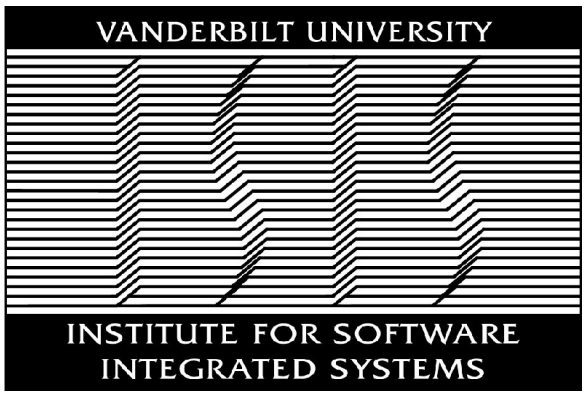


Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37235

The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Design Language, Modeling Framework, and Analysis

Joseph Porter, Graham Hemingway, Harmon Nine, Chris vanBusKirk, Nicholas Kottenstette, Gabor Karsai, and Janos Sztipanovits

TECHNICAL REPORT

ISIS-10-109

NOTICE: This is an excerpt from my dissertation, in preparation for submitting it for publication. This work contains excerpts from papers in the ACES-MB workshop, and the EMSoft conference. Please send comments and corrections to jporter@isis.vanderbilt.edu. Thanks, -Joe

1 INTRODUCTION

Consider a possible digital control system architecture for use in a flight control system. Sensors, actuators, and data networks are designed redundantly to mitigate faults. In our proposed architecture the underlying platform implements a variant of the time-triggered architecture (TTA) [1], which provides precise timing and reliability guarantees. Safety-critical tasks and messages execute according to strict precomputed schedules to ensure synchronization between replicated components and provide fault mitigation and management. Deployed software implementations of the control functions must pass strict certification requirements which impose constraints on the software as well as on the development process. The additional burden of design analysis required to establish safety increases cost and schedule, decreasing the flexibility of the development process.

In modern embedded control system designs, graphical modeling and simulation tools (e.g. Mathworks' Simulink/Stateflow) represent physical systems and engineering designs using block diagram notations. Design work revolves around simulation and test cases, with code generated from models when the design team reaches particular schedule milestones. Control designs often ignore software design constraints and issues arising from embedded platform choices. At early stages of the design, platforms may be vaguely specified to engineers as sets of trade offs [2].

Software development uses Unified Modeling Language Computer-Aided Software Engineering (UML CASE) tools to capture design artifacts and relationships such as types, components, interfaces, interactions, timing, fault handling, and deployment. Software development workflows focus on source code creation, organization, and management, followed by testing and debugging on target hardware. Physical and environmental constraints are often included in models, but the use of those constraints lies on the forefront of model-based embedded systems research. In many cases in practice such constraints only serve as documentation to developers and are encoded manually into test cases, losing many of the potential benefits of formally representing them in design models.

Complete control system software designs rely on multiple aspects. Designers lack tools to model the interactions between the hardware, software, and the environment with the required fidelity. For example, software generated from a carefully simulated synchronous dataflow model of the controller functions may fail to perform correctly when its functions are distributed over a shared network of processing nodes. Cost or availability considerations may force the selection of platform hardware that limits timing accuracy or data precision beyond originally designed bounds. None of the current design, analysis, or development techniques support comprehensive (i.e. multi-domain) validation of certification requirements to meet government safety standards. Model and code analysis tools must all be integrated to have the same semantic view of the design details.

1.1 Overview

We aim to create a Domain Specific Modeling Language (DSML) to address problems of design consistency across the entire development flow for a distributed embedded control system design. Often, the best solutions involve iterating the design cycle as problems are discovered or problem understanding increases. Our DSML captures the relationships between concepts in the different design domains described, and supports the integration of analysis tools and code generation. The ultimate goal is to assess the performance and safety of code generated from the models in the context of the actual platform on which it will execute, subject to platform-induced delays and uncertainties.

High-Confidence Design Challenges

We identify several specific challenges that arise because of inconsistencies between domains in a high-confidence embedded development project. Some of the challenges are fundamental, and others arise because of our attempts to use models to resolve consistency problems.

1. Controller, software, and hardware design domains are highly specialized and often conceptually incompatible. Sharing model artifacts between designers in different domains can lead to consistency problems in engineering solutions or implementations based on incomplete or faulty understanding of design issues. Current state of the art resolves differences in understanding by reviewing many of the details in numerous meetings and personal discussions. Manual reconciliation of issues occurs as individual designers receive assignments to modify and correct the design. In the worst cases serious incompatibilities are not discovered until very late in the design cycle, leading to project overruns and cancellations[3]. Several large modeling tool projects (for example, AADL [4] and Topcased[5]) work to integrate tools from independent research and development teams into a common design environment featuring a standardized modeling language. Resolution of semantic consistency between integrated tools to improve design efficiency is a serious issue in such efforts.

2. Incompatibilities between models and assumptions in different design domains create a related problem. For example, controller design properties which are verified using simulation models may no longer be valid when the design becomes software in a distributed processing network. Currently control designers use conservative performance margins to avoid rework when performance is lost due to deployment on a digital platform.

3. Long development, deployment, and test cycles limit the amount of iterative rework that can be done to get a correct design. If a particular design analysis is costly or time-consuming, the team cannot afford to iterate the design from its early stages in order to resolve problems. Currently high-confidence design requires both long schedules and high costs.

4. Automating steps in different design and analysis domains for the same models and tools requires a consistent view of inferred model relationships across multiple design domains. If integrated tools have different views of the model semantics, then their analyses are not valid when the results are integrated into the same design. Therefore, all of the tools used in the design process must have a consistent view of design details. Explicitly reconciling semantics between formalisms and tools is costly and time-consuming. Often the effort cannot be justified outside of academic research unless the results are applicable to numerous designs.

5. As our research explores new directions in high-confidence design, modification of the ESMoL metamodel (language specification) creates maintenance problems for ESMoL models and for interpreter code that translates them into analysis artifacts and generated code. We would like to isolate interpreter development from the language to a degree in order to allow the ESMoL language to evolve with our research. ESMoL models can be updated to new versions of the language using features built into the tools, but nothing exists yet to handle those problems for interpreter code.

Model-Integrated Solutions

We propose a suite of tools that aim to address many of these challenges. Currently under development at Vanderbilt's Institute for Software Integrated Systems (ISIS), these tools use the Embedded Systems Modeling Language (ESMoL), which is a suite of domain-specific modeling languages (DSML) to integrate the disparate aspects of a safety-critical embedded systems design and maintain proper separation of concerns between control engineering, hardware specification, and software development teams. The Embedded Systems Modeling Language (ESMoL) encodes in models the relationships between controller functions specified in Simulink, software components that implement those functions (i.e. dataflow, messaging interfaces, etc. . .), and the hardware platform on which the software will run. Many of the concepts and features presented here

also exist separately in other tools. We describe a model-based approach to building a unified model-based design and integration tool suite which has the potential to go far beyond the state of the art.

1. The ESMoL language and tools provide a single multi-aspect embedded software design environment so that modeling, analysis, simulation, and code generation artifacts are all clearly related to a single design model. We aim to incorporate models appropriate to the different design domains in a consistent way using the Model-Integrated Computing (MIC) approach discussed below. ESMoL models use language-specified relations to associate Simulink control design structures with software and hardware design concepts to define a software implementation for controllers. Further, ESMoL is a graphical modeling language which integrates into existing Simulink-based control design work flows[6].

2. ESMoL models include objects and parameters to describe deployment of software components to hardware platforms. Analysis artifacts and simulation models generated from ESMoL models contain representations of the behavioral effects of the platform on the original design. We include platform-specific simulations to assess the effects of distributed computation on the control design [7].

3. ESMoL's integrated analysis, simulation, and deployment capabilities can shorten design cycles. The ESMoL tool suite includes integrated scheduling analysis tools([8]) so that static schedules can be calculated in rapid design and simulation cycles. We include automatic generation of platform-specific task configuration and data communications code in order to rapidly move from modeling and analysis to testing on actual hardware.

4. ESMoL uses a two-stage interpreter architecture in order to integrate analysis tools and code generators. The first stage resolves any inferred model relationships from ESMoL models into a model in an abstract language (ESMoL_Abstract), much in the same way that a parser creates an abstract syntax tree for a program under compilation. The ESMoL design language allows relational inference where appropriate in order to make the designer more productive. The Stage 1 interpreter resolves object instances, parameters, and relations, and stores them in an ESMoL_Abstract model. Model interpreters for analysis and generation use this expanded model to guarantee a consistent view of the relationships and details, and to share code efficiently in an integrated modeling tool development project. The two-stage approach also isolates the interpreter code from the structure of the ESMoL language. Changes to the language are principally isolated from the interpreter code by the first stage transformation.

5. We generate analysis models and code from the intermediate language using simple template generation techniques[8]. Round-trip incorporation of calculated schedule analysis results back into the ESMoL model helps to maintain consistency as models pass between design phases.

Fig. 1 depicts a design flow that includes a user-facing modeling language for design and an abstract intermediate language for supporting interpreter development and maintenance. During design, a software modeler imports an existing Simulink control design into the Generic Modeling Environment (GME) [9], configured to edit ESMoL models (Step 1). The modeler then uses the dataflow models imported from Simulink to specify the functions of software components which will be used to implement the controllers. These component specifications represent the behavior of synchronously executing C code compiled from a dataflow model, and which are extended with interfaces defining input and output message structures for data distribution. We also specify the mapping from dataflow I/O ports to and from fields in the messages (Step 2). Designers specify the hardware topology for a time-triggered distributed processing network using another integrated design language (Step 3). A modeler instantiates component instances to create multi-aspect models where logical dependencies, hardware deployment, and timing models can be specified for the software architecture (Steps 4 and 5). Note that the requirements called out in the diagram are still conceptual. We have language elements for capturing end-to-end latency specifications, but more general formal requirements capture is a possible future effort for ESMoL.

A completed model is transformed (via the Stage 1 transformation) into a model in the ESMoL_Abstract language, resolving all implied relationships and structural model inferences (Step 6). Model interpreters

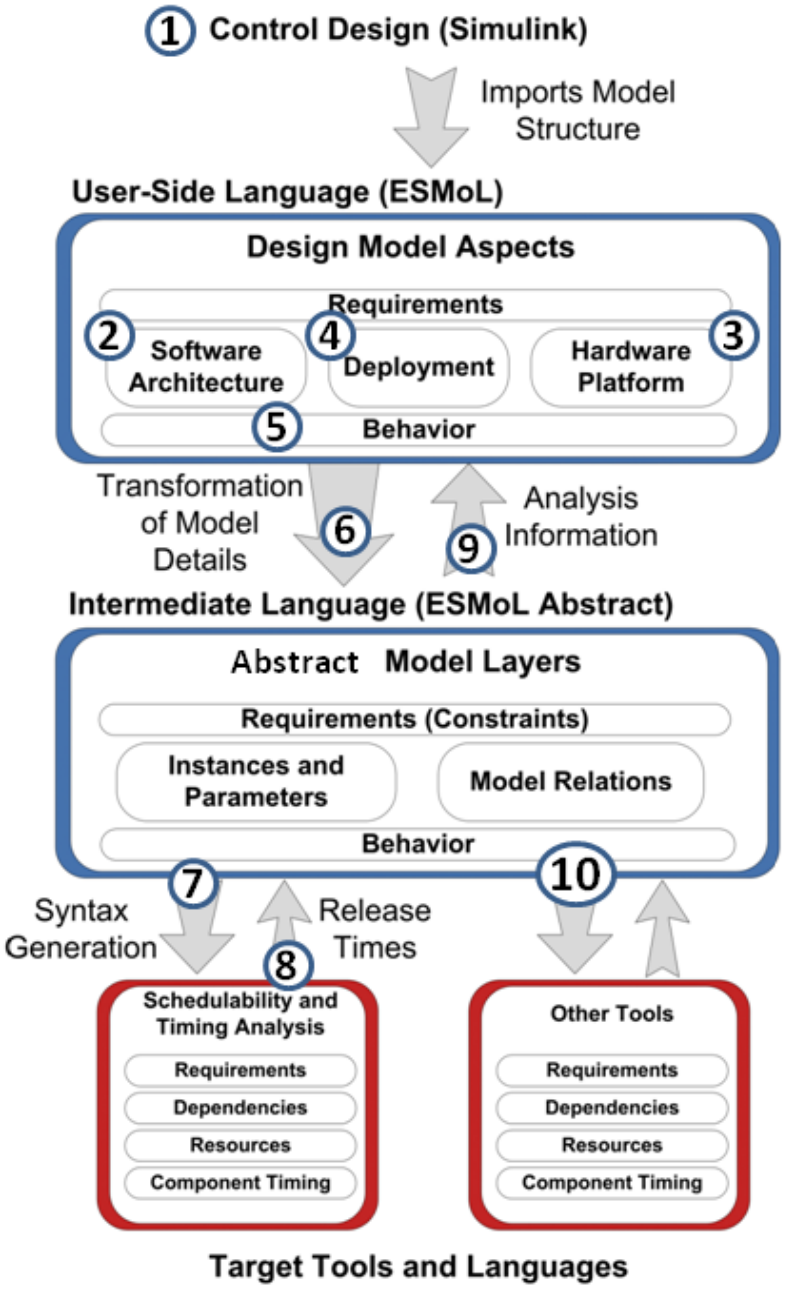


Figure 1: Flow of ESMoL design models between design phases.

for design analysis (in this case calculating time-triggered schedules) are integrated using the Stage 2 model transformation from ESMoL_Abstract models to analysis specifications (Step 7). Another model interpreter imports results from the analysis (in this case, scheduled start times) back into the ESMoL_Abstract and ESMoL models (Steps 8 and 9). Finally, designers can also create platform-specific simulations and generate deployable code using the Stage 2 transformation (Step 10).

In a later section we discuss the relationship between the behavior represented by the original Simulink model and the behaviors represented by ESMoL and ESMoL_Abstract (Steps 5 and 10). ESMoL provides a great deal of modeling flexibility, as subsets of the Simulink model are used in Step 2 to define software components. These subsets can be replicated to model redundant computation networks, for example. In Step 4 they are aggregated to define dataflows, and then partitioned to define deployment of those dataflows. With all of the language flexibility provided, we need to ensure that the synchronous semantics of the original Simulink model are preserved in the distributed implementation in order to ensure that the inherent correctness properties (functional determinism, timing determinacy, and deadlock freedom) are also preserved.

