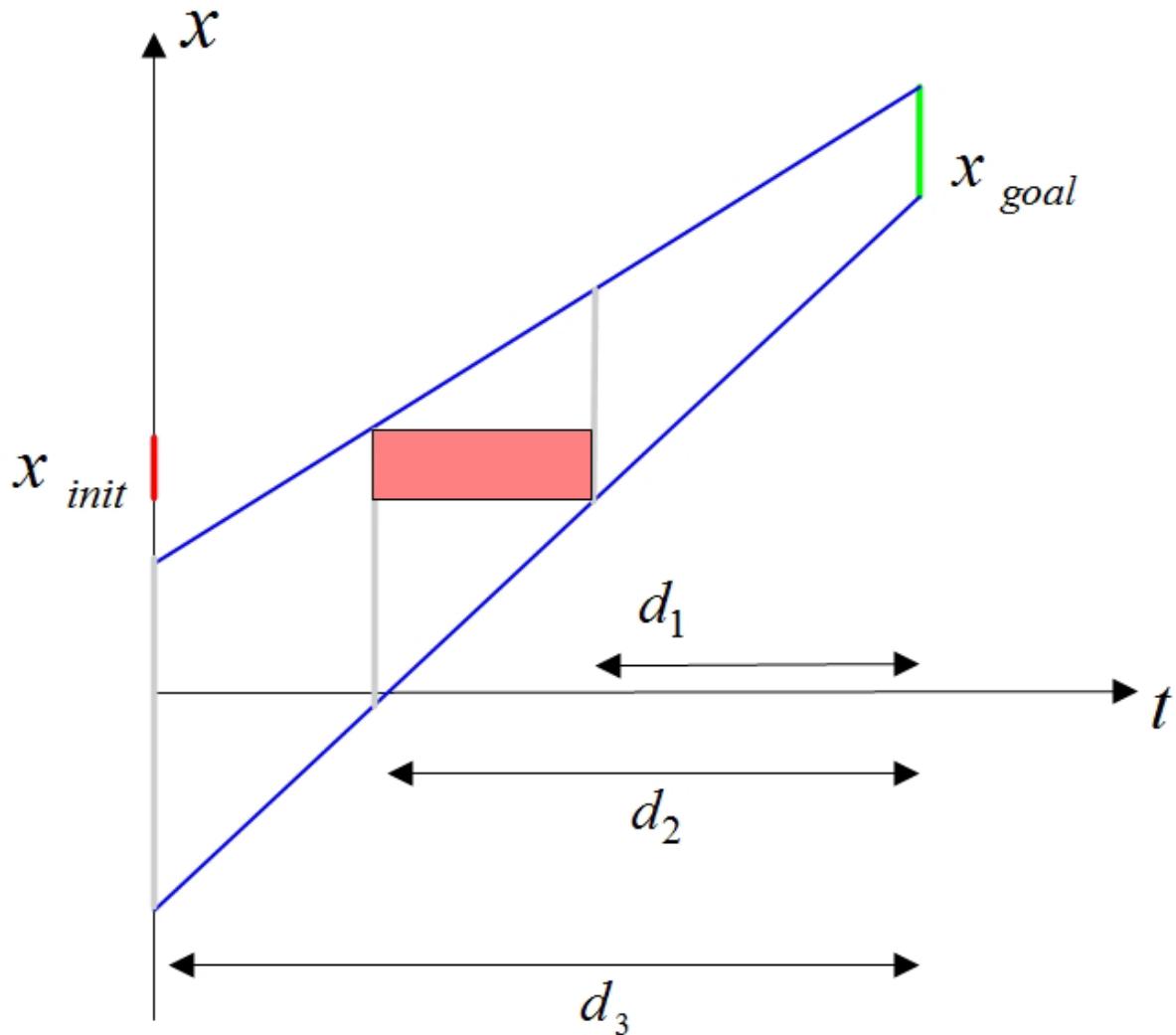
**Figure 7.10-8. Flow tube for door open.**

Cross sections for three durations,  $d_1$ ,  $d_2$ , and  $d_3$ , are shown in gray. The green region to the right is the goal region; the small range of positions for which the elevator door is considered open. Because the dynamics are simple, the flow tube can be easily computed analytically:

$$\begin{aligned} x_{\max}(d) &= \max(x_{goal}) - v_{\min}d \\ x_{\min}(d) &= \min(x_{goal}) - v_{\max}d \end{aligned} \quad (2)$$

Suppose that we now impose an additional constraint: that the mode must begin with the door in a closed position. Figure 7.10-9 depicts this situation. The red line on the  $x$  axis shows  $x_{init}$ , the small range of positions for which the elevator door is considered closed. The pink rectangular region shows the valid initial states and durations for this mode. Note that a

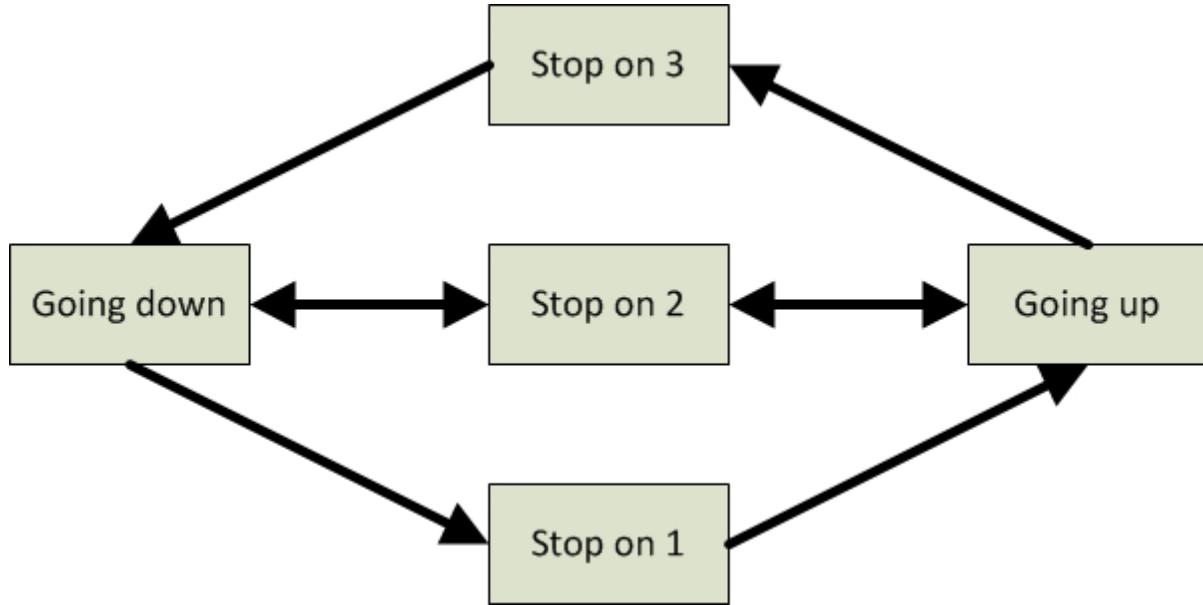
duration less than  $d_1$  is not possible, because the initial region does not fully overlap a cross section with a shorter duration. This is because there is a maximum permitted velocity. Note also that a duration greater than  $d_2$  is not possible, for the same reason. This is because there is a minimum permitted velocity. Thus,  $d_3$  is not a possible duration.



**Figure 7.10-9. Flow tube for door open, with door closed initial constraint.**

This simple example shows how the interaction of actuation limits, plant dynamics, and initial and goal region requirements can result in temporal constraints. The interactions are more complex with higher order systems, but the basic principle is the same. (Think about transition sequence fragments in the automaton, and how these can be used to impose constraints on QSPs.)

Consider, next, another elevator related example, shown in Figure 7.10-10. This shows a hybrid model representing movement of the elevator between three floors. In this model, the modes going up and going down have multiple exit transitions. Flow tubes are used to derive temporal constraints that are different for the different exit transitions.



**Figure 7.10-10. Model of elevator movement between three floors.**

We assume, again, simple dynamics. In this case, the state variable is vertical position of the elevator, and the input is the vertical speed. The going up mode has an exit transition at the second floor, and one at the third floor. The guard conditions for these exits include goal regions for vertical position. Given that the initial region for the going up mode is the first floor, then the dynamics impose different temporal constraints on the second and third floor exit transitions. This is because it takes longer to go from the first to the third floor, than from the first to the second floor.

### 7.10.5 Formal Specification of Hybrid Timed PCCA

#### 7.10.5.1 PCCA

A *Hybrid Timed Probabilistic Concurrent Constraint Automaton* is based on a *Probabilistic Concurrent Constraint Automaton* (PCCA). We define the latter first, and then extend this to achieve a complete specification of the former.

A PCCA [MWI05] consists of a set of automata, where an automaton for component "a" is defined by the tuple:

$$A_a = \langle \prod_a, M_a, T_a, P_{T_a}, P_{\tau_a} \rangle.$$

$\prod_a = \prod_a^m \cup \prod_a^r$  is a finite set of discrete variables for component "a", where each variable  $\pi_a \in \prod_a$  ranges over a finite domain  $D(\pi_a)$ .  $\prod_a^m$  is a singleton set containing *mode* variable  $x_a$ , whose domain  $D(x_a)$ , is the finite set of discrete modes in  $A_a$ . *Attribute* variables  $\prod_a^r$  include inputs, outputs, and any *dependent* variables used to specify behavior.

$M_a$  maps each mode assignment  $(x_a = v_a)$  to a finite domain constraint  $c_a(x_a = v_a)$ .

$T_a$  is a set of transition functions. Given a current mode assignment  $(x_a = v_a)$  and guard  $g_a$ , each transition function  $\tau_a(x_a = v_a, g_a)$  specifies a target mode assignment that the automaton could transition into in the next time step.

$P_{T_a}$  is a transition probability distribution. For each mode variable assignment and guard, there is a probability distribution across all transitions into target modes defined by the set of transition functions.

An entire system (plant) is modeled by composing such automata, resulting in a PCCA model.

#### 7.10.5.2 Extension to Hybrid

The extension of a PCCA automaton to be hybrid is accomplished through the introduction of continuous variables and constraints. The constraints can include both numerical equalities and inequalities. The constraints can be used to express differential, as well as algebraic relations. We assume that the constraints are all linear. Further, for any particular mode assignment, we assume that the numerical constraints are convex.

In order to make a PCCA automaton hybrid, we associate with each mode assignment,  $(x_a = v_a)$ , a set of continuous variables,  $\mathbf{c}$ . These include state variables,  $\mathbf{x}$ , input variables,  $\mathbf{u}$ , and output variables,  $\mathbf{y}$ . Additionally, we associate with the mode assignment a set of *initial* constraints, a set of *goal* constraints, and a set of *operating* constraints. The initial constraints represent guards, over  $\mathbf{c}$ , that must be satisfied in order to make a transition into the associated mode. Thus, the initial constraints are *entry* constraints for the mode. The goal constraints represent guards, over  $\mathbf{c}$ , that must be satisfied in order to make a transition out of the associated mode into a new mode. Thus, the goal constraints are *exit* constraints for the mode; they become part of the transition function,  $\tau_a(x_a = v_a, g_a)$ , defined for PCCAs, where they augment the discrete domain guard conditions  $g_a$ . The operating constraints represent requirements, over  $\mathbf{c}$ , that must be satisfied in order for the associated mode to be marked. If the operating constraints are not satisfied, the automaton exits out of the associated mode immediately, usually to an error mode.

The initial and goal constraints are linear algebraic equality and inequality constraints of the form

$$\begin{aligned} \mathbf{f}(\mathbf{c}) &= \mathbf{c}_1 \\ \mathbf{g}(\mathbf{c}) &\leq \mathbf{c}_2 \end{aligned} \tag{3}$$

where  $\mathbf{c}_1$  and  $\mathbf{c}_2$  are vectors of constants. It is assumed that the initial and goal constraint sets are each convex.

The operating constraints are also of the form of (4), and are convex. In addition, operating constraints can include dynamic constraints in the form of linear difference equation constraints. Specifically, these constraints are expressed as

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \quad \mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \tag{4}$$

where  $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$  are matrices of appropriate dimensions.

Difference engine constraints represent the dynamic state evolution over time, and can be used to compute constraints on this state evolution. For example, suppose that the state vector,  $\mathbf{x}$ , consists of position and speed of a car moving on a straight line. Suppose, further, that the control input vector,  $\mathbf{u}$ , consists of the acceleration, which has limits. The limit on the input, combined with the dynamic constraints, results in limits on the evolution of the position and velocity state. If the maximum acceleration is  $1\text{m/s}^2$ , then the car can't go from 0 to  $10\text{m/s}$  in one second. It also can't go from position  $0\text{m}$ , velocity  $0\text{m/s}$  to position  $10\text{m}$  in one second.

This also shows the relationship between dynamic constraints on state evolution, and temporal constraints on duration of a mode. In the above example, if a guard condition requires the state to be in a certain region (at position  $10\text{m}$ , for example), then the dynamic constraints may impose limits on how long it takes to get there.

### 7.10.5.3 Extension to Timed

As discussed in the previous subsection, dynamic constraints may result in implicit temporal constraints on mode duration. We now extend the model to also allow for explicit temporal constraints, provided by the user. For example, the upper temporal bound  $u_2$  in Figure 7 is a specified bound on the maximum time the elevator door can remain open before it starts to close.

The extension of a PCCA automaton to include explicit temporal constraints is accomplished through the introduction of explicit lower and upper bounds on duration. Thus, we associate with each mode assignment,  $(x_a = v_a)$ , a temporal bound  $[l, u]$ , where  $l$  specifies the minimum feasible duration for the mode, and  $u$  specifies the maximum.

As will be discussed subsequently, the interaction of implicit and explicit temporal bounds is of great interest in verification, and is an important contribution of this work.

### 7.10.6 Mode Specifications

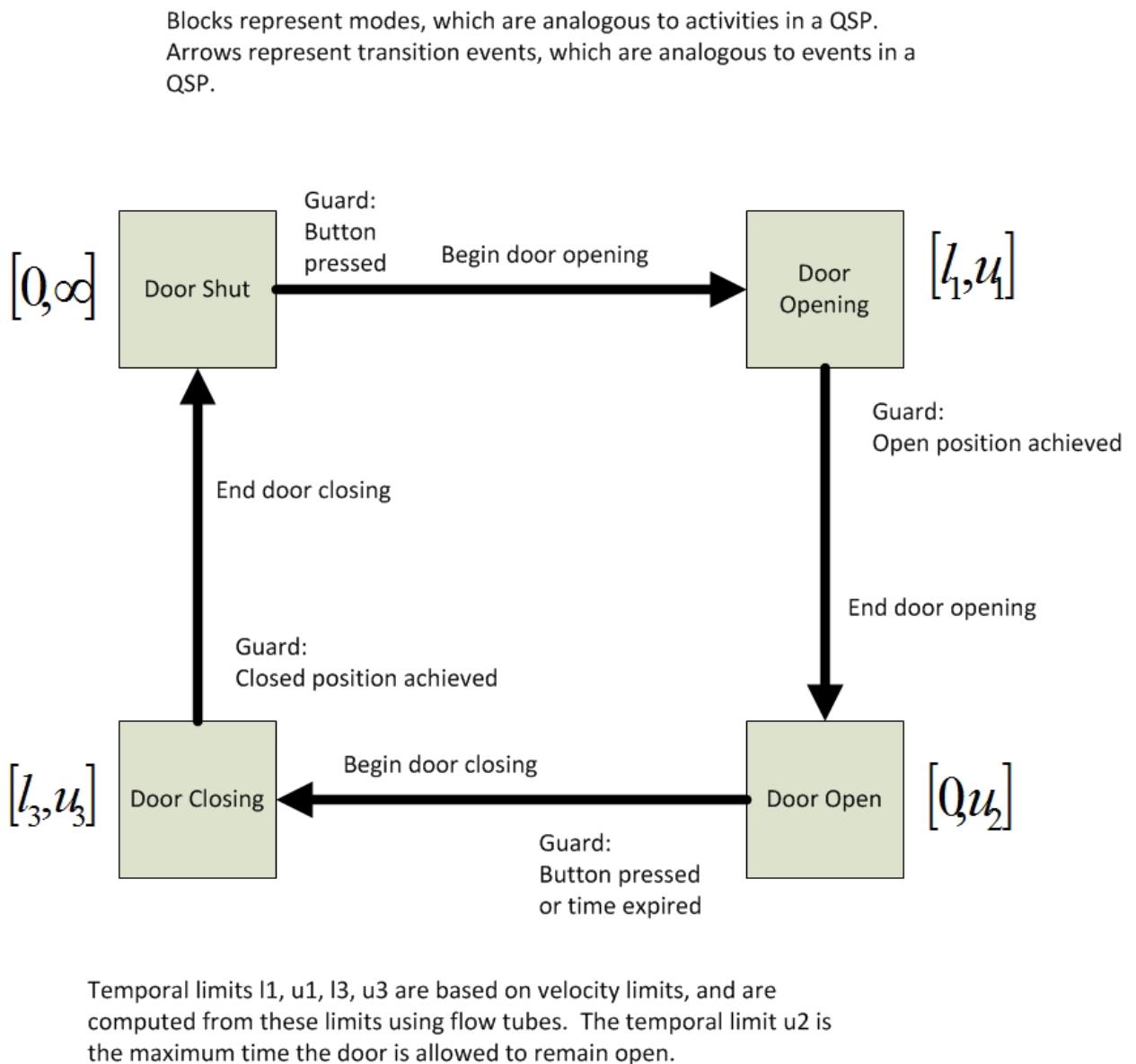
Mode specifications are temporal and state space constraints imposed by the designer for modes in a Hybrid Timed PCCA. They represent additional constraints, beyond the ones inherent in the plant being modeled. This results in a tightening of constraints in the combined system. The temporal and state space constraints in the Mode Specifications have the same form as corresponding constraints in the Hybrid Timed PCCA.

Consider the simple elevator door example in Fig. 7. As shown in the Figure, the door can remain open for an infinite duration (according to the plant model). Most real elevator doors close after a particular, specified duration. This can be accomplished by adding a temporal upper bound,  $u_2$ , to the mode specification for the Door Open mode. This is a constraint imposed by the designer. When combined with the plant model, the following automaton representing both plant model and mode specifications can be inferred.

### 7.10.7 Verification through Qualitative Simulation using Hybrid Timed PCCA

A Hybrid Timed PCCA can be used to generate a qualitative simulation of the possible state evolution trajectories. In this case, the inputs to the qualitative simulator component are the compiled Hybrid Timed PCCA, and a *partial QSP* representing use case requirements, as shown

in Figure 7.10-11. At a minimum, the partial QSP must specify initial and goal qualitative states, but it may also specify intermediate qualitative states and activities that lead from the initial to goal states. The output of such a simulation is a sequence of mode assignments (qualitative states), with associated duration ranges for each mode. An additional byproduct of the forward simulation is a probability indicating the likelihood of the trajectory.



**Figure 7.10-11. Simple elevator door example with upper temporal bound on Door Open mode.**

A key advantage of this qualitative simulation approach, versus a full forward simulation with full continuous dynamics, is that it caches continuous dynamics information in flow tubes, so that this doesn't have to be repeated. This flow tube abstraction is not sufficient to compute individual trajectories the way a full continuous dynamics simulation would. However, the flow tube information does define the boundaries of valid trajectory sets, which is sufficient for

verification. A second key advantage of this qualitative simulation approach versus full forward simulations is that the latter require the user to provide a control policy, while the former (our approach) does not. This is significant, especially during the requirements and early design phases of an electro-mechanical system. The analysts and designers working during these phases are primarily concerned with the capabilities and limits of the electro-mechanical design, and do not want to have to worry about a control policy in addition to this. Our approach allows such users to specify limits of the design, such as maximum speed, maximum actuation force, etc., and to base verification analysis on these limits alone, without having to also develop a control policy.

#### **7.10.7.1 Qualitative Simulation Problem Definition**

In order to define the qualitative simulation problem more formally, consider again Figure 7-10-6, which shows the data structures and operations of the verification process. There are three inputs: 1) the Hybrid Timed PCCA (Hybrid Model), 2) the Mode Specification, and 3) the Partial QSP. There is one output: a set of Qualitative State Trajectories. Each such trajectory is represented as a sequence of HTPCCA modes, with associated durations.

It is important to note two important aspects of this output. First, unlike typical qualitative simulations, which are simply mode sequences, this output incorporates metric time, represented as the associated durations. This is crucial for meaningful verification. Second, the modes in the sequence have associated quantitative information: the compiled flow tubes in the HTPCCA modes, which represent an abstraction of the system dynamics. Inclusion of this (abstracted) quantitative information is also crucial for meaningful verification.