The illustrated design flow represents only a single iteration in the overall development work flow to be discussed later. In the sequel we will use the expression *design flow* to indicate the work of modeling, analyzing, and generating code for a single design. *Development flow* will indicate the macro-level iterative development process which includes one or more design flow iterations.

2 Related Work: Languages and Tools for Embedded Systems Design

A number of projects seek to bring together tools and techniques which can automate different aspects of high-confidence distributed control system design and analysis:

- AADL is a textual language and standard for specifying deployments of control system designs in data networks[4]. AADL projects also include integration with the Cheddar scheduling tool[10]. Cheddar is an extensible analysis framework which includes a number of classic real-time scheduling algorithms[11].
- Giotto[12] is a modeling language for time-triggered tasks running on a single processor. Giotto uses a simple greedy algorithm to compute schedules. The TDL (Timing Definition Language) is a successor to Giotto, and extends the language and tools with the notion of modules (software components)[13]. One version of a TDL scheduler determines acceptable communication windows in the schedule for all modes, and attempts to assign bus messages to those windows[14].
- The Metropolis modeling framework[15] aims to give designers tools to create verifiable system models. Metropolis integrates with SystemC, the SPIN model-checking tool, and other tools for schedule and timing analysis.
- Topcased[5] is a large tool integration effort centering around UML software design languages and integration of formal tools.
- Several independent efforts have used the synchronous language Lustre as a model translation target (e.g. [16] and [17]) for deadlock and timing analysis.
- RTComposer[18] is a modeling, analysis, and runtime framework built on automata models. It aims to provide compositional construction of schedulers subject to requirements specifications. Requirements in RTComposer can be given as automata or temporal logic specifications.
- The DECOS toolchain [19] combines a number of existing modeling tools (e.g. the TTTech tools, SCADE from Esterel Technologies, and others) but the hardware platform modeling and analysis aspects are not covered.

We are creating a modeling language to experiment with design decoupling techniques, integration of heterogeneous tools, and rapid analysis and deployment. Many of the listed projects are too large to allow experimentation with the toolchain structure, and standardization does not favor experimentation with syntax or semantics. Due to its experimental nature some parts of our language and tool infrastructure change very frequently. As functionality expands we may seek integration with existing tools or standards as appropriate.

3 Tools and Techniques

The models in the example and the metamodels described below were created using the ISIS Generic Modeling Environment tool (GME) [9]. GME allows language designers to create stereotyped UML-style class diagrams defining metamodels. The metamodels are instantiated into a graphical language, and metamodel class stereotypes and attributes determine how the elements are presented and used by modelers.

The Model-Integrated Computing (MIC) approach[20] builds up DSMLs by creating specific sublanguages to capture concepts and relationships for different facets of the design domain, and then integrating those sublanguages into a common modeling language by precisely specifying the structural relationships between those sublanguages. In a GME metamodel a sublanguage is called a *paradigm*. We will use the terms sublanguage, language, and paradigm interchangeably. Confusion is resolved by explicitly naming the paradigms involved in the discussion.

The GME metamodeling syntax may not be entirely familiar to the reader, but it is well-documented in Karsai et al [9]. Class concepts such as inheritance can be read analogously to UML. Class aggregation represents containment in the modeling environment, though an aggregate element can also be flagged as a port object. In the modeling environment a port object will also be visible at the next higher level in the model hierarchy, and available for connections. GME allows the specification of association classes which are visualized as edge connections between objects in models. For example, the dot between the *Connectable* class and the *Wire* association class (Fig. 2) represents an edge connecting two objects which are subtypes of *Connectable*. One other useful concept from a GME metamodel is the reference, which is a lightweight object that refers to another object somewhere else in the model hierarchy, in much the same way that a pointer references a concrete object in memory. A reference object appears in the Metamodel specification associated with another class, providing a convenient notation for model connections which span the model hierarchy[9]. This allows the modeler to create a pointer-like object, with is visualized with the same interface (port structure) as the associated class, but which actually refers to the original object.

Another key technology used in the ESMoL tool suite is the GReAT model transformation language (and its associated code generation tools)[21]. The ESMoL suite contains a pair of platform-independent code generators for Simulink and Stateflow models. The transformations take Simulink and Stateflow blocks, and create equivalent models in another language (SFC) that corresponds to an abstract syntax graph for fragments of C code. Functional code generation proceeds by simply traversing and printing the SFC models. Other generators use the UDM C++ modeling API [22] to create code implementing the platform-specific code to wrap functions as tasks, define communication messages structures, and configure a time-triggered virtual machine to execute the generated code. These generators as well as generators for platform-resimulation models are described elsewhere (see Porter et al [6], Thibodeaux [23], and Hemingway et al [7] for details).

Platform-based design partitions design frameworks into designer-supplied components and platform-provided services[2]. High-confidence systems require services and guarantees for correct and efficient execution such as real-time execution, data distribution, and fault tolerance. Platform-based design allows the construction of complex systems by facilitating reuse over common execution behaviors. The platform also defines a formal model of computation (MoC) [24], which predicts how the concurrent objects of an application interact (i.e. synchronization and communication). We use an implementation of the time-triggered architecture as a platform layer in order to reduce timing variances in sensing, actuation, and distributed data communications [1][23]. The central idea of the time-triggered architecture is to provide deterministic and fault-tolerant synchronous execution in order to ensure the consistent behaviors of distributed replicas

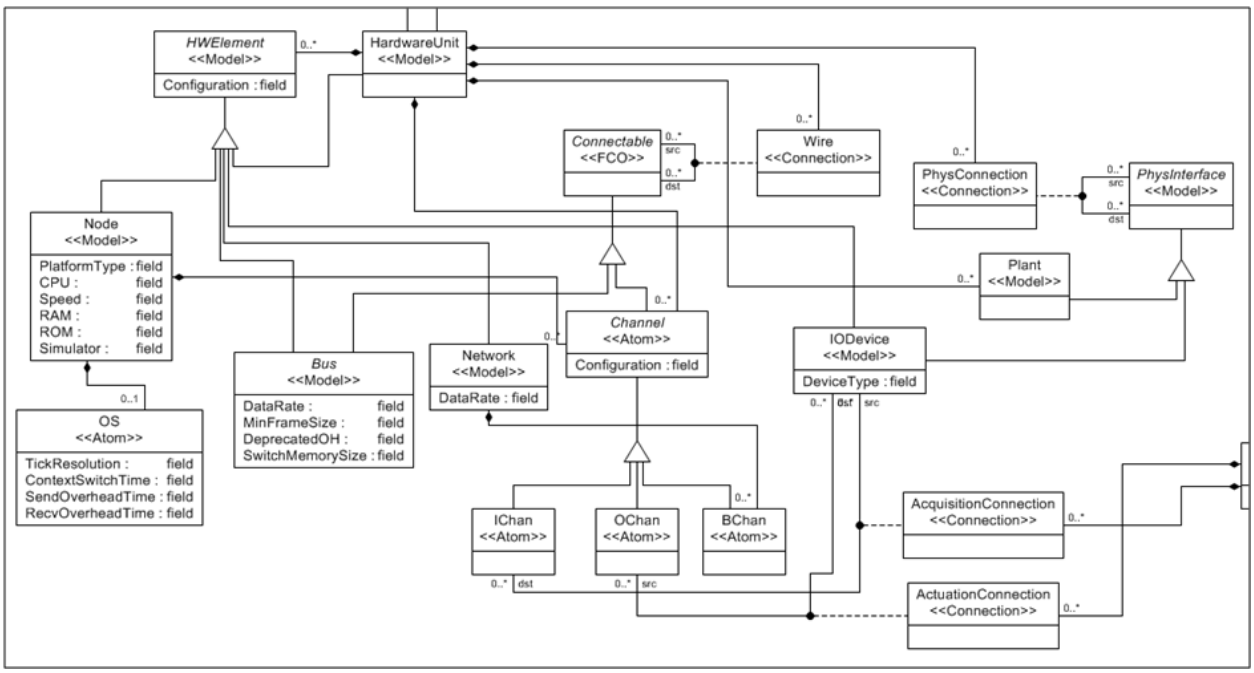


Figure 2: Platforms. This metamodel describes a simple language for modeling the topology of a time-triggered processing network.

of controller components.

4 The ESMoL Languages

To motivate our description of the facets of ESMoL, we focus on an actual control design model for the Starmac quadrotor helicopter [25][26]. Fig. 3 depicts its control architecture, consisting of two nested control loops. From left to right in the diagram, the *Input Filters* restrict the input trajectory commands to prevent maneuvers beyond the physically safe limits of the helicopter. The *Outer Loop* PD controller takes the requested position reference and the position data from the sensors, and calculates the attitude required for the quadrotor to achieve the requested change in position. *Saturation* is another limiter to ensure that the commanded attitude actuation is realizable. The *Inner Loop* PD controller takes the attitude command from the *Outer Loop* and measured attitude data, and calculates the motor thrusts required to achieve the commanded attitude. *Motor Compensator* filters the thrust commands to account for response delays in the motors which drive the rotors. Finally, the *Dynamic Model* describes the physical behavior of the helicopter, including the imprecision introduced by the sensors which measure position and attitude. The ESMoL model examples given below come from the design model for the quadrotor, except where noted.

Requirements Analysis (RA)

Formal requirements modeling offers great promise, but in ESMoL requirements modeling is still in conceptual stages. Informally, we require stability of the software-implemented closed-loop control system over the full range of possible inputs, and satisfaction of the calculated timing constraints (task release times and deadlines).

Functional Design (FD)

In ESMoL, functional specifications for components can appear in the form of Simulink/Stateflow models or as existing C code snippets. ESMoL does not support the full semantics of Simulink. In ESMoL the execution of Simulink data flow blocks is restricted to periodic discrete time, consistent with the underlying time-triggered platform. This also restricts the type and configuration of blocks that may be used in a design. Continuous integrator blocks and sample time settings do not have meaning in ESMoL. C code snippets are allowed in ESMoL as well. C code definitions are limited to synchronous, bounded response time function calls which will execute in a periodic task with a fixed amount of memory.

An automated importer constructs an ESMoL model from a Simulink control design model. The new model is a structural replica of the original Simulink model, only endowed with a richer software design environment and tool-provided APIs for navigating and manipulating the model structure in code. The Simulink and Stateflow sublanguages of our modeling environment are described elsewhere[27]. The ESMoL language evolved from another DSML known as ECSL-DP. They share many concepts, but ESMoL departs from many of the modeling structures previously described by Neema in order to increase the flexibility and generality of the language.

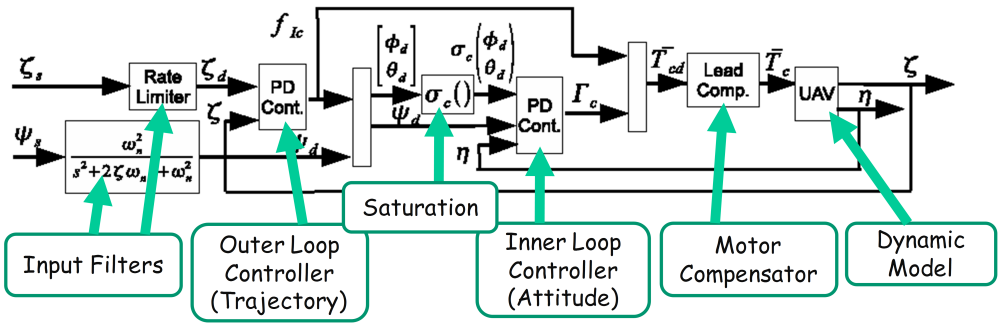


Figure 3: Basic architecture for the quadrotor control problem.

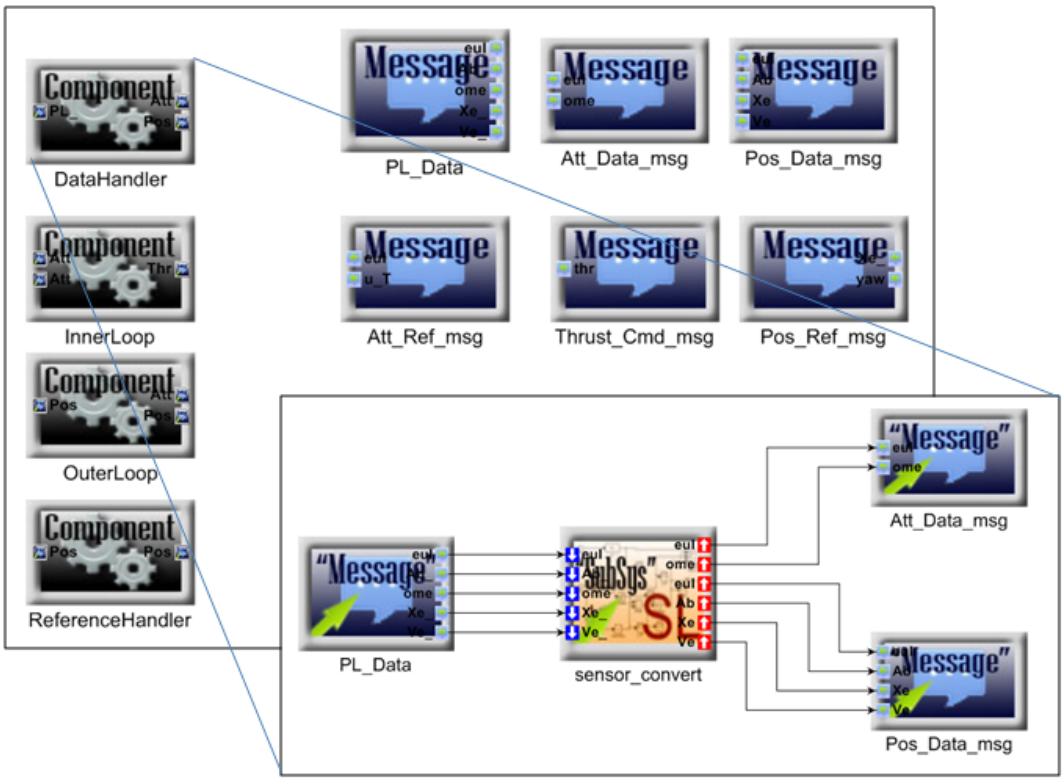


Figure 4: Quadrotor component types model from the *SysTypes* paradigm.