The HTPCCA forms a graph where the nodes are modes, and the edges are transitions. The qualitative simulation problem, and the verification process itself, is defined as one of searching this graph to produce a mode sequence (Qualitative State Trajectory) that satisfies the Partial QSP. The qualitative simulation is accomplished by deciding mode transitions, and duration in each mode. Thus, qualitative simulation amounts to assignments to two kinds of discrete decision variables (mode, and duration), subject to logical (discrete) and numeric (continuous) constraints. Note that a discrete time approach is used in modeling the continuous dynamics. This is consistent with commonly used standard controller synthesis and verification analysis approaches.

Before describing the algorithm that performs the verification process, it is useful to gain understanding through a series of simple examples that illustrate the key concepts.

After these examples, we describe the algorithms of the verification process. As described previously in the overview section, the verification process has two main steps: model compilation, and qualitative simulation. We begin the algorithm description with a discussion of the model compilation process. This is followed by a discussion of the algorithms for qualitative simulation. As will be explained in more detail, the qualitative simulation uses a generate and test search, where a candidate mode sequence and duration schedule is generated, and then checked to determine whether all qualitative and temporal constraints are satisfied.

### 7.10.7.2 Qualitative Simulation Examples

#### 7.10.7.2.1 Qualitative Simulation Example 1

The first example uses a simple model and partial QSP, as shown in Figure 7.10-12. In this case, the use case partial QSP is actually fully specified. It represents going from Start to Finish within duration bounds  $\llbracket l, u \rrbracket$ .

The qualitative simulation process begins at a mode in the model that satisfies the initial mode requirements of the partial QSP. Recall that the partial QSP, even if it isn't fully specified, must specify the initial and final state. In this example, the initial mode for the partial QSP is "Start", and this matches the mode "A" in the model.

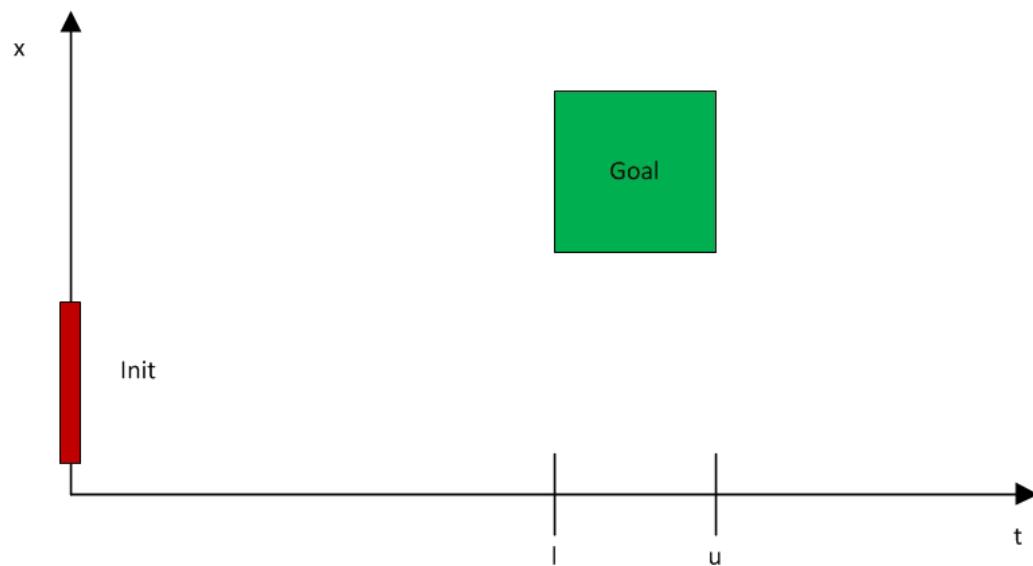
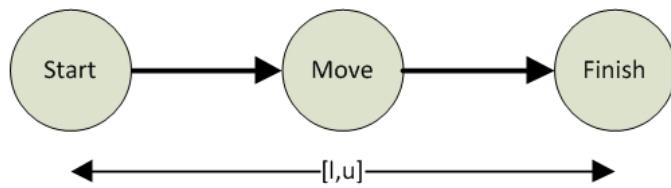
The search process investigates possible exit transitions out of the initial mode. For this example, there is one exit transition in the model: to the mode "Move". This matches the "Move" mode in the QSP.

To achieve a match, it is typically necessary to attempt to adjust the flow tube computed for the model so that it satisfies the requirements of the use case QSP. This is because the model provides a template description of dynamic behavior in each mode, which can be adapted during Qualitative Simulation to a wide range of possible use cases. This adjustment must be made in such a way that the inherent dynamic information in the flow tube is not altered.

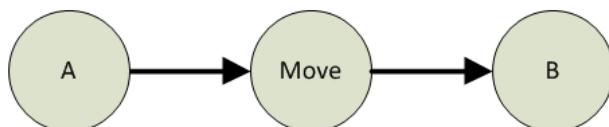
Flow tube adjustment requires *parameterized flow tubes*, which are discussed in more detail in Section 7.4.1. A simple adjustment, which can always be accomplished easily, is a shifting of a flow tube in position. This type of adjustment will be used extensively in the following examples.

Use of parameterized flow tubes in this way supports the key concept of computing flow tubes once for the model, and then re-using them extensively in the qualitative simulation to determine possible trajectories. This avoids needless re-computation of flow tubes, especially when there is a large set of complex use cases.

## Use Case 00 QSP



## Model 00



This model represents getting from point A to point B, subject to velocity constraints.

[show flow tube, combination with use case, shifting of parameters]

**Figure 7.10-12. Simple velocity limited system for getting from one position to another.**

For the model in this simple example, the flow tube representing the movement is computed based on the velocity limits. Problems with more complex dynamics, such as acceleration limited dynamics, require a more sophisticated flow tube computation, as will be discussed further in Section 7.3.1. The simple velocity-limited flow tube computation is used here for illustrative purposes, though it can also be usefully applied to a wide variety of problems.

Suppose that the nominal goal in the model is  $x_g = [x_{gmin}, x_{gmax}] = [-2, 2]$ , where  $x_g$  is the goal position. Suppose that the velocity limits are in the range  $v_{lim} = [v_{min}, v_{max}] = [1, 2]$ . The flow tube, as a function of duration back from the goal can be expressed as

$$\begin{aligned} x_{max} &= x_{gmax} - v_{min} d \\ x_{min} &= x_{gmin} - v_{max} d \end{aligned} \quad (5)$$

Here,  $d$  corresponds to duration. For a discrete time system,  $d$  is a multiple of the time increment,  $\delta t$ . Assuming a time increment of  $\delta t = 1$ , the flow tube cross sections for each  $d$  are as shown below.

$x_{max}$	$x_{min}$	$d$
2	-2	0
1	-4	1
0	-6	2
-1	-8	3
-2	-10	4
-3	-12	5
-4	-14	6
-5	-16	7
-6	-18	8

We assume in this example that the maximum duration specified in the model is 8.

Suppose, now, that the goal in the QSP is specified to be  $x_{gqsp} = [x_{gqspmin}, x_{gqspmax}] = [8, 12]$ .

Shifting the model flow tube by 10 causes the goal range in the shifted model flow tube to match the goal in the QSP:  $x_g + 10 = x_{gqsp}$ . Thus, the shifted flow tube cross sections are

$x_{max}$	$x_{min}$	$d$
12	8	0
11	6	1
10	4	2
9	2	3
8	0	4
7	-2	5
6	-4	6
5	-6	7
4	-8	8

This simple example shows how a flow tube in the model is adjusted to meet the requirements of the QSP. When this can be done, then the transition out of the mode in the model is valid, and the qualitative simulation search can proceed along that branch.

Once a transition has been established, the next step is to check whether state constraints cause a tightening of the duration range. In the model and shifted flow tubes, the duration range is  $[0,8]$ . Suppose now that the initial position range specified in the QSP is

$x_{initqsp} = [x_{initqspmin}, x_{initqspmax}] = [5,7]$ . Checking against the shifted flow tube cross sections, this means that durations of 2, 3, 4, and 5 are valid. A duration of 1 doesn't work if the initial position state is 5. A duration of 6 doesn't work if the initial position state is 7. Thus, the tightened duration range is  $[2,5]$ , which is less than the original duration range of  $[0,8]$ . Note that this tightening can result in a duration range that is empty. For example, if

$x_{initqsp} = [15,17]$ , then none of the shifted flow tube cross sections match, and the tightened duration range is empty. This implies that the transition actually can't be accomplished, and that the search branch should be pruned.

At this point, we have a qualitative simulation trajectory that matches the state requirements of the QSP. The final step is to check temporal constraints, to make sure that the tightened temporal constraints in the qualitative simulation trajectory are consistent with temporal constraints specified in the QSP. This is generally accomplished using STN analysis techniques (see thesis, Stedl). In this simple example, suppose that the QSP temporal constraint is  $[l,u] = [1,4]$ . Combining this with the tightened temporal constraint from the flow tube results in an overall temporal constraint of  $[2,4]$  for the "move" mode in the qualitative simulation. This is a further tightening, resulting from the user specified temporal constraint in the QSP. In this case, the duration is non-empty, so the qualitative simulation trajectory is valid. If the QSP temporal constraint is  $[l,u] = [6,8]$ , then the tightened duration range is empty, and the qualitative simulation trajectory is not valid.

In this example, the goal range of the model flow tube perfectly covered the goal requirement specified in the QSP. This will not always be the case. For example, if the goal range of the model flow tube were  $[-1,1]$  instead of  $[-2,2]$ , then the model flow tube cross sections are too small; they constrain the trajectory set too tightly. This may result in failure to cover the initial region constraint. If, on the other hand, the goal range of the model flow tube were  $[-3,3]$ , then the model flow tube cross sections would be too large; they do not sufficiently constrain the trajectory set. This may result in admission of infeasible trajectories.

We address this problem in two ways: adjustment, and relaxation. Just as the goal position was shifted in the previous example, it is often possible to efficiently adjust the model flow tube so that its goal range matches that of the QSP. This is always possible for position goal range, even with higher-order flow tubes.

In some cases, it is prohibitively expensive to adjust the model flow tube adequately. In this case, a relaxation of the problem is first solved to obtain a preliminary qualitative simulation trajectory. If this trajectory is feasible, further analysis is performed to solve the full (un-relaxed) problem. The relaxation is obtained by finding a model flow tube (possibly through efficient adjustment) that represents an "outer" approximation. In the previous example, this is the model flow tube with goal range  $[-3,3]$ , resulting in flow tube cross sections that are too large. The representation is therefore complete, but not sound; it will admit some infeasible trajectories (that is why it is a relaxation). If the qualitative simulation trajectory resulting from this relaxed flow tube is not feasible, then the corresponding search branch can be pruned and need not be investigated further. If the trajectory is feasible, further analysis is needed to confirm that the un-relaxed problem requirements are also satisfied by the trajectory. This analysis amounts to a full re-computation of the trajectory flow tubes, based on goal ranges that match the QSP requirement exactly. This full re-computation is what we originally did for META. The point is that the relaxation filter should prune out the majority of branches so that a full flow tube re-computation is only rarely necessary. Use of relaxations is a well known approach in the solution of complex problems. Our use of it in this case is an interesting and novel application of this technique.

This issue is complex, and is introduced in summary here first. See Appendix x for further details on flow tube adjustment through use of parameterized flow tubes. Further details of the relaxation approach will be presented subsequently.

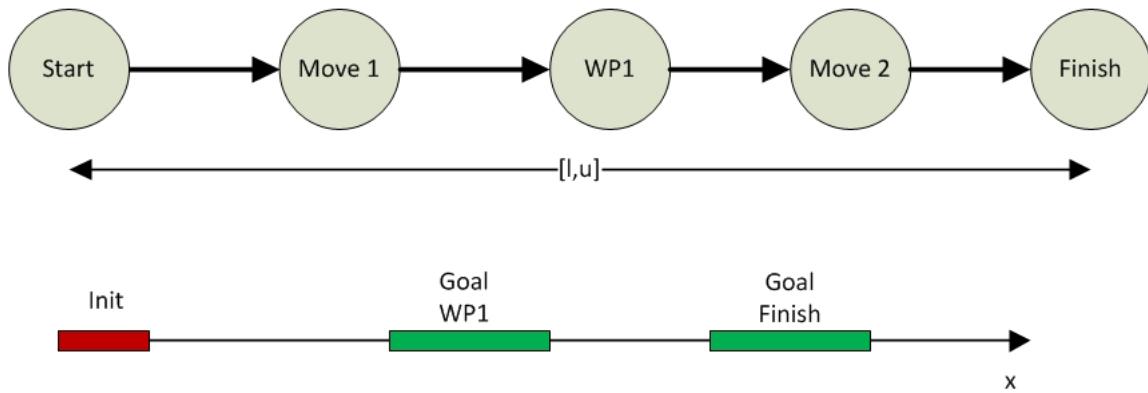
#### **7.10.7.2.2 Qualitative Simulation Example 2**

In the next example, shown in Figure 7.10-13, the QSP is slightly more complicated; an intermediate waypoint (WP1) has been added before the goal. Note, however, that the model stays the same. This is a key concept: many different QSP's, representing many different use cases, can share the same model, and thus, the flow tubes in the model. This allows for evaluating a design (as described by the model), for a large number of use case. This allows for the evaluation to be both thorough and time efficient.

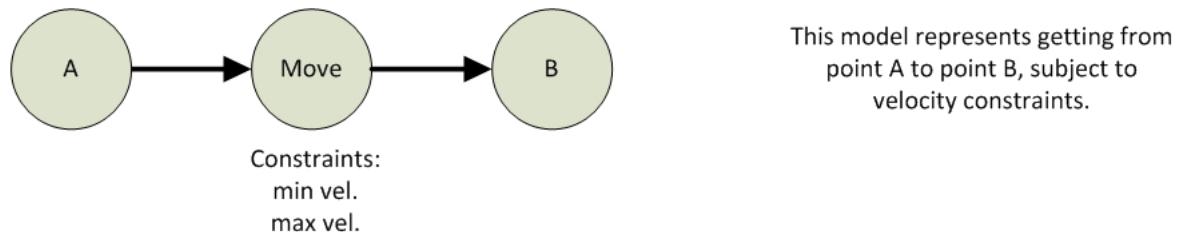
Suppose that the goal in the QSP for WP1 is specified to be  $x_{gqsp} = [8,12]$ , and the goal to finish is  $x_{gqsp} = [16,20]$ . Working backwards from the finish goal, the shifted flow tube for Move 2 has cross sections

$x_{max}$	$x_{min}$	$d$
20	16	0
19	14	1
18	12	2
17	10	3
16	8	4
15	6	5
14	4	6
13	2	7
12	0	8

Use Case 00b QSP



Model 00

**Figure 7.10-13.** Simple velocity limited system for getting from one position to another.

The goal for WP1 becomes the initial region for mode Move 2. Checking against the shifted flow tube cross sections, this means that durations in the range  $[4,8]$  are valid. The shifted flow tube for Move 1 is the same as the one for Move in the previous example. Thus, we have possible durations of  $[2,5]$  and  $[4,8]$  for Move 1 and Move 2 respectively. The possible duration range for the sequence is therefore  $[6,13]$ . Suppose that the QSP temporal constraint is  $[l,u]=[4,10]$ . The tightened temporal range is then  $[6,10]$ , which is non-empty, so the trajectory is feasible.

This example could be extended with many more waypoints, and the same approach, with the same model, would be used. This shows how flow tubes computed for the model are used repeatedly in evaluation of the use case.

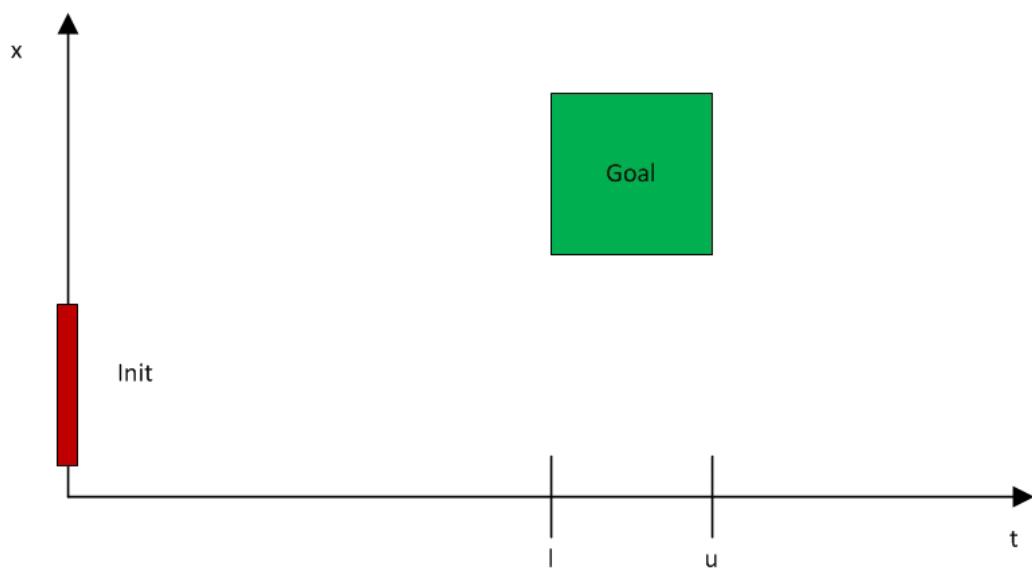
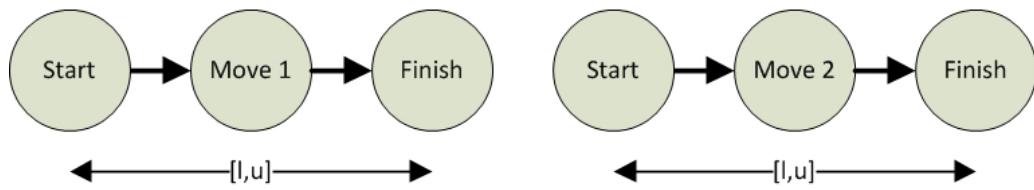
#### 7.10.7.2.3 Qualitative Simulation Example 3

In the next example, shown in Figure 7.10-14, the model provides two paths to the finish. The discrete mode variable "Gear" has value 1 in mode "Move 1", and has value 2 in mode "Move 2". These two modes have different velocity constraints (second gear has higher minimum and maximum velocities than first gear).

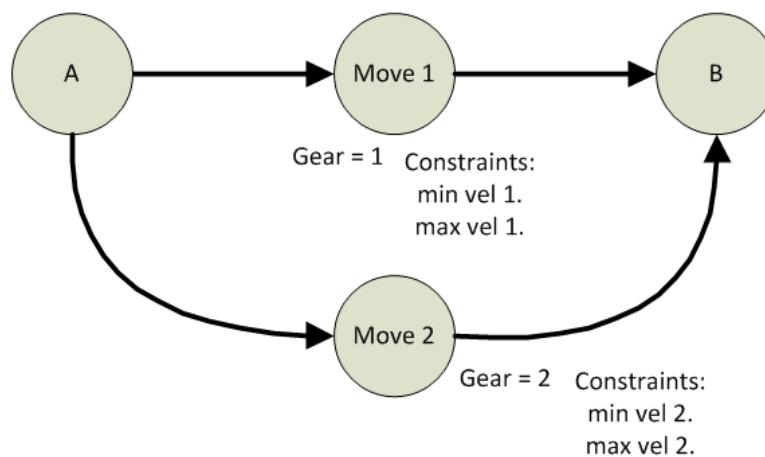
Two use case QSP's are shown. One specifies that the "Move 1" path should be used (indicating that the movement should be in first gear). The other specifies that the "Move 2" path should be used (indicating second gear). Thus, in this case, the discrete mode variable "Gear" is fully specified.

Suppose that the velocity limits are  $v_{lim} = [v_{min}, v_{max}] = [1,2]$  for "Move 1", and  $v_{lim} = [v_{min}, v_{max}] = [2,4]$ . The cross sections of the model flow tube for "Move 1" have been shown previously. The cross sections of the model flow tube for "Move 2" are

Use Case 01 QSP: mode parameters fully specified



Model 01



**Figure 7.10-14.** Simple velocity limited system for getting from one position to another.

$x_{max}$	$x_{min}$	$d$
2	-2	0
0	-6	1
-2	-10	2
-4	-14	3
-6	-18	4
-8	-22	5
-10	-26	6
-12	-30	7
-14	-34	8

Suppose, now, that the goal in both QSP's is specified to be  $x_{gqsp} = [x_{gqspmin}, x_{gqspmax}] = [8, 12]$ , and that the initial range in both QSP's is  $x_{initqsp} = [x_{initqspmin}, x_{initqspmax}] = [5, 7]$ , as was the case in Example 1. The shifted flow tube for "Move 1" is as was shown in Example 1, and the valid duration (from the valid cross sections) is  $[2, 5]$ , as before. The cross sections for the shifted flow tube for "Move 2" are

$x_{max}$	$x_{min}$	$d$
12	8	0
10	4	1
8	0	2
6	-4	3
4	-8	4
2	-12	5
0	-16	6
-2	-20	7
-4	-24	8

Valid durations are 1 and 2, so the valid duration range is  $[1,2]$ . The duration ranges may then be further tightened based on the  $[l,u]$  specification in the QSPs.