Component Design (CD)

In the component design phase (CD) we specify software interfaces for the functions which will run in the distributed controller network. A component type has a unique name (i.e. *InnerLoop*), and information to find or generate its implementation in C (in this case, the file name and model path to the Simulink subsystem "QuadRotor/STARMAC/InnerLoop"). A component specification contains a reference to a Simulink subsystem, as well as references to message structure objects. The message structure objects will represent message types, and each reference from a component definition represents an interface through which that message is sent or received. Internally, the direction of the connection from the message reference to the ports on the Simulink object determine whether the port sends or receives. We do not allow multi-directional message transfers on the same interface. When the component is instantiated in the design model (e.g., in the logical architecture diagram described below) the message references specified here will appear as ports on blocks representing the instance. Connections to and from those ports represent the transfer of an instance of that data message into or out of the component instance.

Fig. 4 shows an example of a model from the component interface definition language. Message fields and their sizes are specified here, as well as component implementations and interfaces. These specifications define software component types in an ESMoL model, which are instantiated and assigned to hardware in the architecture and deployment models, respectively. The quadrotor model has four different component types (each instantiated once) and six message types (instantiated as the ports objects appearing on the component instances later in the design). The breakout inset in the figure shows the internals of the *DataHandler* component specification. The *sensor_convert* subsystem block in the center is a reference to a Simulink block specifying the data conversions that transform raw sensor data into scaled, formatted data for use by the controller blocks.

The blocks on the outer edges of the figure (Fig. 4) are references to messages defined at the top level of

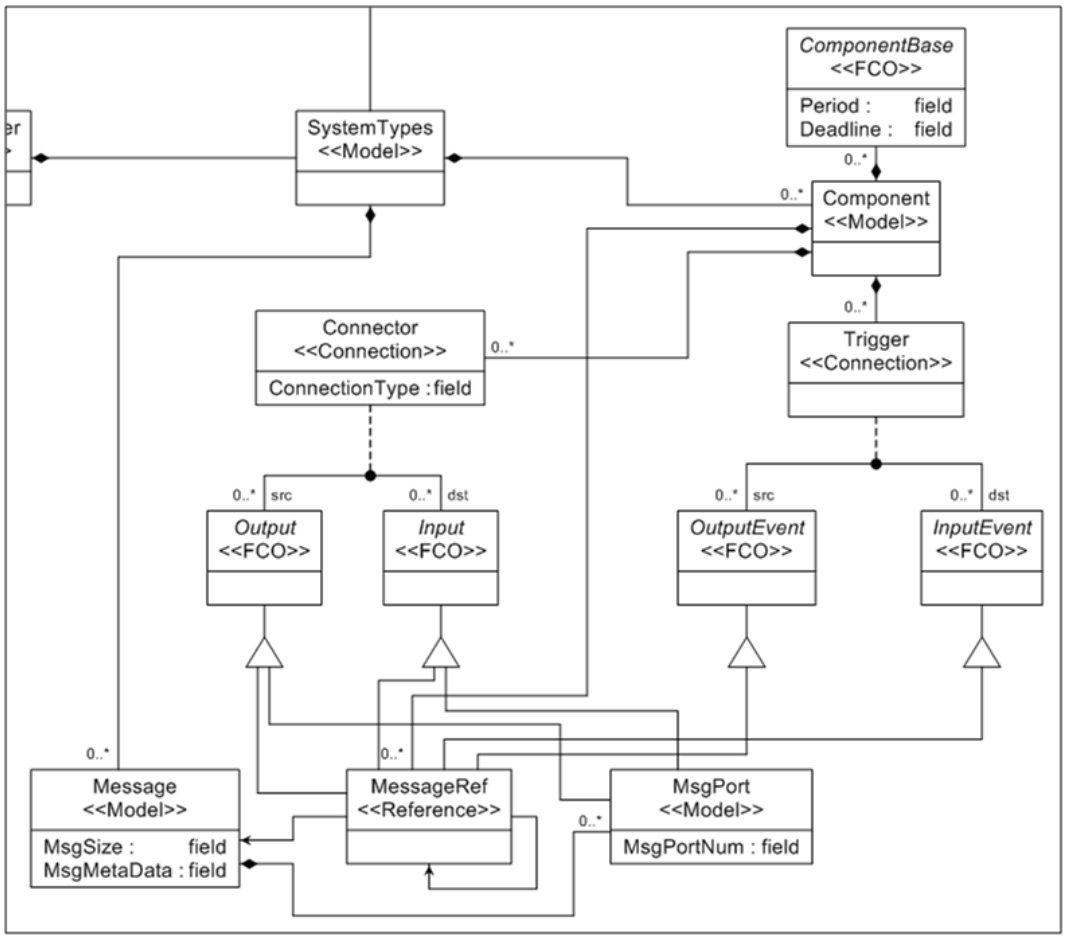


Figure 5: SystemTypes Metamodel.

the system types model. On the left is the raw data message from the sensors. On the right are the attitude data message (for local consumption by the inner loop), and the position data message (sent remotely to the outer loop). The three message reference blocks in the inset appear as ports on the *DataHandler* block (top left in the figure). Inside the component type definition, ports on the message objects correspond to C structure fields. The field types are inferred from the data types imported from the connected Simulink signal port objects. The connections between the message ports and the Simulink reference block ports describe the direction and details of data flow between the implemented message structures and the specified functional block.

Fig. 5 portrays the *SystemTypes* sublanguage, which encodes these structures and relations. Components of different types (here Simulink block references or C code blocks) specify the component functions. Message references (*MessageRef* objects) define interfaces on the components, and ports on message objects (*MsgPort* objects) represent message data fields as in the *DataHandler* example. The *Input* and *Output* port classes are typed according to the implementation class to which they belong (i.e., either Simulink signal ports or C function arguments). The connections between the block reference and the *MsgPort* objects describe the details required to marshal and demarshal the data fields in the messages for use by the specified function. Synchronous, periodic, discrete-time Simulink blocks and bounded-time synchronous C function calls are compatible at this modeling stage, because their model elements both represent the code that will finally implement the functions. These units are modeled as blocks with ports, where the ports represent parameters passed into and out of C function calls. The *Trigger* and *Event* types are not discussed here, as they relate

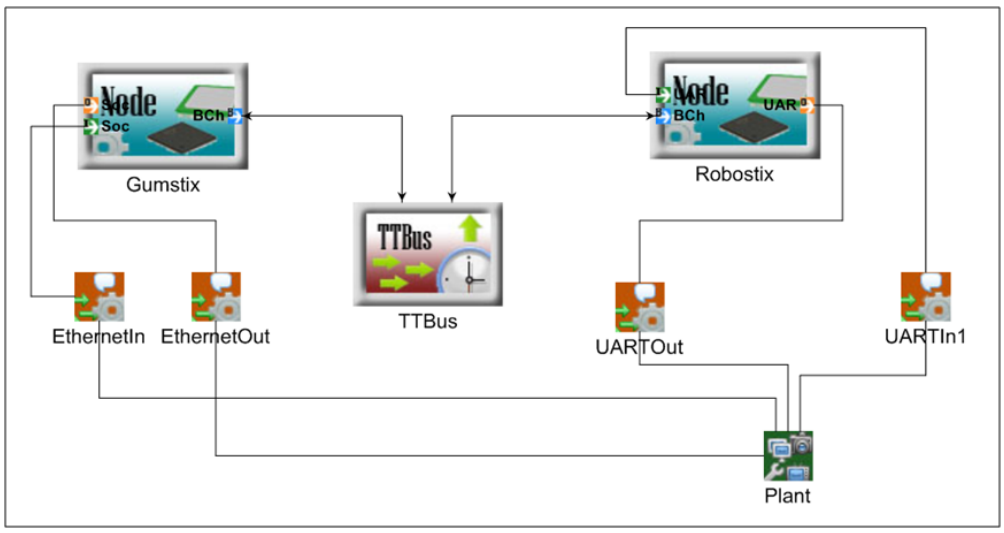


Figure 6: Overall hardware layout for the quadrotor example.

to future work in the ESMoL tool suite.

Hardware Architecture (HwA)

Fig. 6 illustrates the example platform model. The quadrotor architecture is deployed to a small embedded processor assembly manufactured by Gumstix, Inc. The outer loop position control is handled by an Intel PXA ARM processor (the Gumstix board), and attitude control and vehicle I/O are handled by an Atmel Atmega128 AVR processor (the Robostix board). The I/O occurs over serial connections to the sensors and motor actuators. The serial devices reside within the processor, and are modeled in the diagram as objects connecting the input and output ports on the processor to the object representing the plant dynamics. The two processors communicate via a synchronous I^2C bus which runs a software emulated time-triggered protocol.

A simple platform definition language (Fig. 2) contains relationships and attributes describing time-triggered networks. The models contain network topology and parameters to describe behavioral quantities like data rates and bus transfer setup times. Platforms are defined hierarchically as hardware units with ports for interconnections. Primitive components include processing nodes and communication buses. Behavioral semantics for these networks come from the underlying time-triggered architecture. The time-triggered platform provides services such as deterministic execution of replicated components and timed message-passing. Model attributes for hardware also capture timing resolution, overhead parameters for data transfers, and task context switching times.

Architecture Language

Logical architecture, deployment, and timing/execution models represent different design aspects for the same set of component instances. GME allows us to define the language in such a way that these three model aspects are simply different views of the same set of model elements. Together, the information in the three aspects define a model which is complete with respect to scheduling analysis, platform-specific simulation, and code generation.

System design models defined in the architecture language do not necessarily represent complete designs. For simple designs (such as the quadrotor example) a single architecture model can capture all of the details of the software model. More complex designs require an additional layer of organization which is not described here. It suffices to say that designs represented in the ESMoL Architecture language can be considered

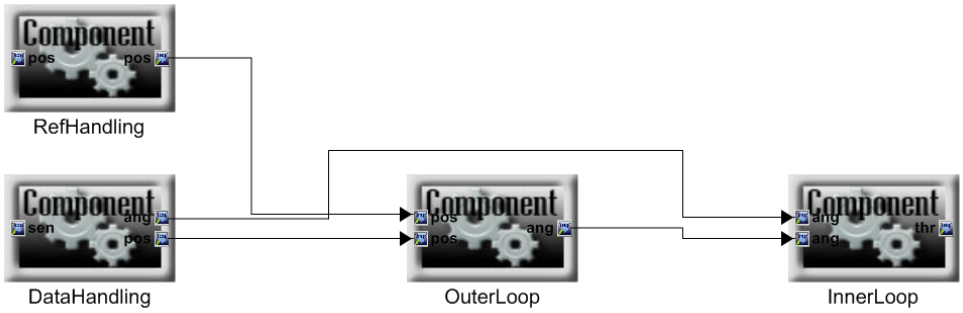


Figure 7: Quadrotor architecture model, Logical Architecture aspect.

as fragments which can be assembled into more complex structures. This is an active area of research for our ESMoL modeling efforts, as the higher-level architecture models should also account for fault modeling, evaluation, and performance issues.

- **Logical Software Architecture (SwA) Aspect** Fig. 7 portrays an ESMoL model example specifying logical data dependencies between quadrotor software component instances, independent of their distribution over different processors. The software architecture model describes the logical dataflow dependency relationships between component instances. Semantics for SwA Connections are those of task-local synchronous function invocations (with shared memory messaging) or message transfers between remote tasks using time-triggered communication. In this model the interpretations for the dependency links have not been specified. Those details appear in the deployment model.

  For the quadrotor, the *RefHandler* and *DataHandler* components receive and process data from the sensors. They pass their formatted data to the respective control blocks. The *OuterLoop* calculates an attitude reference to achieve the requested position. The *InnerLoop* issues thrust commands to achieve the requested attitude.

  In a design model, creation of a (GME) reference object to one of the component types corresponds to instantiation. Fig. 8 illustrates this idea. Using the same controller components along with a few new components to implement voting logic, we have specified the logical architecture for a triply-redundant version of the quadrotor model. Each ESMoL component type is used multiple times in a single design, expanding the model structure far beyond the size and scope of the original Simulink design. This particular model diagram is only shown to illustrate the instantiation mechanism.

- **Deployment Models (SY, DPL)** Fig. 9 displays the deployment model – the mapping of software components to processing nodes, and data messages to communication ports. Two of the four components are mapped to each of the two processors. For the quadrotor, the *RefHandler* and *OuterLoop* tasks run on the Gumstix processor. The *InnerLoop* and *DataHandler* tasks run on the Robostix processor. *RefHandler* receives position commands from a socket connection. *DataHandler* receives sensor data from a UART channel (a processor port in the model diagram). Position and attitude data are exchanged over the time-triggered bus, so the corresponding message ports are connected to bus channel objects on their respective processors. *InnerLoop* sends thrust commands through a UART channel, hence the connection to the appropriate processor port.

  In the figure the dashed connection from a component to a node reference represents an assignment of that component to run as a task on the node. The port connections represent the hardware channel through which that particular message will travel. Remote message dependencies are assigned to bus channels on the node. Local data dependencies are not specified here, as they are represented in the logical architecture. *IChan* and *OChan* port objects on a node can also be connected to message objects on a component. These connections represent the flow of data from the physical environment

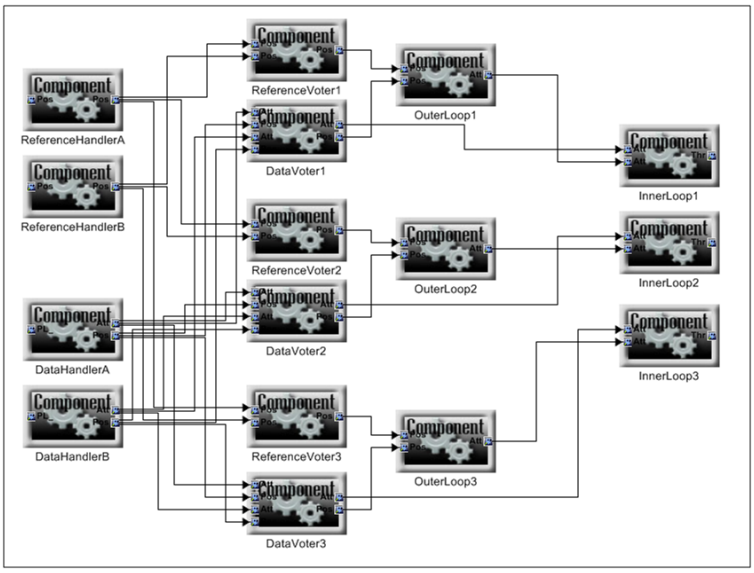


Figure 8: Triply-redundant quadrotor logical architecture. This is not part of the actual quadrotor model, and is only given for illustration.