### 7.10.7.3 Compilation of Hybrid Timed PCCA

There are two major steps in this compilation process. In the first step, a *flow tube* is computed for each mode. This computation is based on the initial, goal, and operating constraints for the mode. The resulting flow tube is a representation of the allowed state evolution of the continuous state of the mode over time. In the second step, temporal constraints implied by the flow tubes are combined with explicitly specified temporal constraints, resulting in an overall tightening of mode duration constraints. The compilation process results in flow tubes, and tightened temporal constraints, both of which are useful for performing qualitative simulations, and other kinds of analysis.

#### 7.10.7.3.1 Flow Tube Compilation

The flow tube is an abstraction, and a relaxation, in that it replaces the dynamics (difference equation constraints, actuation limits), which imply state evolution, with an explicit representation of the envelope of state evolution over time. Thus, for each time increment in the duration of a mode, the flow tube contains a convex set of feasible points in the continuous state space. The flow tube does not, however, provide detailed trajectory information about how a particular point in state space evolves, as specified by the difference equations.

Flow tubes can be used to determine whether a set of initial, goal, and operating constraints for a mode is consistent, and to establish duration bounds for the mode implied by the dynamics. In particular, flow tubes represent durations for which the initial, goal, and operating constraints are feasible. They are thus very useful for a variety of verification tasks.

A flow tube consists of a set of *cross sections*, where each cross section represents the feasible set of states at a particular duration increment. Each cross section is represented by a convex polytope of the form

$$\mathbf{Hx} \leq \mathbf{K} \quad (6)$$

The initial and goal regions of a mode are specified in this form as well. The algebraic portion of the operating constraints is also specified in this form.

Pseudocode for an algorithm for computing a sequence of cross sections, using backward reach set analysis is shown below.

```

ComputeModeFlowTubes(mode M) {
    for each goal region, g, in a M's transition function {
        ComputeGoalFlowTube(g, M);
    }
}

ComputeGoalFlowTube(g, M) {
    flow_tube = {};
    p1 = g;
    kmax = increment index corresponding to u(M);
    l_implicit = -1;
    u_implicit = -1;
    for k = 1 to kmax {
        p1 = ComputeOneStepBackwardReachSet(p1, dynamics(M));
        insert {k, p1} into flow_tube
    }
}

```

```

if (l_implicit < 0) { // Hasn't been set yet
    if (intersection(p1, init(M)) == init(M)) { // if p1 covers M completely
        // The duration is valid
        l_implicit = k;
    }
}

if (not(l_implicit < 0) and u_implicit < 0) { // l_implicit set, but
u_implicit not set
    if (not (intersection(p1, init(M)) == init(M))) {
        // The duration is not valid
        u_implicit = k - 1;
    }
}
}

// Result to be cached for g is flow_tube, l_implicit, and u_implicit
}

ComputeOneStepBackwardReachSet(p1, dynamics, M) {
    extreme_inputs = ExtremeInputCombinations(M);
    backward_vertices = {};

    for each extreme_input, inp, in extreme_inputs {
        for each vertex, v, in p1 {
            backward_vertex = OneStepBackwardDynamics(v, inp, dynamics);
            insert backward_vertex into backward_vertices;
        }
    }

    p_ret = ConvexHull(backward_vertices);
    return p_ret;
}
}

```

The function ComputeModeFlowTubes computes a flow tube, and implicit temporal bounds, for each goal region associated with an exit transition of the mode. ComputeModeFlowTube computes the flow tube for a particular goal. It accomplishes this by stepping through the discrete durations, limited by  $u(M)$ , the explicit bound on maximum duration for the mode. For each such duration, starting with 1, it computes the previous reach set (cross section) from the current one. Initially, the current reach set is the goal region. It checks if the duration is valid based on whether the reach set fully covers  $init(M)$ , the initial region for the mode. It then uses this information to set the implicit lower and upper bounds ( $l_{\text{implicit}}$ ,  $u_{\text{implicit}}$ ). It is assumed that there are no gaps in this interval where the duration is not valid.

The function ComputeOneStepBackwardReachSet computes the reach set in the backward direction, from a current reach set, and the dynamics of  $M$ . ExtremeInputCombinations returns the (cached) set of all possible combinations of extreme inputs (inputs where constraints are active for all elements). For each vertex in the current (specified) reach set, and for each input combination, it computes the backward vertex. It then takes the convex hull of all the backward vertices and returns that.

The function OneStepBackwardDynamics computes the backward vertex. Starting from the forward dynamics:

$$\mathbf{x}(k+1) = \mathbf{Ax}(k) + \mathbf{Bu}(k) \quad (7)$$

Then the backward dynamics are:

$$\mathbf{x}(k) = \mathbf{A}^{-1}(\mathbf{x}(k+1) - \mathbf{Bu}(k)) \quad (8)$$

This is used by OneStepBackwardDynamics to compute a state at increment k, given a state at increment k + 1, and a control input at k.

A problem with the above algorithm is that the polytopes grow exponentially in complexity as duration increases. This is because all combinations of input extreme values is exponential in the number of input elements. One way to circumvent this problem is to choose a fixed set of vertex directions for the polytopes. This limits the complexity of the polytopes, but the approximations will not be complete, and may miss significant regions of state space. This can be mitigated by using a large set of directions.

Suppose that we have decided on a set of directions:  $\{\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_m\}$ . Each direction is a unit vector in  $\Re^n$ . Using this approach, a flow tube cross section is the set of scalar values associated with each direction. The set of vertices for the cross section are  $\{s_1 \mathbf{r}_1, s_2 \mathbf{r}_2, \dots, s_m \mathbf{r}_m\}$ , where  $s_i$  is the scalar for direction  $i$ .

Computation of the scalars is accomplished using an addition at the end of OneStepBackwardDynamics. The convex hull computed by OneStepBackwardDynamics is represented by

$$\mathbf{Hx} \leq \mathbf{K} \quad (9)$$

The problem formulation then involves maximizing each scalar  $s_i$  subject to the constraints

$$\begin{aligned} \mathbf{x} &= s_i \mathbf{r}_i \\ \mathbf{Hx} &\leq \mathbf{K} \end{aligned} \quad (10)$$

or

$$\mathbf{Hr}_i s_i \leq \mathbf{K} \quad (11)$$

Each row of this is solved separately for  $s_i$  as if the inequality were an equality (to get the maximum), and then the minimum of these solutions is used as the value of  $s_i$ .

After the scalars for each direction have been computed, all vertices of the reach set are known, and this is returned.

#### 7.10.7.3.2 Temporal Constraint Compilation

There are two points in the validation process where temporal constraint compilation is performed. The first is for the model, independently of any use cases. For this, the compilation is very simple. Because the model represents possible transitions, but not the actual transition sequence for an entire plan, the most that can be done here is to tighten the temporal bounds specified in the model based on the temporal bounds implied by the flow tubes.

The second point where temporal constraint compilation is performed is during the qualitative simulation process, when the model is used to generate a mode sequence that satisfies a use case. When a candidate mode sequence is generated, temporal bounds from both the model and the use case have to be applied. This can imply a further tightening of temporal constraints for

each mode duration. To make this implied tightening explicit, we use a temporal constraint compilation technique based on *Simple Temporal Networks* (STN) [MMT98, HOF05]. This is a fast compilation technique that provides explicit tightest bounds on all mode durations. This allows the qualitative simulation to efficiently determine feasible duration schedules.

#### 7.10.7.4 Qualitative Simulation of a Compiled Hybrid Timed PCCA

After the model is compiled, the compiled HTPCCA is used to generate a qualitative simulation of the possible state evolution trajectories. As described previously, the inputs to the qualitative simulator component are the compiled HTPCCA, and a *partial QSP* representing use case requirements, as shown in Figure 7.10-6. The output is a sequence of mode assignments (qualitative states), with associated duration ranges for each mode. An additional byproduct of the forward simulation is a probability indicating the likelihood of the trajectory.

The qualitative simulation is accomplished by deciding mode transitions, and duration in each mode. Thus, qualitative simulation amounts to assignments to two kinds of discrete decision variables (duration index, transition index), subject to logical (discrete) and numeric (continuous) constraints. Valid mode transitions are determined by the guard (exit) conditions, which include the continuous goal regions. Valid mode durations are constrained by the tightened lower and upper temporal bounds for each mode.

Pseudocode for the qualitative simulation algorithm is shown below.

```

QualSim(HTPCCA model, QSP useCase) {
    // model is the compiled HTPCCA model.
    // useCase is a QSP representing use case requirements.

    Boolean satisfied = false;
    InitializeCandidateGenerator();
    while ((not satisfied) & (candidate = GenerateCandidateTrajectory(model,
useCase)) {
        satisfied = CheckCandidateTrajectory(candidate, model, useCase);
    }

    if satisfied
        return candidate; // Return successful candidate, if one exists.
    else
        return failure;
}

InitializeCandidateGenerator() {
    InitializeModelGraphSearch();
}

CandidateTrajectory candidate = GenerateCandidateTrajectory(HTPCCA model, QSP
useCase) {

    // This uses two data structures that are maintained statically within this
    function
}

```

```

// to represent the current state of the search.
// The two data structures are currentModeSequence, which represents the
state of the
// search through the model's mode graph, and currentSchedule, which
represents the state
// of the search through the possible schedules for the mode sequence
represented by
// currentModeSequence.

if ((currentModeSequence) & (MoreSchedules currentSchedule)) {
    candidate.modeSequence = currentModeSequence;
    candidate.schedule = GetNextSchedule();
    return candidate;
} else if (MoreModeSequences) {
    candidate.modeSequence = GetNextModeSequence();
    ApplyTemporalConstraints(candidate, useCase); // Apply additional
temporal constraints from use case.
    candidate.schedule = GetFirstSchedule();
    return candidate;
} else {
    return empty; // No more candidates
}
}

```

```

Boolean satisfied = CheckCandidateTrajectory(CandidateTrajectory candidate,
HTPCCA model, QSP useCase) {
    for mode in candidate.modeSequence {
        as nextMode = GetNextModeInSequence(mode, candidate.modeSequence);
        as transition = GetTransition(mode, nextMode);
        as ft = transition.flowTube;
        as rInit = nextMode.initialRegion;
        as rGoal = ft.goalRegion;
        FormulateLinearConstraint(rGoal subset of rInit);

        // Incorporate constraints from useCase
        if ((activity = GetActivityInUseCase(mode)) exists) {
            activityInit = activity.initialRegion;
            activityGoal = activity.goalRegion;

            if (activityInit exists) {
                FormulateLinearConstraint(rInit subset of activityInit);
            }

            if (activityGoal exists) {
                FormulateLinearConstraint(rGoal subset of activityGoal);
            }
        }
    }
}

```

```

        }
    }
}

return CheckLinearConstraints();
}

```

The current implementation of the algorithm uses a relatively simple, brute-force candidate generator. A more sophisticated approach, which would likely yield more efficient generation of good candidates, is to use the candidate generation algorithm in *Conflict-directed A\** [WR03].

The following sub-sections provide detailed information on two important aspects of the verification problem: 1) adaptation of model flow tubes to use case requirements; and 2) determination of probabilistic certificates of validation.

#### 7.10.7.4.1 Adapting Model Flow Tubes to Use Case

As introduced previously in Section 7.2.1, flow tubes in a model provide a template or prototype of the model’s dynamic behavior. The use case requirements typically do not match the template exactly. Therefore, an attempt is made to achieve a match by adjusting the model flow tubes (template) so that they satisfy the use case requirements. Such adjustment must not compromise the dynamic behavior expressed by the flow tube.

In the example in Section 7.2.1, the model specifies distance limits between points A and B, based on the dynamics and temporal limits. Suppose that in the model template, A is set to the origin. In this case, B can be thought of as representing the distance that can be achieved, not an absolute point. The use case QSP in this example has initial region requirements that are not at the origin, and goal region requirements that do not include point B. However, through a simple shift of position of the model flow tube, a match is achieved without compromising the dynamic constraints expressed in the flow tube.

A full discussion of parameterized flow tubes is beyond the scope of this document. However, we will describe a specific type of parameterization, *position shifting*, which can always be used, and which is useful for many types of problems. Position shifting was introduced in the example of Section 7.2.1 for simple velocity-limited flow tubes, but it can be extended to general polytope flow tube representations.

In order to understand how position shifting is accomplished for general flow tubes, consider the previous pseudocode for the function CheckCandidateTrajectory. It involves formulating a set of linear constraints, and then checking them. The formulations are for subset relationships between polytopes. Suppose that we have two polytopes,  $P_1$ , and  $P_2$ , and we wish to formulate linear constraints that require  $P_1 \subset P_2$ . The two polytopes are represented by sets of linear constraints of the form

$$\begin{aligned} \mathbf{H}_1 \mathbf{x} &\leq \mathbf{K}_1 \\ \mathbf{H}_2 \mathbf{x} &\leq \mathbf{K}_2 \end{aligned} \quad (12)$$

Let  $\mathbf{v}_1 = \mathbf{v}_{11}, \mathbf{v}_{12}, \dots, \mathbf{v}_{1p}$  be the vertex set for  $P_1$ . Then, the subset check for  $P_1 \subset P_2$  is formulated as

$$\begin{aligned} \mathbf{H}_2 \mathbf{v}_{11} &\leq \mathbf{K}_2 \\ \mathbf{H}_2 \mathbf{v}_{12} &\leq \mathbf{K}_2 \\ \dots \mathbf{H}_2 \mathbf{v}_{1n} &\leq \mathbf{K}_2 \end{aligned} \quad (13)$$

The first of these constraints is expanded as

$$\begin{aligned} \mathbf{H}_2(1;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(1;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(1;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(1) \\ \mathbf{H}_2(2;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(2;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(2;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(2) \\ \dots \\ \mathbf{H}_2(m;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(m;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(m;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(m) \end{aligned} \quad (14)$$

Now, suppose that we want to allow  $P_1$  to shift in position in order to achieve the subset condition. We accomplish this by introducing a shift variable,  $s$ , into 14, where  $s$  is associated with the coefficients corresponding to position (in this case, the first column of  $H_2$ ).

$$\begin{aligned} \mathbf{H}_2(1;1)s + \mathbf{H}_2(1;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(1;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(1;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(1) \\ \mathbf{H}_2(2;1)s + \mathbf{H}_2(2;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(2;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(2;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(2) \\ \dots \\ \mathbf{H}_2(m;1)s + \mathbf{H}_2(m;1)\mathbf{v}_{11}(1) + \mathbf{H}_2(m;2)\mathbf{v}_{11}(2) + \dots + \mathbf{H}_2(m;n)\mathbf{v}_{11}(n) &\leq \mathbf{K}_2(m) \end{aligned} \quad (15)$$

This can be expressed as

$$\mathbf{H}_2(:,1)s \leq \mathbf{C}_2$$

where  $\mathbf{C}_2$  is the vector  $\mathbf{K}_2 - \mathbf{H}_2 \mathbf{v}_{11}$ . Similar constraints are formulated for the other vertices in 16.

In this way, constraints are formulated that can then be checked to validate whether the flow tube constraints are satisfied. The constraint formulation makes use of the position shifting flexibility, but the dynamic constraint information in the flow tubes is preserved.

#### 7.10.7.4.2 Obtaining a Probabilistic Certificate of Validation

In order to compute a *probabilistic certificate of validation* using reach set analysis, we take probability distributions representing uncertain variables, and we set-bound them. This converts a probabilistic problem into a deterministic one. Uncertainty is propagated by propagating the set bounds according to the dynamics model. For example, a set bound on inputs implies set bounds on state according to the dynamics model. This approach is based on previous related research on reach set analysis incorporating uncertainty [APLS08, RLML06].

Our dynamic models are represented as linear difference equations.

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \quad (17)$$

We introduce uncertainty in two places, corresponding to the right-hand side of (17). First, we model disturbances in the inputs by adding a probabilistic component to the input vector  $\mathbf{u}$ . Thus, each element of  $\mathbf{u}$  becomes a random variable with some probability distribution. For purposes of analysis, we replace the distribution with a set bound, which is easier to propagate.

For a typical vehicle model, inputs correspond to forces acting on the vehicle, such as torque exerted by the engine on the wheels, wind gusts blowing against the vehicle, and bumps over which the vehicle rolls.

When there are multiple inputs, each may have its own probability distribution. Thus, a row in (17) is

$$x(k+1) = a_{11}x_1(k) + a_{12}x_2(k) + \dots + a_{1n}x_n(k) + b_{11}u_1(k) + b_{12}u_2(k) + \dots + b_{1p}u_p(k) \quad (18)$$

Here,  $A$  has  $n$  columns, and  $B$  has  $p$  columns. Each element of  $\mathbf{u}$  is given a set bound that covers some percentage of its probability distribution. This percentage of probability corresponds to the "success" contribution of that variable. In other words, it is a necessary condition for the variable to be within its set bounds for the plan to succeed. The probability of being within the set bounds is the percentage of the probability distribution that is covered. We refer to this individual success probability as  $p_s(u_i)$ . Figure 7.10-15 shows a block diagram of the input noise model, and how set bounds are used to cover a percentage of the probability distribution.

Figure 7.10-16 shows how set bounds on noise are translated into safety bounds on the input.

Since each of the input variables must be within its set bounds, the overall probability of success is the product of the individual probabilities.

$$p_s = \prod_{i=1}^p p_s(u_i) \quad (19)$$

This assumes that each input element is independent of the others (don't need joint probability distributions). This is not always true.

The success probability given by (18) is really just the success probability for a single time increment in an activity. Expanding over all time increments requires multiplying the probabilities for each time increment.