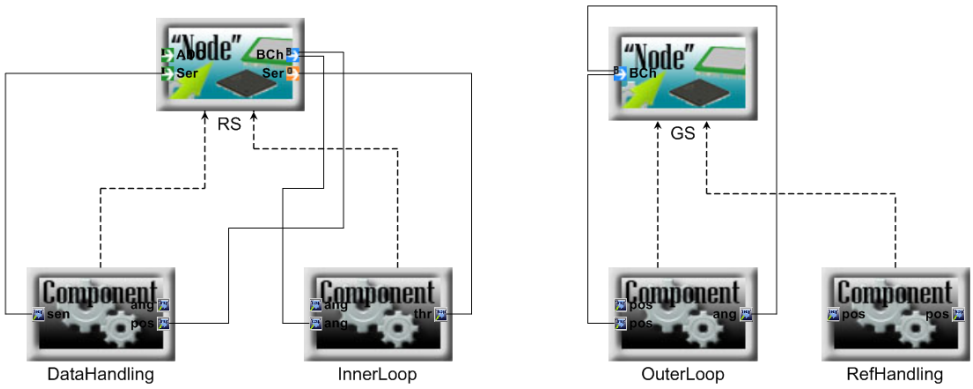


Figure 9: Quadrotor architecture model, Deployment aspect.

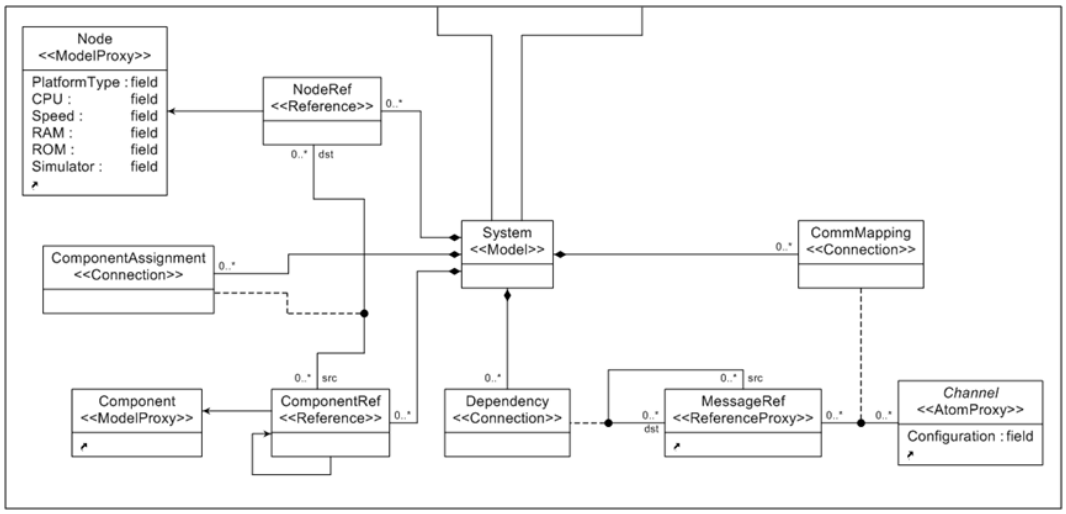


Figure 10: Details from the deployment sublanguage.

through sensors (*IChan* objects) or the flow of data back to the environment through actuators (*OChan* objects). Model interpreters use deployment models to generate platform-specific task wrapping and communication code as well as scheduling problem specifications.

The metamodel in Fig. 10 illustrates the classes and relationships for both the logical architecture connections and the deployment mapping. GME metamodels have a separate visualization aspect that allows us to define aspects in ESMoL and indicate which classes and connections should be visible in each aspect. *ComponentRef* objects are software component instances, and are visible in both aspects. In the logical architecture aspect, *Dependency* connectors define message transfers between component instance ports. The ports represent interfaces for each component instance. For the deployment aspect we add *NodeRef* objects (node references) and connectors (*ComponentAssignment* and *CommMapping*) to identify the mapping of tasks and messages to the platform model.

The deployment aspect captures the assignment of component instances as periodic tasks running on a particular processor. In ESMoL a task executes on a processing node at a single periodic rate. All components within the task execute synchronously. Data sent between tasks take the form of messages in the model. For data movement, the runtime provides logical execution time semantics found in time-triggered languages such as Giotto [28] – message transfers are scheduled after the deadline of a sending task, but before the release of the receiving tasks. Tasks never block, but execute with whatever data is available for each period.

- **Timing Models** Fig. 11 shows the quadrotor timing and execution model, where the designer attaches timing parameter blocks (of type *TTExecInfo*) to components and messages. *TTExecInfo* block configuration parameters include execution period and worst-case execution time. In the quadrotor model all task and message transfers are timed. The quadrotor data network runs at a rate of $20ms$. Particular timings for tasks and data transfers will be discussed below in the evaluation discussion.

  The timing sublanguage (Fig. 12) allows the designer to specify component execution constraints. Individual components can be annotated with timing objects that indicate whether they should be executed strictly (i.e., via statically scheduled time-triggered means), or as periodic real-time or sporadic tasks. Messages are similarly annotated. The annotation objects contain parameters such as period and worst-case execution time that must be given by the designer. Automated scheduling analysis fills in the schedule fields.

  The execution model also indicates which components and messages will be scheduled independently, and which will be grouped into a single task or message object. The time order of the message writer

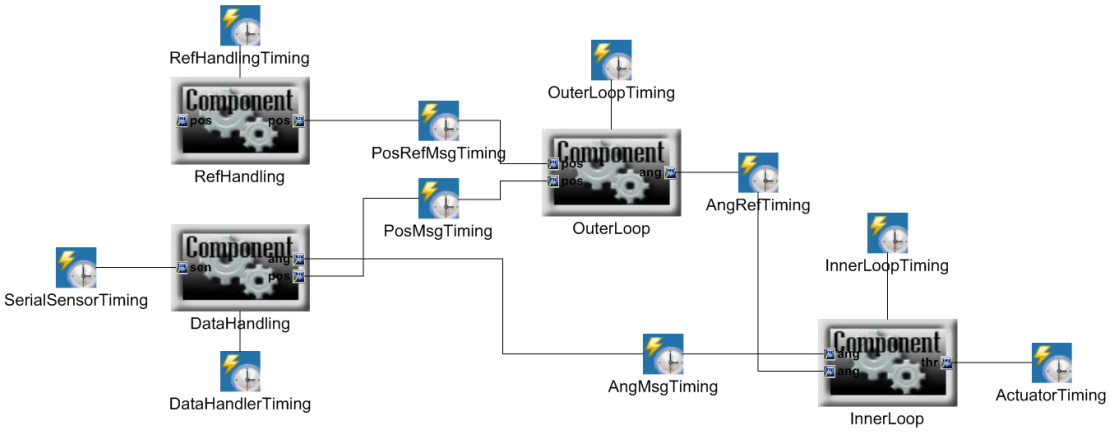


Figure 11: Quadrotor architecture model, Timing aspect.

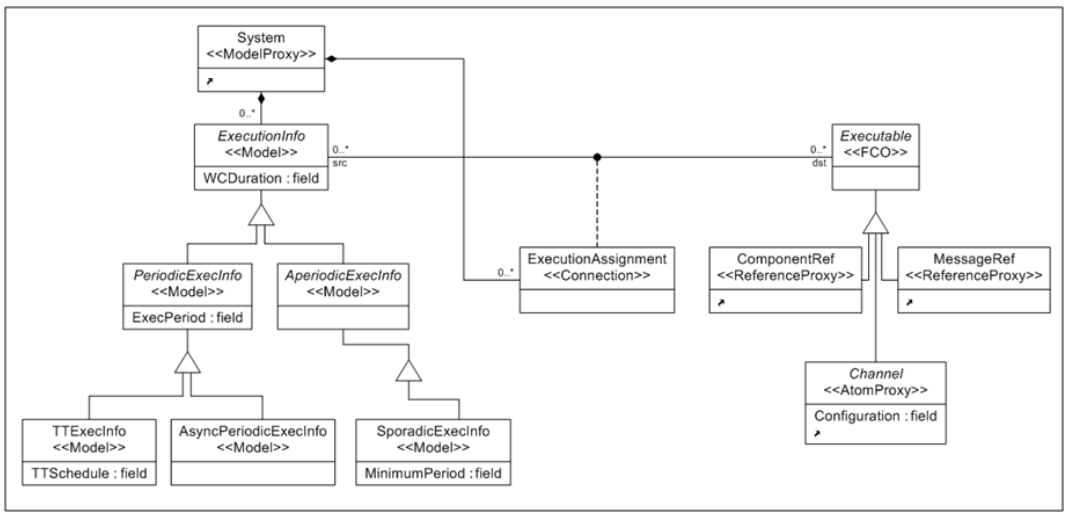


Figure 12: Details from timing sublanguage.

and readers are enforced by the static schedule. The locality of a message transfer is specified in the logical architecture and deployment aspects. In the case of processor-local data transfers, transfer time is neglected – reads and writes occur in locally shared memory. After a static schedule has been calculated, task and message release times are also stored in the timing objects.

Behavior of the deployed software components depends on the execution times of the functions on the platform, the calculated schedule, and coordination between distributed tasks. The calculated static execution schedule can be used to simulate the control design with additional delays to assess the impact of the platform on performance.

5 Integrating Tools with ESMoL

Figure 1 depicts a design flow that includes a user-facing modeling language for design and an abstract intermediate language for supporting interpreter development and maintenance. A completed ESMoL model is transformed (via the Stage 1 transformation, Step 6 in the figure) into a model in the ESMoL_Abstract language, where all implied relationships and structural model inferences have been resolved. Model interpreters for calculating time-triggered schedules, creating platform-specific simulations, and generating deployable code are integrated using the Stage 2 transformation.

Rather than designing a user-friendly graphical modeling language and directly attaching translators to analysis tools, we created a simpler abstract intermediate language whose elements are similar to those of the user language. The first model transformation flattens the user model into the abstract intermediate form, translating parameters and resolving special cases as needed. Generators for code and analysis are attached to the abstract modeling layer, so the simpler second-stage transformations are easier to maintain, and are isolated from changes to the user language.

In the model integrated computing approach, domain specific modeling languages represent different aspects of the design, with the aim of consistently integrating different concepts and details for those design aspects and integrated analysis tools. Our tools enforce a single view of structural inference in the design model. We will cover some of the transformation details to illustrate this concept. This approach can be considered as an implementation of the tool integration ideas in [29], but with variations of the details included in the design language.

5.1 Stage 1 Transformation

Stage 1 translates ESMoL models into an abstract intermediate language that contains explicit relation objects that represent relationships implied by structures in ESMoL (Fig. 13). This translation is similar to the way a compiler translates concrete syntax first to an abstract syntax tree, and then to intermediate semantic representations suitable for optimization. Stage 1 was implemented using the UDM model navigation API, and written in C++. The ESMoL_Abstract target model is the source for the transformations implemented in Stage 2.

Each analysis translation works from a single view of the design model, simplifying the implementations of tool-specific translations. As an example, consider the model shown in Fig. 7. Component *DataHandler* sends data messages to the other two components, as denoted by the dependency arrows. The deployment view (Fig. 9) shows that each component executes on a different processor. Locally, the port object on each component (in both diagrams) represents the component's view of the data message sent over the wire. The solid connections in the deployment diagram indicate which device on the processing node will be used to transfer the data. Specified messages will participate in processor-local synchronous data flows, or time-triggered exchanges over the network. All of these connections and entities are related to a single semantic message object, which is related to other elements in different parts of the user model (see the FormattedData message in Fig. 18). The execution aspect contains timing information objects, which provide information for fully specifying the various data transfers.

The first stage transformation checks constraints to ensure that each object is used correctly throughout the design, ensuring well-formedness. The Stage 1 transformation then reduces this complex set of relations

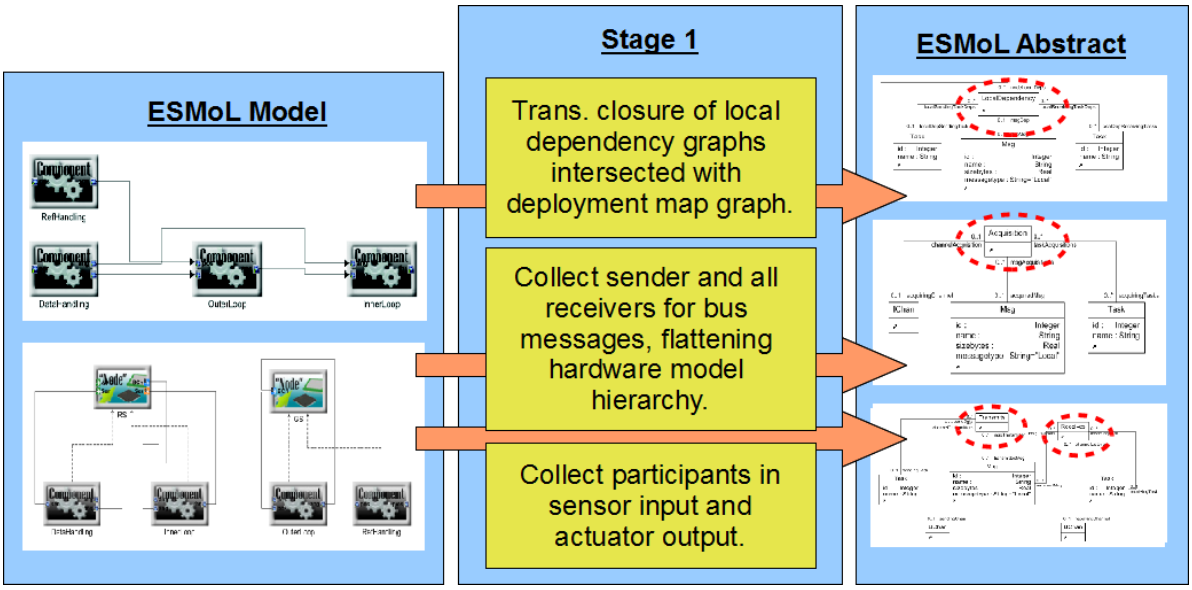


Figure 13: Stage 1 Transformation.