$$p_s = \prod_{k=1}^K \left[ \prod_{i=1}^p p_s(u_i) \right] \quad (20)$$

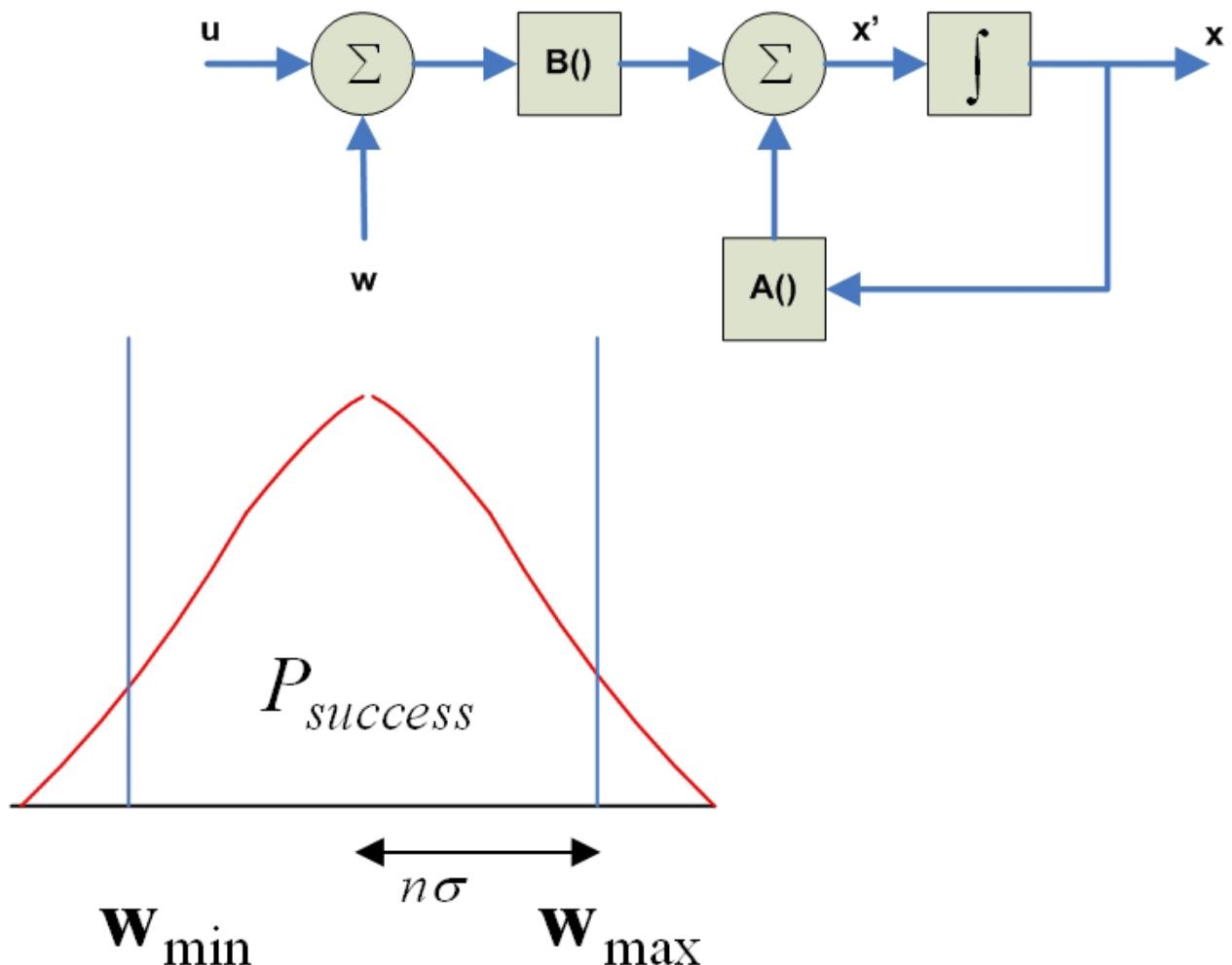
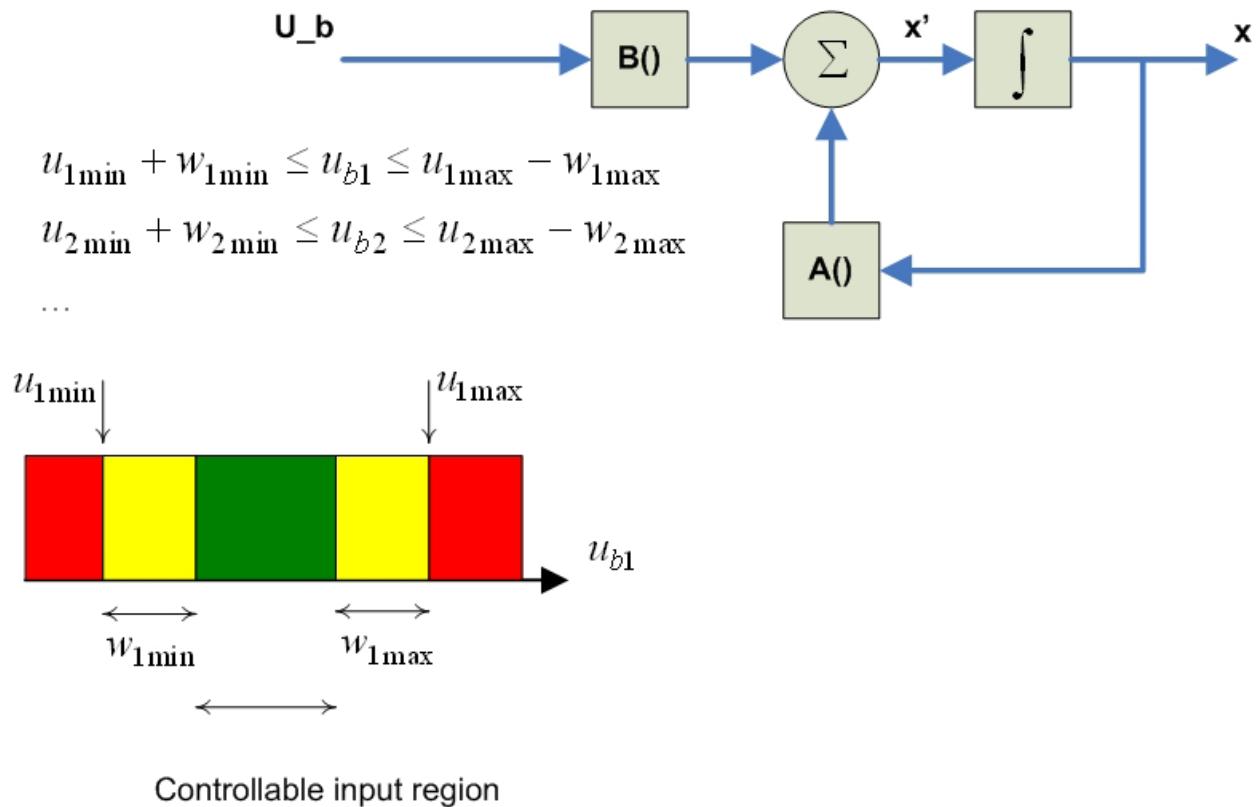


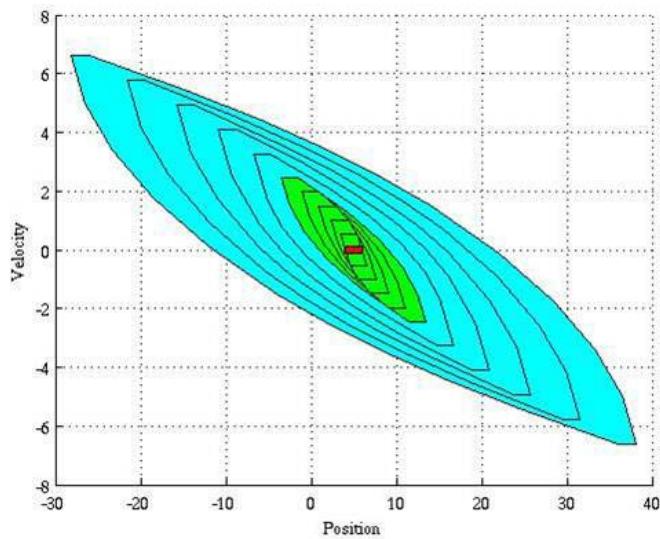
Figure 7.10-15. Set bounds cover a percentage of the noise distribution.



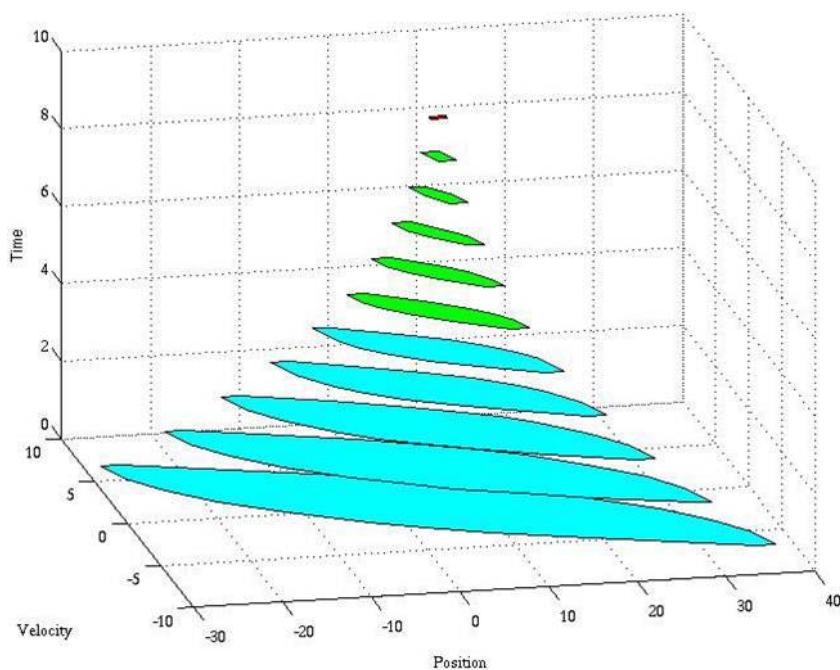
**Figure 7.10-16. Set bounds on noise are translated to safety bounds on the input.**

Figure 7.10-17 shows an example flow tube sequence corresponding to two successive activities. It is assumed, for this figure, that there are no disturbances, so there is no input uncertainty. The goal region is shown in red, the green cross sections are for the second activity, and the light blue cross sections are for the first.

Figure 7.10-18 shows an example flow tube sequence where there is uncertainty due to input disturbances. The figure on the left shows the flow tube sequence without uncertainty, as before, and then, superimposed, shows the more conservative cross sections in dark blue. The cross sections are computed to guarantee a 0.95 chance of success. The figure on the right is similar, but with a 0.68 chance of success. The conservative cross sections are shown in purple. Note that these are bigger than the ones for the 0.95 success case, because more failure is allowed.

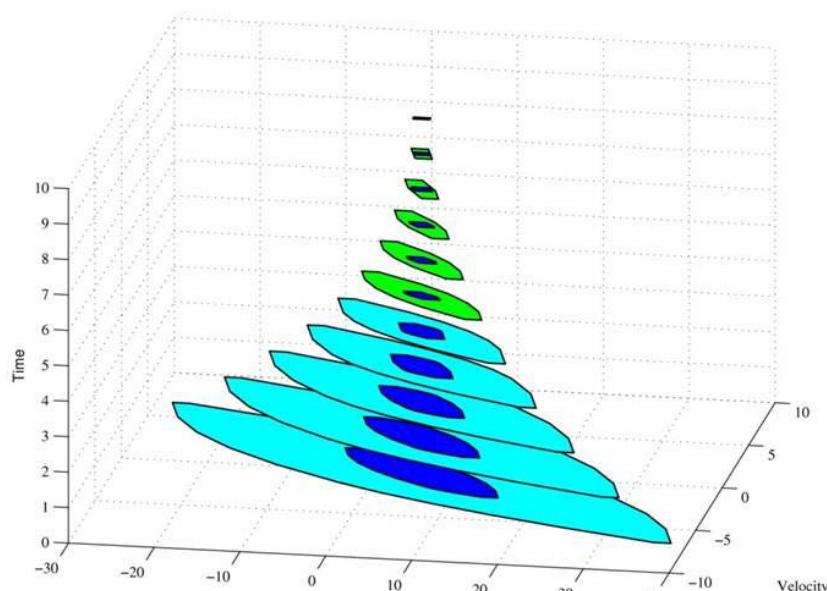


(a) Flow tube set with no uncertainty, view 1.

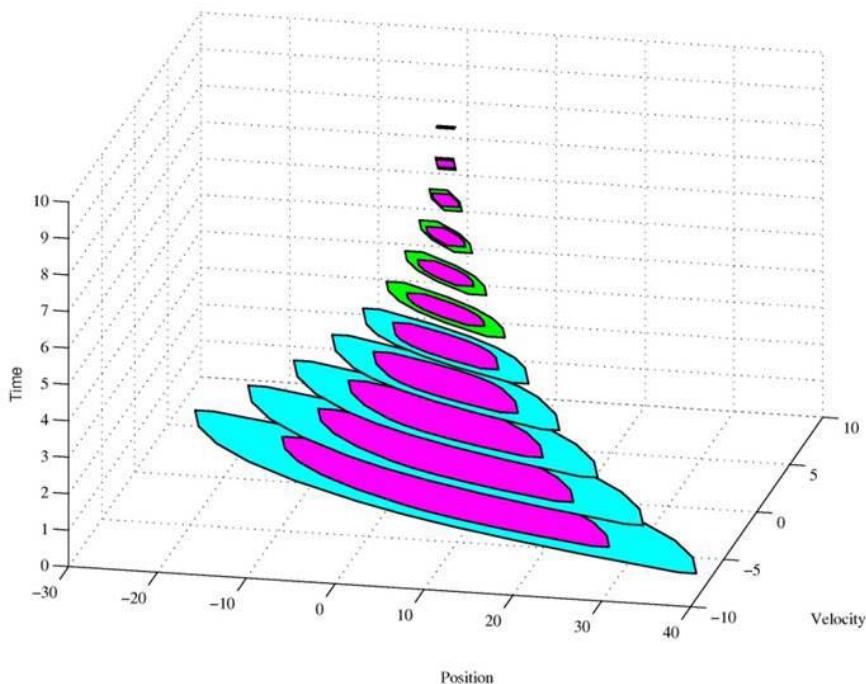


(b) Flow tube set with no uncertainty, view 2.