Specified ESMoL Relation Sets	ESMoL_Abstract Relation
$CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$ $AC_{id_{Ch}} = \{(obj_{IChan}, obj_{MsgInst}) \mid id(obj_{IChan}) = id_{Ch}\}$ $NC_{id_N} = \{(obj_{Node}, obj_{IChan}) \mid id(obj_{Node}) = id_N \wedge parent(obj_{IChan}) = obj_{Node}\}$ $CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$	$Acq = \{(obj_{MsgInst}, obj_{CompInst}, obj_N, obj_{Ch}) \mid (obj_N, obj_{CompInst}) \in CA_{id_N} \wedge (obj_{Ch}, obj_{MsgInst}) \in AC_{id_{Ch}} \wedge (obj_N, obj_{IChan}) \in NC_{id_N} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$

Table 1: Acquisition relation transformation details.

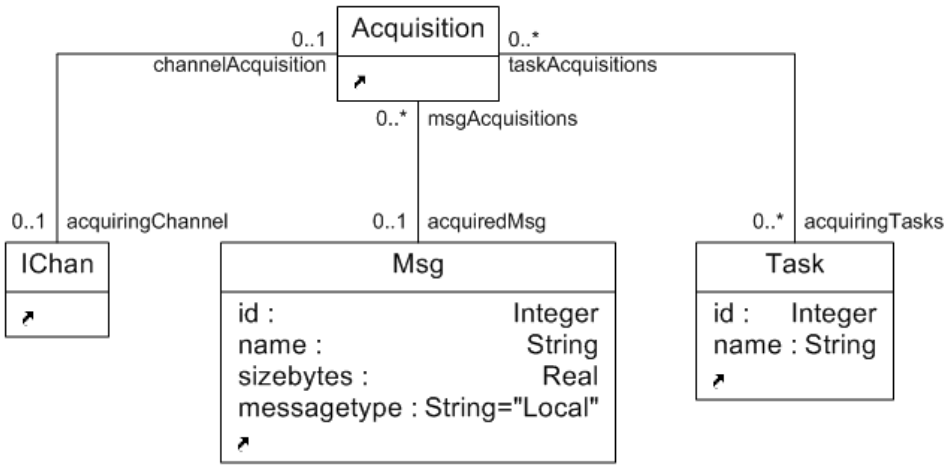


Figure 14: Acquisition relation in ESMoL Abstract, representing the timed flow of data arriving from the environment.

to a single message object with relations to the other objects that use it. Timing parameters from the platform model are used to calculate a behavioral model for messages and components, including component start times, message transfer times, and the duration of each message on the bus.

We describe here some of the transformations of user-facing ESMoL language objects and relations to a more compact set of relations that simplify generation of design artifacts from the model. The most direct example of such a semantic assumption is the single-message abstraction. Data transfers between the functional code and the message fields must be compatible. We enforce compatibility both by constraint checking, and by the use of a single ESMoL_Abstract message instance object for all participants in the data interchange. The *Signal* object in the abstract graph represents the transfer of a single datum to or from the message. For simplicity and clarity we will not show the *Signal* objects in the diagrams, as they are numerous.

The transformations described here capture different forms of the single-message transformation. This is not a complete description of the entire first stage transformation, but provides a representative subset for illustration.

In the formal descriptions below, Obj_{Type} (capitalized) is the set of objects of type $Type$, and obj_{Type} (lowercase) is an instance from that set. We also use two functions $id : Obj_{Type} \rightarrow \mathbf{Z}^+$ for a unique identifier of an object, and $parent : Obj_{Type1} \rightarrow Obj_{Type2}$ to find the parent (defined by a containment relation in the model of an object). The parent relation is unique.

Acquisition: From the Environment to Data

In ESMoL_Abstract *Acquisition* objects relate all of the different model entities (and therefore, their design parameters) that participate in the collection of data from an input device such as an analog to digital converter or serial link. The Stage1 transformation enforces certain cardinality constraints to ensure the validity of this transformation – for example, each message instance is related to exactly one sender and possibly multiple receivers. A message relationship can be implied by different types of connections in ESMoL, so Stage1 must determine that only one such relationship exists.

The ESMoL relations shown in Table 1 are described as follows:

- CA **ComponentAssignment**: (the dashed connection shown in Fig. 9) assigns a task to run on a particular processor (id_N).
- AC **AcquisitionConnection**: (the directed connection from processor object ports to component

Specified ESMoL Relation Sets	ESMoL_Abstract Relation
$CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$	
$AC_{id_{Ch}} = \{(obj_{OChan}, obj_{MsgInst}) \mid id(obj_{OChan}) = id_{Ch}\}$	$Act = \{(obj_{MsgInst}, obj_{CompInst}, obj_N, obj_{Ch}) \mid (obj_N, obj_{CompInst}) \in CA_{id_N} \wedge (obj_{Ch}, obj_{MsgInst}) \in AC_{id_{Ch}} \wedge (obj_N, obj_{OChan}) \in NC_{id_N} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$
$NC_{id_N} = \{(obj_{Node}, obj_{OChan}) \mid id(obj_{Node}) = id_N \wedge parent(obj_{OChan}) = obj_{Node}\}$	
$CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$	

Table 2: Actuation relation transformation details.

message ports) assigns a hardware input peripheral data channel (modeled as an object of type $ICHan$) to a data-compatible message structure in the component.

- NC: Containment relationship of the channel object (port) in the Node object.
- CC: Containment of the message instance object (port) in the component instance object.

The metalanguage for ESMoL_Abstract captures the structural semantic reductions shown in Table 1 in a compact form (see Fig. 14), so that all of the consumers of the input data get the same consistent structural view of the model. This transformation takes the ESMoL objects described in the left column of the table and produces a single relation for each collection representing an ESMoL_Abstract data acquisition specification. The modeling tools provide a programming interface for traversing, reading, and editing the models. The collected relations are also more efficiently processed by synthesis interpreters, as they avoid extra traversals to gather the objects.

Actuation: From Data to the Environment

The transformation to an Actuation object is nearly identical to that of the Acquisition transformation, but the data direction, cardinalities, and types involved are different. The chief difference is that actuation objects can only have one associated task, where acquisition data may be broadcast to multiple tasks. Table 2 gives the details of the transformation from relations in ESMoL to the actuation relation in ESMoL_Abstract. Fig. 15 shows the structure of the resulting classes in ESMoL_Abstract.

Local Dependencies: Data Movement within Nodes Local dependencies represent not only direct data dependencies between nodes on a particular processor, but also implied dependencies through remote data transfer chains starting and ending on the same processor. This is modeled as the set LD_{TC} of all pairs in the transitive closure of dependencies starting with the message instance $obj_{MsgInst1}$. The collected set of local dependencies (*Locals*) intersects this set with those message instances contained in components on the current processing node (i.e. from the set CA_{id_N}). Table 3 gives the transformation details.

Bus Transfers: Data Movement Between Nodes Bus transfers are slightly more complicated, as they involve two or more endpoints. Table 4 and Fig. 17 contain the details. The send and receive relations are modeled separately as they have different cardinalities (one sender and possibly multiple receivers). The platform-specific code generators produce separate files for each processor (recall that the network may be heterogeneous). Fig. 18 shows an example of the objects and parameters based on our design example. The object diagram is an instance of the abstract language constructs shown in Figs. 14, 16, and 17. The diagram depicts ESMoL_Abstract relations of type *Acquisition*, *LocalDependency*, *Transmits*, and *Receives*. These objects are involved in collecting position data from the sensors (task *DataHandler* from data channel

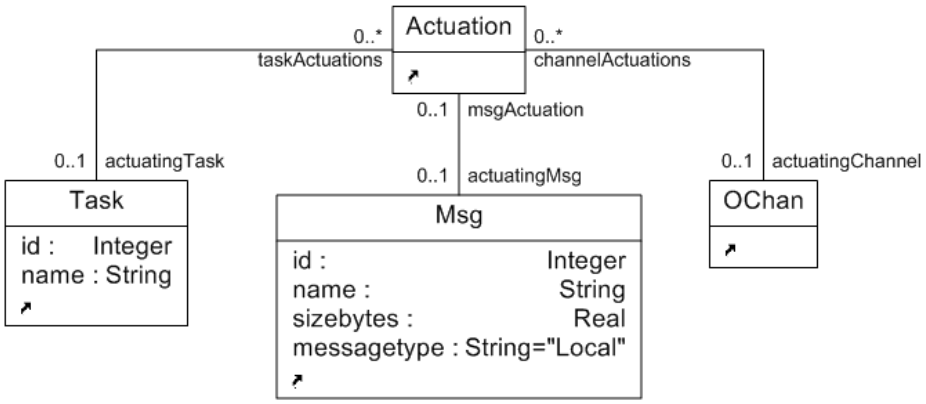


Figure 15: Actuation relation in ESMoL Abstract, representing the timed flow of data back into the environment.

Specified ESMoL Relation Sets	**ESMoL_Abstract Relation**
$CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$ $LD_{id_1} = \{(obj_{MsgInst1}, obj_{MsgInst}) \mid id(obj_{MsgInst1}) = id_1\}$ $LD_{TC} = \{(obj_{MsgInstj}, obj_{MsgInstj+1}) \mid$ *in the sequence* $((obj_{MsgInst1}, obj_{MsgInst2}) \in LD_{id_1}, (obj_{MsgInst2}, obj_{MsgInst3}) \in LD_{id_2}, \ldots (obj_{MsgInstj}, obj_{MsgInstj+1}) \in LD_{id_j})\}$ $CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$	$Locals = \{(obj_{MsgInst1}, obj_{CompInst1}, obj_{MsgInst2}, obj_{CompInst2}, obj_N) \mid (obj_N, obj_{CompInst1}) \in CA_{id_N} \wedge (obj_N, obj_{CompInst2}) \in CA_{id_N} \wedge (obj_{MsgInst1}, obj_{MsgInst2}) \in LD_{TC} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$

Table 3: Local (processor-local) data dependency relation.

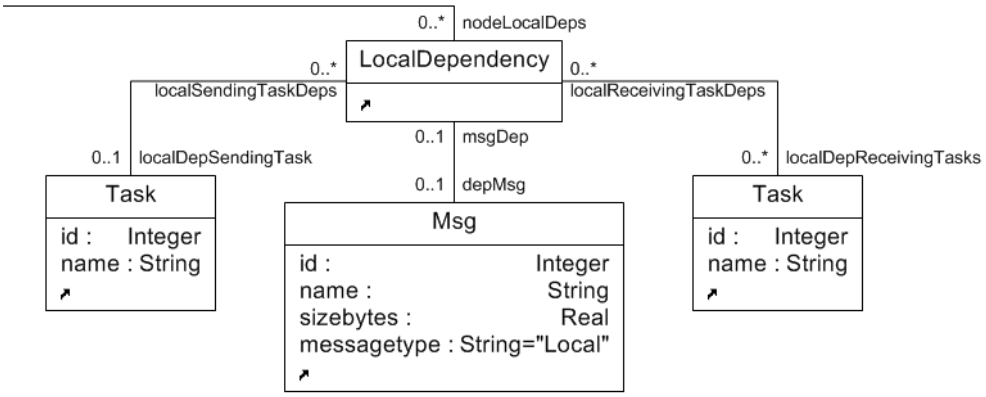


Figure 16: Local dependency relation in ESMoL Abstract, representing data transfers between components on the same processing node.

Specified ESMoL Relation Sets	**ESMoL_Abstract Relation**
$CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$ $AC_{id_{Ch}} = \{(obj_{MsgInst}, obj_{BChan}) \mid id(obj_{BChan}) = id_{Ch}\}$ $NC_{id_N} = \{(obj_{Node}, obj_{BChan}) \mid id(obj_{Node}) = id_N \wedge parent(obj_{BChan}) = obj_{Node}\}$ $CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$	$Trn = \{(obj_{MsgInst}, obj_{CompInst}, obj_N, obj_{Ch}) \mid (obj_N, obj_{CompInst}) \in CA_{id_N} \wedge (obj_{Ch}, obj_{MsgInst}) \in AC_{id_{Ch}} \wedge (obj_N, obj_{BChan}) \in NC_{id_N} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$

Table 4: Transmit relation transformation details. This represents the sender side of a remote data transfer between components.

Specified ESMoL Relation Sets	Semantic Construct
$CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$ $AC_{id_{Ch}} = \{(obj_{BChan}, obj_{MsgInst}) \mid id(obj_{BChan}) = id_{Ch}\}$ $NC_{id_N} = \{(obj_{Node}, obj_{BChan}) \mid id(obj_{Node}) = id_N \wedge parent(obj_{BChan}) = obj_{Node}\}$ $CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$	$Rcv = \{(obj_{MsgInst}, obj_{CompInst}, obj_N, obj_{Ch}) \mid (obj_N, obj_{CompInst}) \in CA_{id_N} \wedge (obj_{Ch}, obj_{MsgInst}) \in AC_{id_{Ch}} \wedge (obj_N, obj_{BChan}) \in NC_{id_N} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$

Table 5: Receive relation transformation details.

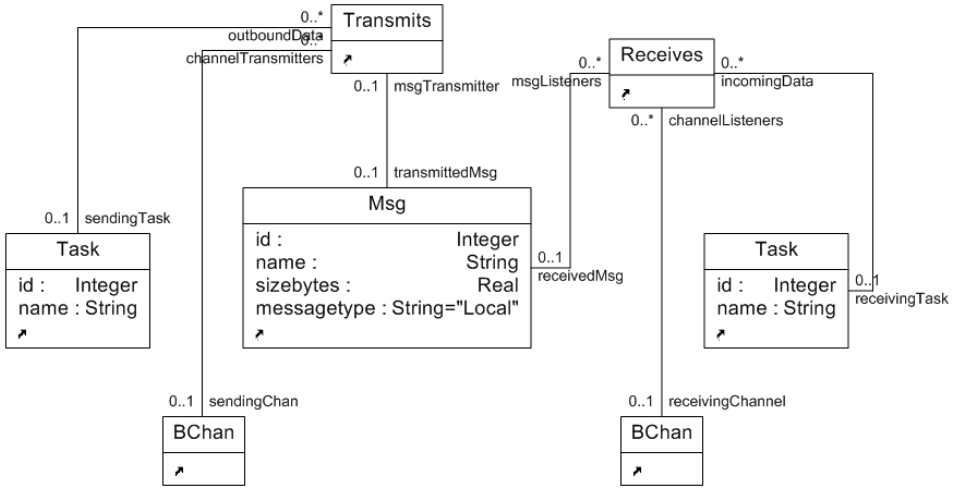


Figure 17: Transmit and receive relations in ESMoL Abstract, representing the endpoints of data transfers between nodes.

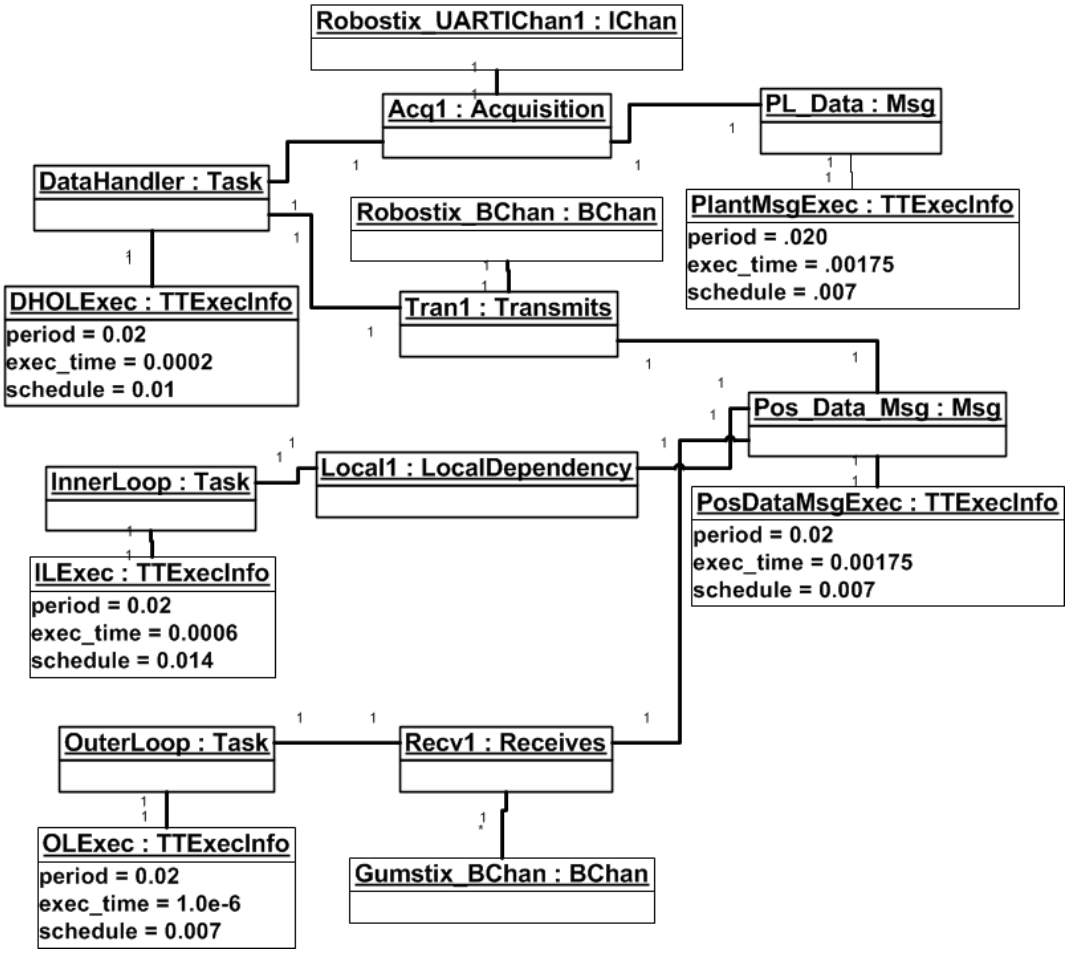


Figure 18: Object diagram from part of the message structure example from Figs. 7 and 9.

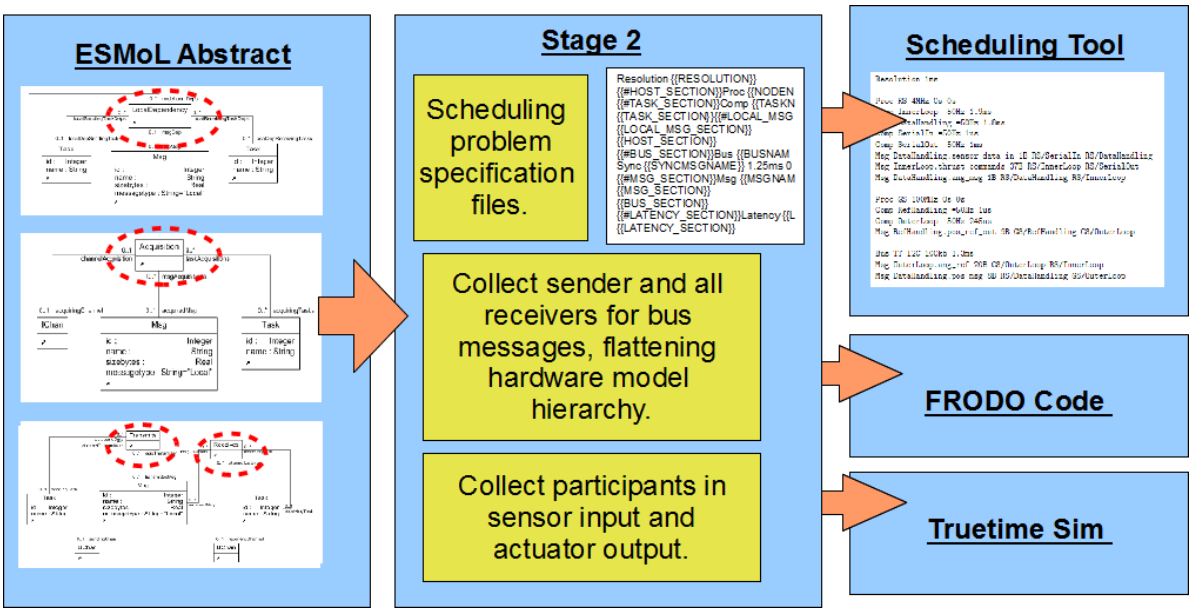


Figure 19: Stage 2 Interpreter.

Robostix_UARTChan1), and then redistributing it locally to the InnerLoop task as well as remotely to the *OuterLoop* task through the bus channel interfaces on the Robostix and Gumstix nodes.

5.2 Stage 2 Transformation Outputs: Analysis Models and Code

Stage 2 generates analysis models and code from ESMoL_Abstract models (Fig. 19). To perform the actual generation of analysis models and code, we use the CTemplate library[30] called from C++. The current Stage 2 interpreter is generally used in a particular sequence:

1. Generation of the scheduler specification.
2. Creation of a TrueTime simulation model.
3. Generation of platform-specific code using the FRODO virtual machine API.

We will cover details for generation of scheduling problem specifications and FRODO-specific code. The TrueTime code generation is documented elsewhere[7].

Scheduling Problem Generation

The control design models provide task period configurations, and either profiling or static analysis provides worst-case execution time parameters for each component instance. Data transfer rates and overhead parameters for communication buses are stored in the platform model. [8] describes the mapping of model structure, execution information, and platform parameters into actual constraint model details, extending earlier work on constraint-based schedule calculation[31]. The Gecode constraint programming tool [32] solves these constraints for task release and message transfer times on the time-triggered platform. The scheduling process guarantees that the implementation meets the timing requirements required by the control design process.

Fig. 20 portrays the steps a model transformation takes while distilling details from ESMoL and creating a scheduling problem model whose syntax represents the proper sets of behaviors. If the schedule is feasible, task and message release time results are fed back into the ESMoL model as configuration parameters. We describe the steps indicated in the diagram here:

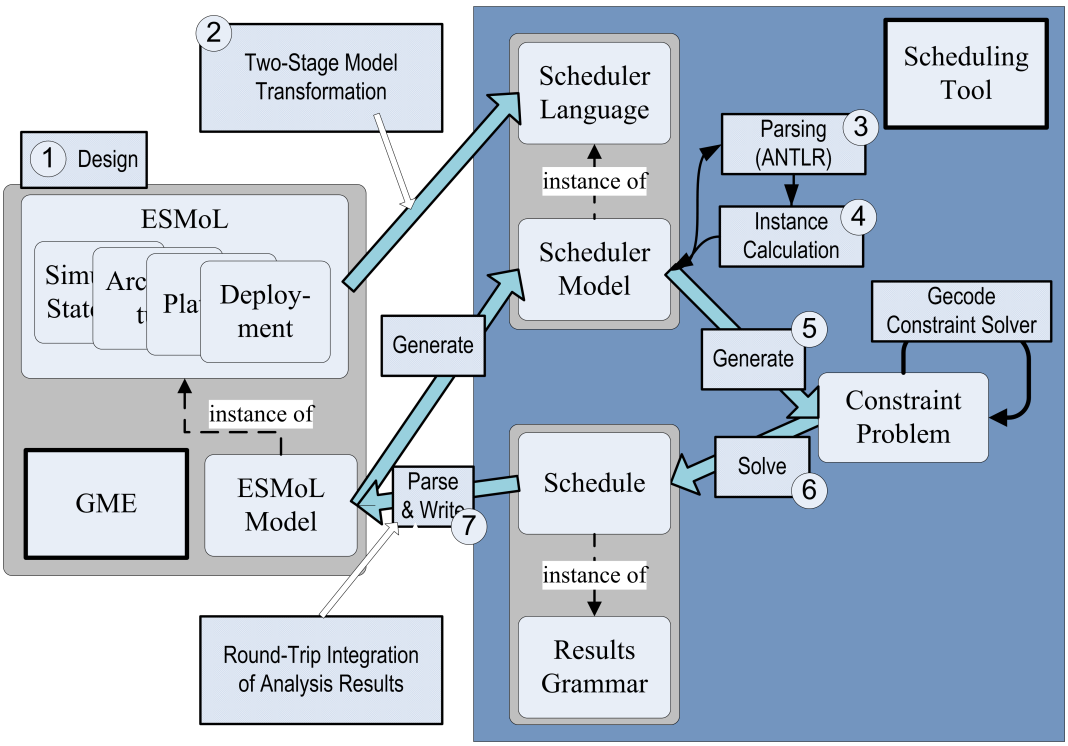


Figure 20: Integration of the scheduling model by round-trip structural transformation between the language of the modeling tools and the analysis language.

1. We start with a design model specified using ESMoL.
2. The two-stage transformation converts the model to an equivalent model in ESMoL_Abstract, and then invokes the templates to generate a scheduling problem specification.
3. We invoke the scheduling tool, which performs the following steps:
   (a) Parses the problem specification to import the model into the constraint generation environment.
   (b) Calculates the hyperperiod length to determine the number of instances required for each task and message.
   (c) Translates task and message instance relationships into constraints in Gecode (as described in [8]).
   (d) Solves the constraint problem, possibly indicating infeasibility.
   (e) If a valid schedule results, it is written out to a file.
4. The results are imported into the ESMoL model and written to the appropriate objects.

Table 6 contains the distributed schedule specification for our quadrotor example, including the following elements:

- **Resolution** (seconds) specifies the size of a single processing tick for the global schedule. This should correspond to the largest measurable time tick (quantum) of the processors in the network. All tasks and messages in the schedule timeline are discretized to this resolution.
- **Proc** specifies a processing node. Parameters are name, processor speed (Hz), and message send/receive overhead times (these default to zero seconds if unspecified). Processor names must be unique.
- **Comp** (or task) belongs to the most recently specified processor. A component is characterized by its name, period, and worst-case execution time (WCET) (both in seconds). We do not address the manner in which the WCET is to be obtained.
- **Bus** specifies a bus object, characterized by name, transfer speed (bits per second), and transfer overhead (also in seconds).
- **Msg** includes a name, byte length, sending task, and list of receiving tasks.

Task and message names are unique only within their scope (processor or bus, respectively). When used in other scopes they are qualified with their scope as shown (e.g. P3/T1). The timing constraints include the various platform overhead parameters. For example, once the message length is converted from bytes to time on the bus, we add the transfer overhead to represent the setup time for the particular protocol. Engineers must measure or estimate platform behavioral parameters and include them in models for the platform[2].

Scheduling specifications are created in the Stage 2 interpreter from the template shown in Table 7. The Stage 2 scheduler generation logic traverses the ESMoL_Abstract model and fills in the structures which are used to fill in the template when the CTemplate generator is invoked. In CTemplate, each `{{#...}} {{/...}}` tag pair delimits a section which can be repeated by filling in the proper data structure in the code. The other tags `{{...}}` are replaced by the string specified in the generation code.