**Figure 7.10-17. Flow tube set for two successive activities, assuming no disturbances.**



(a) Flow tube set with uncertainty, probability of success = 0.95.



(b) Flow tube set with uncertainty, probability of success = 0.68.

**Figure 7.10-18. Flow tube set for two successive activities, with input disturbances.**

The formulation can easily be extended to handle model disturbances. These correspond to some level of uncertainty in the elements of  $A$  and  $B$ . Thus, each element has its own probability distribution, which may be computed using the uncertainty propagation techniques in the papers that Johan sent. This overall approach is analogous to that used in Kalman filtering, and in Lars Blackmore’s work.

As with the input elements, the parameters are converted into random variables. The difference here is that the state variables are not probabilistic; there is only one true state. This is a little trickier than the input case. For the input case, the set bounds are on the input variables, not the coefficients of  $B$ . For the model disturbances, the probability is associated with the coefficients of  $A$ , and possibly also of  $B$ .

The simplest way to model this is with one or more additional random variables added at the end of (18). These are then set bound, resulting in a success probability for each one. These probabilities are then multiplied, as before, to get an overall probability.

### 7.10.8 Bibliography

- [APLS08] Abate, A. Prandini, M. Lygeros, J. Sastry, S. (2008) “Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems”, *Automatica*, vol. 44, number 11, pages 2724–2734, Elsevier
- [HOF05] Hofmann, A.G. (2005) *Robust execution of bipedal walking tasks from biomechanical principles*, Ph.D. Thesis, MIT
- [HW06] Hofmann, A.G. Williams, B.C., (2006) “Exploiting spatial and temporal flexibility for plan execution of hybrid, under-actuated systems”, *Proc. AAAI*
- [MMT98] Muscettola, N. Morris, P. Tsamardinos, I., (1998) “Reformulating temporal plans for efficient execution”, In *Principles of Knowledge Representation and Reasoning*
- [MWI05] Martin, O.B. Williams, B.C. Ingham, M.D., (2005) “Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata”, *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol 20, number 1, AAAI Press
- [RLML06] Rakovic, S.V. Kerrigan, E.C. Mayne, D.Q. Lygeros, J., (2006) “Reachability analysis of discrete-time systems with disturbances”, *Automatic Control, IEEE Transactions on*, vol. 51, number=4, pages=546–561
- [WK91] Williams, B.C. Kleer, J., (1991) “Qualitative reasoning about physical systems: a return to roots”, *Artificial Intelligence*, vol 51, number 1-3, pages 1–9
- [WR03] Williams, B.C. Ragno, R.J., (2003) “Conflict-directed A\* and its role in model-based embedded systems”, *Journal of Discrete Applied Mathematics*

# **META Adaptive, Reflective, Robust Workflow (ARRoW)**

## **Phase 1b Final Report**

### **TR-2742**

## **Appendix 7.11 – Programmatic**

**13 October 2011**

---

**Reporting Period:** 10/14/2010 through 10/13/2011

**Contract Number:** HR0011-10-C-0108

**Prepared For:**

Defense Advanced Research Projects Agency  
3701 North Fairfax Drive  
Arlington, VA 22203-1714

**Prepared by:**

BAE Systems Land & Armaments L.P. (BAE Systems)  
4800 East River Road  
Minneapolis, MN 55421-1498

---

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

## Table of Contents

7.11	Programmatic Summary .....	1
7.11.1	Financial Data.....	1
7.11.2	Schedule.....	2

## 7.11 Programmatic Summary

### 7.11.1 Financial Data

The financial data below reflects the data available as of September 30, 2011. The data for Phase 1a (CLIN 0001) and 1b (CLIN 0002) is included for convenience; however, the scope of this report is Phase 1b (CLIN 0002) only.

R&D Status Report Program Financial Status												
CLIN	Negotiated Value	Funded Value	WBS or Task Element	Current		Cumulative to Date			At Completion			
				Actual Expend	Planned Expend	Actual Expend	% Budget Complete	At Completion	Latest Revised Estimate	Remarks		
0001	\$3,693,699	\$3,693,699	1.1	\$ -	\$ 299,706	\$ 458,079	96%	\$ 299,331	\$ 474,903	Note 1		
			1.2	-	919,141	951,875	96%	918,533	986,836	Note 1		
			1.3	-	719,135	633,636	96%	718,912	656,908	Note 1		
			1.4	-	858,265	680,276	96%	856,538	705,261	Note 1		
			1.5	-	145,424	141,951	96%	145,240	147,165	Note 1		
			1.6	-	466,883	418,567	96%	466,459	433,940	Note 1		
			Subtotal 0001		3,408,553	3,284,384	96%	3,405,013	3,405,013			
			Fee					288,686	288,686			
			Total 0001					3,693,699	3,693,699			
0002	9,461,917	9,461,917	2.1	\$ 172,858	\$ 723,498	\$ 952,268	79%	\$ 799,843	\$ 1,198,176	Note 2, Note 3		
			2.2	407,171	1,112,059	1,179,044	89%	1,231,480	1,324,334	Note 2, Note 3		
			2.3	767,718	4,698,862	5,136,428	97%	5,160,714	5,289,558	Note 2, Note 3		
			2.4	82,512	222,440	87,217	49%	252,027	177,163	Note 2		
			2.5	55,648	205,298	170,140	93%	259,404	183,144	Note 2		
			2.6	120,446	927,828	440,081	80%	1,019,487	550,581	Note 2		
			Subtotal 0002		1,606,353	7,889,986	7,965,179	91%	8,722,956	8,722,956		
			Fee					738,961	738,961			
			Total 0002					9,461,917	9,461,917			
			MR or UB									
	13,155,616	13,155,616	Cost Subtotal	1,606,353	11,298,538	11,249,563	93%	12,127,969	12,127,969			
			Fee Subtotal					1,027,647	1,027,647			
			TOTAL					13,155,616	13,155,616			

Is current funding sufficient for the current fiscal year? YES

What is the next FY funding requirement at current anticipated levels? For FY12 CLIN 0001 \$ - CLIN 0002 \$ -

Have you included in the report narrative any explanation of the above data and are they cross-referenced? YES

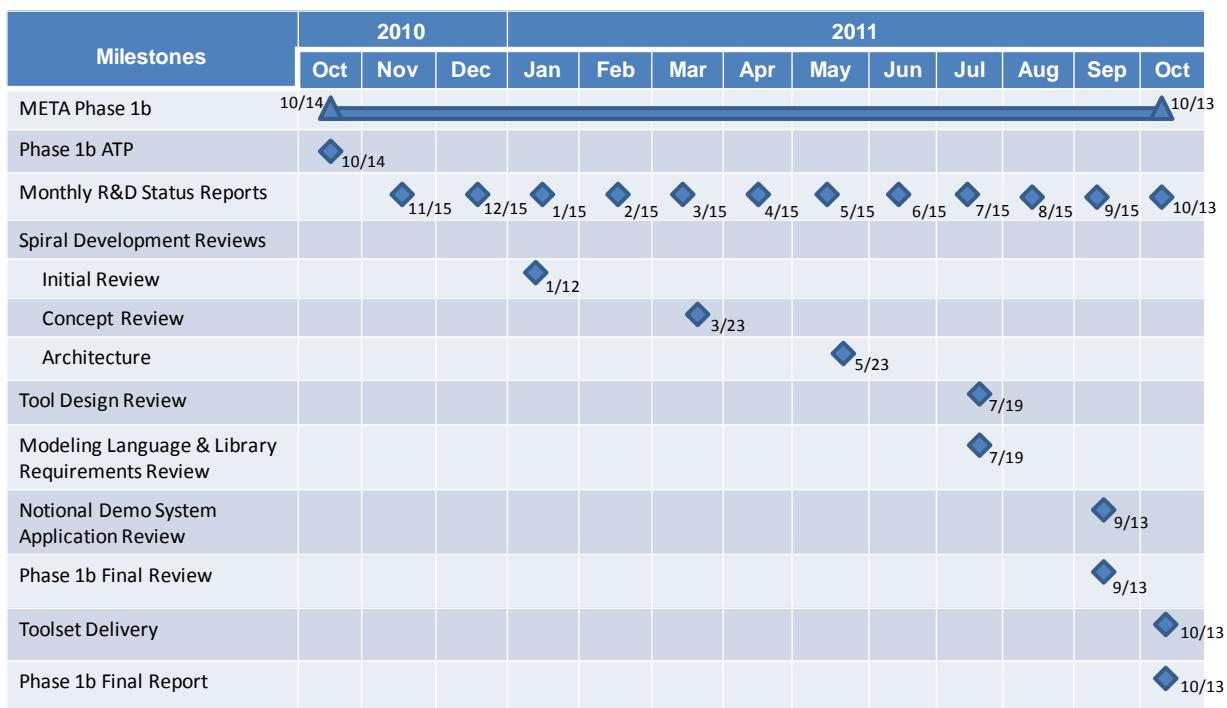
What are the estimated termination costs (non-binding)? \$ 878,407

**Note 1:** Remaining spending is reserved to cover potential future billing rate increases.

**Note 2:** Latest Revised Estimate includes the anticipated spending during 1-13 October, as well as a reserve to cover potential future billing rate increases.

**Note 3:** Notional Demo efforts were largely completed early in September and focus shifted to finishing Tool Design and Delivery and Language efforts.

### 7.11.2 Schedule



The schedule above indicates the date that each of the milestones on the chart, depicted as small blue triangles with a date in the lower right corner, were achieved. All milestones (which reflect reviews and product deliveries) and the deliverables required along with them (presentation material, demonstrations, and software) were completed and delivered on time. Bi-weekly review meetings with DARPA program management were held from October 2010 through April 2011. At the Principal Investigator Meeting in May 2011, Paul Eremenko informed the META community that the bi-weekly review meetings would be replaced with meetings between the Principal Investigator of each of the META program teams and DARPA program management, to be scheduled at the discretion of DARPA program management. Dr. Steven Banks met with Paul Eremenko on June 21, 2011 to discuss various topics, and again on August 9<sup>th</sup>, 2011 to discuss the overall technical approach formulated by the BAE Systems team.