Producing the *Proc* and *Comp* lines from the model API is straightforward as the output mirrors the model hierarchy, so these lines require only simple traversals of the model. Each generated line uses parameters from the respective model object to fill in the blanks. The parameters are shown only in the generated output, though the object diagram in Fig. 18 illustrates a good example of parameter layout and disposition. In order to produce each *Msg* line, many relations must be collected (as shown in Table 3 and Fig. 16) and distilled into the right relationships. This requires more complex traversal code often involving multiple

```
Resolution 1ms

Proc RS 4MHz 0s 0s
Comp InnerLoop =50Hz 1.9ms
Comp DataHandling =50Hz 1.8ms
Comp SerialIn =50Hz 1us
Comp SerialOut =50Hz 1ms
Msg DataHandling.sensor_data_in 1B RS/SerialIn RS/DataHandling
Msg InnerLoop.thrust_commands 37B RS/InnerLoop RS/SerialOut
Msg DataHandling.ang_msg 1B RS/DataHandling RS/InnerLoop

Proc GS 100MHz 0s 0s
Comp RefHandling =50Hz 1us
Comp OuterLoop =50Hz 245us
Msg RefHandling.pos_ref_out 9B GS/RefHandling GS/OuterLoop

Bus TT_I2C 100kb 1.3ms
Msg OuterLoop.ang_ref 20B GS/OuterLoop RS/InnerLoop
Msg DataHandling.pos_msg 8B RS/DataHandling GS/OuterLoop
```

Table 6: Scheduling spec for the Quadrotor example.

```
Resolution {{RESOLUTION}}

{{#HOST_SECTION}}Proc {{NODENAME}} {{NODEFREQ}} {{SENDOHD}} {{RECVOHD}}
{{#TASK_SECTION}}Comp {{TASKNAME}} ={{FREQUENCY}} {{WCEXECTIME}}
{{TASK_SECTION}}{{#LOCAL_MSG_SECTION}}Msg {{MSGNAME}} {{MSGSIZE}} {{SENDTASK}} {{RECVTASKS}}
{{LOCAL_MSG_SECTION}}
{{HOST_SECTION}}
{{#BUS_SECTION}}Bus {{BUSNAME}} {{BUSRATE}} {{SETUPTIME}} {{#BUS_HOST_SECTION}}{{NODENAME}} {{BUS_HOST_SECTION}}
{{#MSG_SECTION}}Msg {{MSGNAME}} {{MSGSIZE}} {{SENDTASK}} {{RECVTASKS}}
{{MSG_SECTION}}
{{BUS_SECTION}}
{{#LATENCY_SECTION}}Latency {{LATENCY}} {{SENDTASK}} {{RECVTASK}}
{{LATENCY_SECTION}}
```

Table 7: Stage 2 Interpreter Template for the Scheduling Specification

passes through the model objects. To write a new generator similar to this one, the developer uses the interpreter API and the transformed abstract syntax graph model. In the abstract language traversal we collect the *LocalDependency* objects and filter them by processor. Each *LocalDependency* object contains all of the information necessary to fill out the parameters in the template and create a new Msg line in the scheduler specification file (within the proper *Proc* scope).

While we do not list here the details related to the solution of scheduling specifications, it may be useful to document some of the scheduler limitations. More details regarding these limitations may be found in [8].

- We do not support preemptive scheduling of tasks or messages, as our runtime provides conflict-free task execution and data communication during nominal operation.
- The overhead parameters may be an overly simplistic model for some cases. Each processor and bus pair may have different parameters, depending on the bus type and the protocol used.
- We do not perform optimization on the schedule, so performance cost functions are not taken into account. For control problems where the execution time changes yield irregular performance changes, this is a more serious issue (see for example [33]).

Platform-Specific Code Generation

Time-triggered execution requires configuration with the computed cyclic schedule. Code generated for the virtual machine conforms to a particular synchronous execution strategy – each task reads its input variables, invokes its component functions, and writes its output variables. The schedule calculation assumes logical execution time semantics, where task input data is ready before task release, and output data is not assumed ready before the task completes[12]. Data structures describe the invocation times and configuration parameters for tasks and messages on each processor. Each message configuration instance also includes local buffer addresses where the timed communication controller in the virtual machine can store incoming and outgoing message data.

The generated code for the Quadrotor model in Table 8 was produced from the template description for the platform-specific code generator in Table 9. The FRODO virtual machine generation template brings together all of the ESMoL_Abstract relations described in the earlier section. The template and generated code segment above correspond to the second-stage interpreter that creates the static schedule structures used by the virtual machine. The tasks, messages, and peripherals listed here come from the *Acquisition*, *Actuation*, *Transmit*, and *Receive* relation objects. The various connected objects are sorted according to schedule time, and then the template instantiation uses the object parameters to create the tables in a manner similar to that described for the scheduler specification generation above. The *LocalDependency* relations do not appear in this template. The scheduler creates constraints that must be satisfied for each local dependency, but local message transfers take place automatically in shared memory as tasks write to and read from processor-local message structures. Therefore, any valid task and message schedule will satisfy them. In a different part of the FRODO template, the local dependencies determine which message fields must be used as arguments to the component function calls (not shown here).

```
////////////////////////////// SCHEDULE TABLE ////////////////////////////////

portTickType hp_len = 20;

task_entry tasks[] = {
   { DataHandling, "DataHandling", 4, 0},
   { InnerLoop, "InnerLoop", 9, 0},
   {NULL, NULL, 0, 0}
}

msg_entry msgs[] = {
   { 1, MSG_DIR_RECV, sizeof( OuterLoop_ang_ref ),
      (portCHAR *) & OuterLoop_ang_ref,
      (portCHAR *) OuterLoop_ang_ref_c, 7, 0, 0},
   { 2, MSG_DIR_SEND, sizeof( DataHandling_pos_msg ),
      (portCHAR *) & DataHandling_pos_msg,
      (portCHAR *) DataHandling_pos_msg_c, 11, 0, 0},
   { -1, 0, 0, NULL, NULL, 0, 0, 0}
}

per_entry pers[] = {
   { 1, "UART", IN, 0, 0, sizeof( DataHandling_sensor_data_in ),
      (portCHAR *) & DataHandling_sensor_data_in,
      (portCHAR *) DataHandling_sensor_data_in_c, 2, NULL, 0, 0},
   { 2, "UART", OUT, 0, 0, sizeof( InnerLoop_thrust_commands ),
      (portCHAR *) & InnerLoop_thrust_commands,
      (portCHAR *) InnerLoop_thrust_commands_c, 14, NULL, 0, 0},
   { -1, NULL, 0, 0, 0, 0, NULL, NULL, 0, NULL, 0, 0 }
}
```

Table 8: Generated code for the task wrappers and schedule structures of the Quadrotor model.

```
////////////////////////////// SCHEDULE TABLE ///////////////////////////////

portTickType hp_len = {{NODE_hyperperiod}};

{{#SCHEDULE_SECTION}}
task_entry tasks[] = {
{{#TASK}}
   { {{TASK_name}}, "{{TASK_name}}", {{TASK_startTime}}, 0},{{TASK}}
   {NULL, NULL, 0, 0}
}

msg_entry msgs[] = {
{{#MESSAGE_NAME}}
   { {{MESSAGE_index}}, {{MESSAGE_sendreceive}}, sizeof( {{MESSAGE_name}} ),
         (portCHAR *) & {{MESSAGE_name}},
         (portCHAR *) & {{MESSAGE_name}}_c, {{MESSAGE_startTime}},
         pdFALSE},
{{MESSAGE_NAME}}
   { -1, 0, 0, NULL, NULL, 0, 0}
}

per_entry pers[] = {
{{#PER_NAME}}
   { {{PER_index}}, "{{PER_type}}",
         {{PER_way}}, 0, {{PER_pin_number}}, sizeof( {{PER_name}} ),
         (portCHAR *) & {{PER_name}},
         (portCHAR *) & {{PER_name}}_c,
         {{PER_startTime}}, NULL},
{{PER_NAME}}
   { -1, NULL, 0, 0, 0, 0, NULL, NULL, 0, NULL }
}
{{SCHEDULE_SECTION}}
```

Table 9: Template for the virtual machine task wrapper code. The Stage 2 FRODO interpreter invokes this template to create the wrapper code shown in Table 8.

6 Synchronous Semantics

We will briefly present a formal argument for the preservation of synchronous Simulink block firing orders as we use the Simulink blocks to define software components, their deployment to the hardware, and impose a time-triggered execution schedule on the design. Our semantic argument is only valid for synchronous data flow (SDF) specifications. We do not claim to represent the full generality of Simulink specifications, rather we restrict ourselves to dataflow graphs without conditional execution. Our graphs must contain only tasks that execute in periodic, discrete time, have no delay-free loops, all delay elements must be initialized with a data token, and initial block firing orders must include the outputs of the delay elements This final assumption can be satisfied by considering the outputs of the delay elements as additional inputs to the component. Then all dependent blocks will be able to fire as early as necessary in the schedule. Our restrictions on execution are consistent with those required by the Mathworks Real-Time Workshop Embedded Coder product, which forces models to have fixed-step execution and task periods harmonic with the configured time step size for code generation.

Consider a synchronous acyclic graph $\mathbb{G} = (V, E)$ representing the connectivity of a Simulink dataflow model, where edges abstract the transfer of data between blocks (without the data type information, data capacity, or multiplicity). Let $exec : \mathbb{G} \to \mathbb{R}$ represent the task duration for vertices (obtained by analysis or measurement), and the communication message transfer time for edges.

Let $C_V \subseteq V \times \mathbb{Z}^{|V|}$ be the set of all possibly concurrent firing orders for the blocks represented by the set V which respect the partial order specified by the edge set E (i.e. $(v_1, v_2) \in E \Rightarrow c(v_1) < c(v_2) \forall c \in C_V$ where the pairs in c are interpreted as functions on V). Note that C_V should only be taken up to isomorphism, eliminating orderings that are equivalent.

Consider the synchronous execution of $\mathbb{G}$, where the full graph is executed on a periodic schedule at instants $\{T_s k\}, (k = 0, 1, \ldots, \infty)$, and completes each execution before the next cycle. For embedded code generation, Simulink requires models to execute with fixed-step semantics, so this is not an overly strong restriction.

Next, we allow manipulations of the dataflow graph $\mathbb{G}$ as follows: Let $\mathbb{G}' \subseteq \mathbb{G}$ be a subgraph. Assume that $\mathbb{G}' = (V', E')$ represents a well-formed functional dataflow. Let $C_{V'} \subseteq V' \times \mathbb{Z}^{|V'|}$ be a set of possible orderings for V' created by restricting C_V to the vertex set V'. All orderings in $C_{V'}$ are also valid in C_V, if we adjoin proper orderings from $C_{V-V'}$. All orderings in $C_{V'}$ are also synchronous orderings, as they respect the partial order defined by $\mathbb{G}'$. We can also continue this construction for products. Let $\mathbb{G}' \subseteq \mathbb{G}$ and $\mathbb{G}'' \subseteq \mathbb{G}$, where we uniquely identify the vertex sets V' and V'' so that $V' \cap V'' = \emptyset$. Then for $\mathbb{G}' \times \mathbb{G}''$ we have $C_{V'} \times C_{V''}$. Considering the concurrent execution of $\mathbb{G}'$ and $\mathbb{G}''$, both graphs execute synchronously if executed according to an order from $C_{V'} \times C_{V''}$.

Let $\mathbb{G}_{s_1}, \ldots, \mathbb{G}_{s_I}$ be subgraphs of a Simulink dataflow $\mathbb{G}$. These represent ESMoL-specified dataflows. Let $\mathbb{G}_s = \times_{i \in [1,I]} \mathbb{G}_{s_i}$ where each vertex $v \in V_s$ is given a unique identity as above. Consider the product of the restricted orderings $C_{V_s} = \times_{i \in [1,I]} (V_{s_i} \times \mathbb{Z}^{|V_{s_i}|})$. Then the specified dataflow $\mathbb{G}$ is synchronous if executed according to an order from C_{V_s}.

Consider the following partitions on $\mathbb{G}_s$:

Let $C_p : V_s \to [1, P]$ assign component blocks to physical processors.

Let $C_T : V_s \to [1, N]$ assign components to computational tasks. We need to ensure that all components belonging to the same task also belong to the same processor $(\forall n \in [1, N], \exists p \in [1, P] : \{v \in V_s | C_T(v) = n\} \Rightarrow \{v | C_P(v) = p\}$.

Let $\{b_1, \ldots, b_B\}$ be a set of physical communication buses, $\{t_1, \ldots, t_N\}$ be the set of tasks, and $\{p_1, \ldots, p_P\}$ be the set of physical processors. Let $B_E = \{b_1, \ldots, b_B\} \cup \{t_1, \ldots, t_N\} \cup \{p_1, \ldots, p_P\}$. Let $C_E : E_S \to B_E$ represent the communication mode for each data message represented by a graph edge. Data can travel remotely (via a data bus b_i) or locally in shared memory (between components within a task t_j or between tasks on a processor p_k).

Let $D \subseteq C_{V_s}$ be the subset of the orderings for $\mathbb{G}_s$ restricted such that if $o \in D$, then

$$\forall v_1, v_2 \in V_s, C_P(v_1) = C_P(v_2) \Rightarrow o(v_1) \neq o(v_2)$$

D is the restriction of the synchronous orderings on V_s to the hardware partitioning, where two vertices cannot have the same order if they share a resource. Note that if the design is not schedulable, D will not exist.

Finally, consider the schedule. Let $S : \mathbb{G}_s \rightarrow \mathbb{R}^{|V_s|+|E_s|}$ represent start times for all elements of the dataflow graph $\mathbb{G}_s$.

Let $D' \subseteq D$ satisfy ($\forall o' \in D'$):

$$\forall v_1, v_2 \in V_s, e = (v_1, v_2) \in E_s \wedge C_P(v_1) \neq C_P(v_2)$$
$$\Rightarrow S(v_2) > S(v_1) + exec(v_1) + exec(e) \wedge o'(v_2) > o'(v_1)$$

.

If two components have a dependency through a remote message, then their start times are constrained by the start time of v_1, the duration of v_1, and the duration of the message represented by the edge e. This models the logical execution time semantics of time-triggered execution. Note that the addition of the edge time may push the order values farther apart by allowing other tasks to execute during the data transfer time, so the reduction involved in D' may be significant. Again, we assume that the model is schedulable.

Since the final set of orderings D' was constructed by reduction from the initial set of orderings C_V, any scheduling policy for ESMoL that enforces the constraints and partitionings shown above will maintain the synchronous semantics of the original Simulink model if the ESMoL model is schedulable. Note that we have not dealt with delays. The scheduling tool described in Porter et al[8] conforms to the constraints as described, if combined with the Stage1 logic to create local dependencies for transitive remote connections as described above. Unfortunately, the scheduler does not enforce end-to-end latencies well, an issue which the authors are currenlty working to address.

7 Evaluation

Our approach for creating high-confidence designs varies somewhat from the traditional V-diagram development model (see Fig. 21). In the traditional model we move down the V, refining designs as we proceed, with the level of integration increasing as the project progresses. We recognize that system integration is often the most costly and difficult part of development. Lessons learned during integration frequently occur too late to benefit project decision-making. We aim to automate much of the integration work, and therefore shorten design cycles. Beyond that, we want to enable feedback of models and analysis results from later design stages back to earlier design cycles (along the dashed lines in the conceptual diagram) to facilitate rapid rework if necessary. The goal is that the overall project can rapidly move towards a correct implementation that most accurately reflects our current understanding of the design problem.

Our case study covers the incremental development of software for the Starmac quadrotor aircraft [25, 26]. We deployed our software to the same hardware as the Starmac controller (with the exception of our internal I^2C link, where the Starmac design used a UART), and tested in a hardware-in-the-loop environment which simulated the Starmac dynamics. Specifically, we conducted three development phases (each with a corresponding set of design models), each of which successively refined the design while preserving the component structure:

1. **Communications Test:** We designed and deployed a shell of the controller architecture, where the software controller components received and sent messages of the proper size, but the system functions only copied data from the input ports to the output ports of each component. The Mathworks xPC Target Hardware-in-the-Loop (HIL) simulator injected known data patterns into the deployed dataflow implementation to ensure that all data paths were valid given the configured schedule.

2. **Quad Integrator Test:** We designed and deployed a simplified version of the quadrotor which acted only along a single axis of motion, removing the rotational dynamics. We were able to validate our control design approach (see [34]), and determine a method for gain adjustments required for stable operation of the deployed controller.

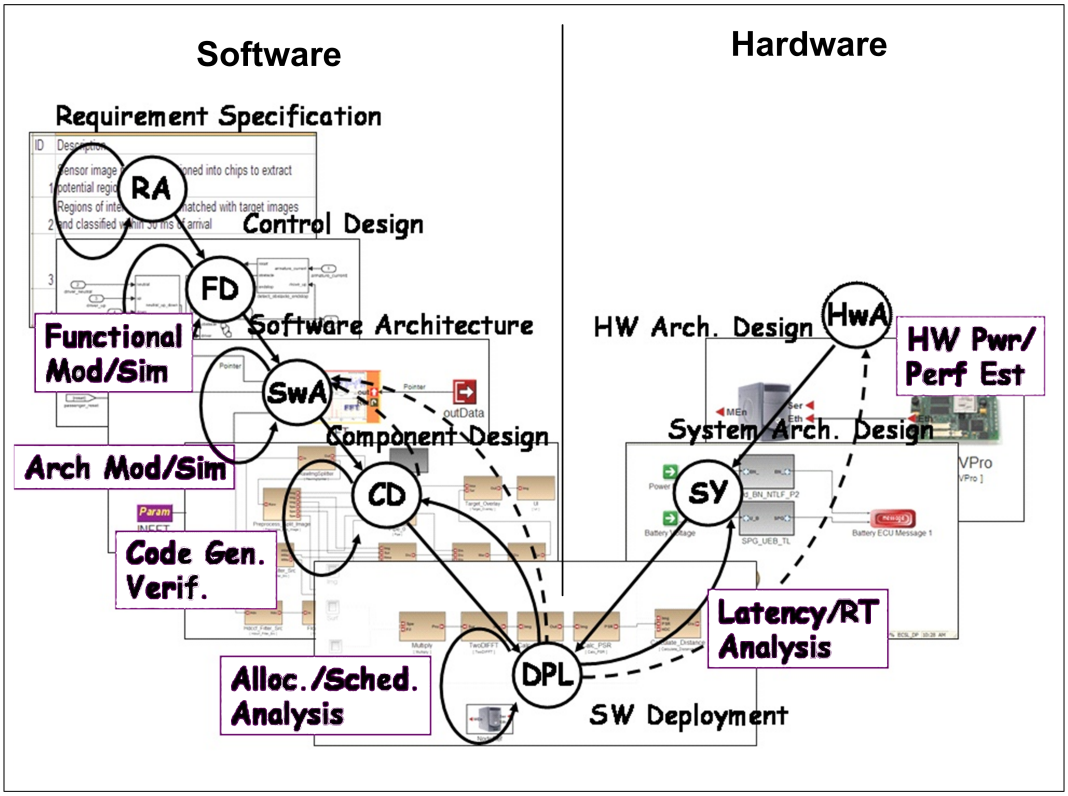


Figure 21: Conceptual development flow supported by the tool chain.

3. **Quadrotor Test:** The final phase evaluated the full quadrotor dynamics and controller implementation. We tested trajectory tracking with the full platform delay effects.

Each of the three development phases answers a set of questions regarding the correctness of the design under nominal operating conditions:

- **Communications Test:**
  1. Is the hardware configuration valid for this software configuration?
  2. Does our deployment mapping communicate the right amounts of data round trip?
  3. Does the configured schedule avoid communication conflicts?
  4. Is data corrupted by the communication protocols or software?
  5. How much delay is introduced by the configured schedule?
- **Quad Integrator Test:**
  1. Does our methodology for selecting stabilizing gains for the control loops adequately handle the schedule delay introduced by data buffering, network communication, and the calculated schedule?
  2. Is our sampling process sufficient for the platform and essential control architecture?
  3. Are there any numerical problems that arise in our functional dataflow implementation considering normal input value ranges?
- **Quadrotor Test:**
  1. Given the additional functions and dimensions in the dataflow, can we still properly answer all of the questions from the previous phase?
  2. Does the full configuration track a reference trajectory?

Fig. 22 is a conceptual depiction of our evaluation environment. The Mathworks xPC Target simulation software runs on a generic small-form-factor PC, with ethernet for configuration and data collection. The xPC system contains an 8-port RS-232 serial expansion card, which communicates with the controller hardware on one port. The simulator and controller send sensor and actuator data back and forth on a single full-duplex serial link running at 57600 baud. The controller hardware consists of two processor boards – the Gumstix Linux board runs the *OuterLoop* controller and the *RefHandler* data input tasks. The Gumstix board has access to an ethernet port, through which the host machine sends new controller software for both control boards. We also use secure shell connections to start and stop the controller, and to monitor for error messages which are printed to the console. An internal I^2C connection allows the two control boards to exchange sensor data and attitude control commands. The Robostix AVR board runs the *InnerLoop* attitude controller and the *DataHandler* sensor data distribution component. One Robostix UART device connects to the xPC simulator as described above. Digital I/O pins allow the monitoring of timing information for the Robostix. We embedded commands to toggle the I/O pins in the controller software, and connected the pins to the LogicPort logic probe. The probe software shows timing traces for evaluating schedule operation (as in Fig. 35). A software AVR simulator was also used to evaluate timing and stack usage for the software running on the AVR. The Robostix board runs FreeRTOS. A Windows virtual machine on the host PC runs the ESMoL modeling tools, logic probe display software, and Simulink which configures and compiles models for the xPC target software. The Linux-based host itself runs the cross-compilers for the controller targets, and secure shell connections to the Gumstix board for status monitoring.

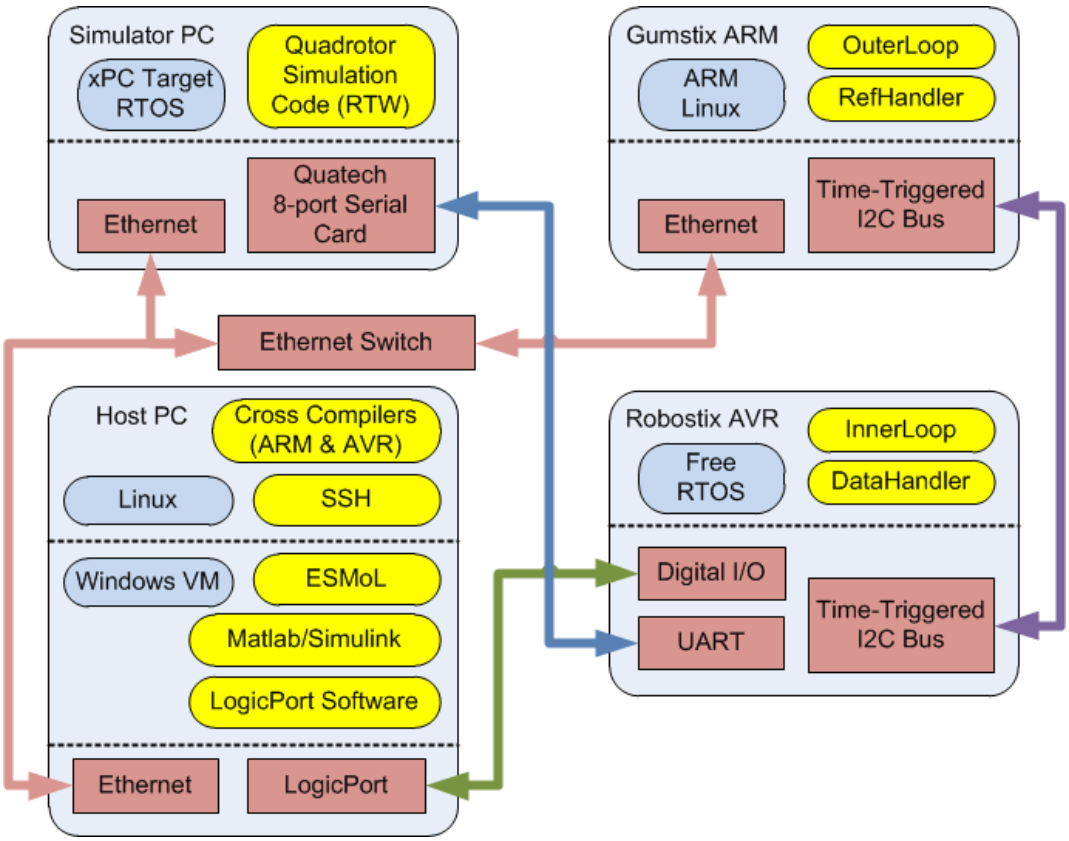


Figure 22: Hardware in the Loop (HIL) evaluation configuration.

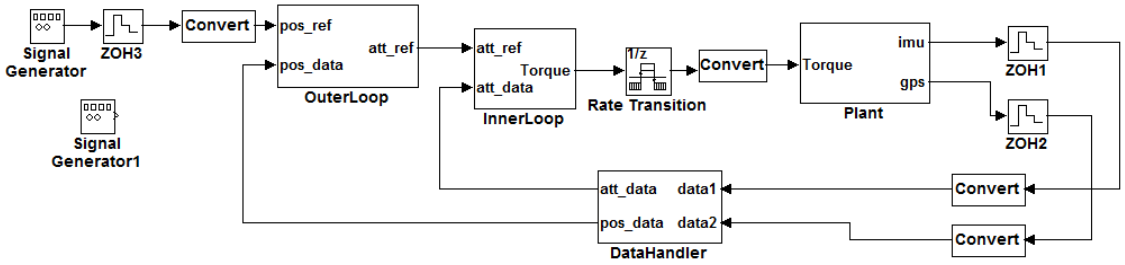


Figure 23: Communications test model.

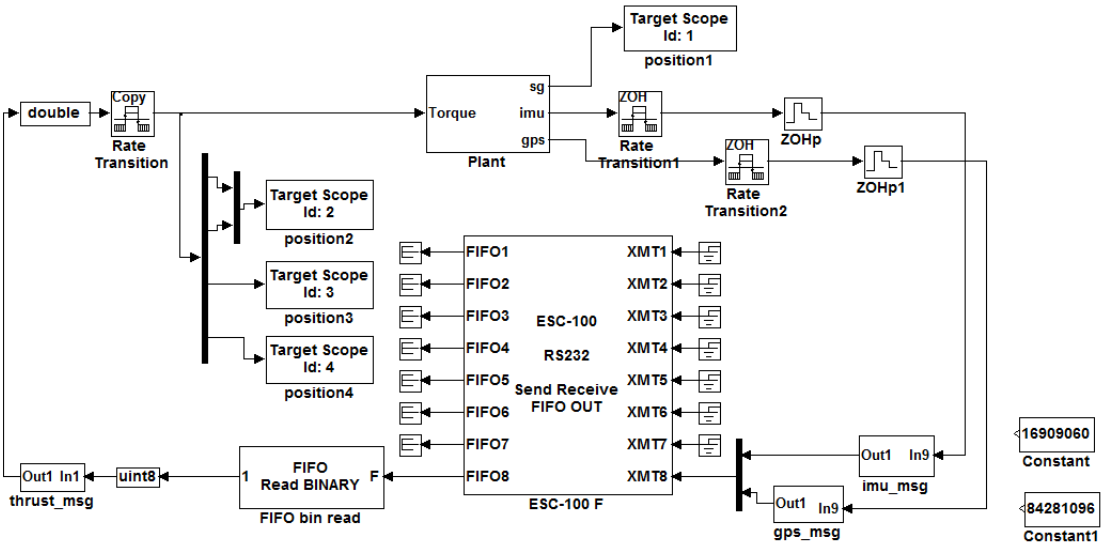


Figure 24: Communications test plant model using the Mathworks xPC Target.

7.1 Communications Test

Fig. 23 displays the simple model used to test data flow over the communication channels. The blocks contain only pass-through elements – multiplexers, demultiplexers, and gains. With this model we verified that data flowed correctly through all of the data paths in the system. The *InnerLoop*, *OuterLoop*, and *DataHandler* components were all realized in software from an ESMoL model, and deployed to the hardware platform. The execution of the components is controlled by a simple time-triggered virtual machine that releases tasks and messages at pre-calculated time instants.

The Mathworks xPC Target simulated the plant dynamics for this test, which in this case amounted only to signal generators to create known data for the simplified controller blocks, and scopes to visualize data received from the controller board. We compared the input and output traces for (delayed) equality (Fig. 24).

During this phase we found problems with the I^2C communications link. The scheduling and timed execution both required precise coordination to prevent data corruption. We also manually discovered a deadlock condition in our communications controller logic. Increasing the speed of the I^2C link from 100 kbits/sec to 400 kbits/sec resolved both the scheduling problem and the deadlock.

7.2 Quad Integrator Model

Our second evaluation phase controls a continuous-time system whose model represents a simplified version of the quadrotor UAV. This model still follows the basic component architecture for the control design (see Fig. 3), but excludes the nonlinear rotational dynamics of the full quadrotor while retaining the difficult coupled stability characteristics. Fig. 26 shows a Simulink model containing the simplified dynamics. The example model controls a stack of four integrators (and motor lag) using two nested PD control loops, as shown in the Simulink diagram of Fig. 25. The Plant block contains the integrator models representing the vehicle dynamics. The two control loops (*InnerLoop* and *OuterLoop*, as shown in Fig. 25) are deployed to the Robostix and Gumstix processors, respectively. We refer to this example as the Quad Integrator model. All of the controller components run at a frequency of 50Hz.

Our controller evaluation method is based on sector theory, proposed originally by Zames[35] to analyze nonlinear elements in a control design. Sectors provide two real-valued parameters which represent bounds on the possible input/output behaviors of a control loop. Kottenstette presented a sector analysis block for

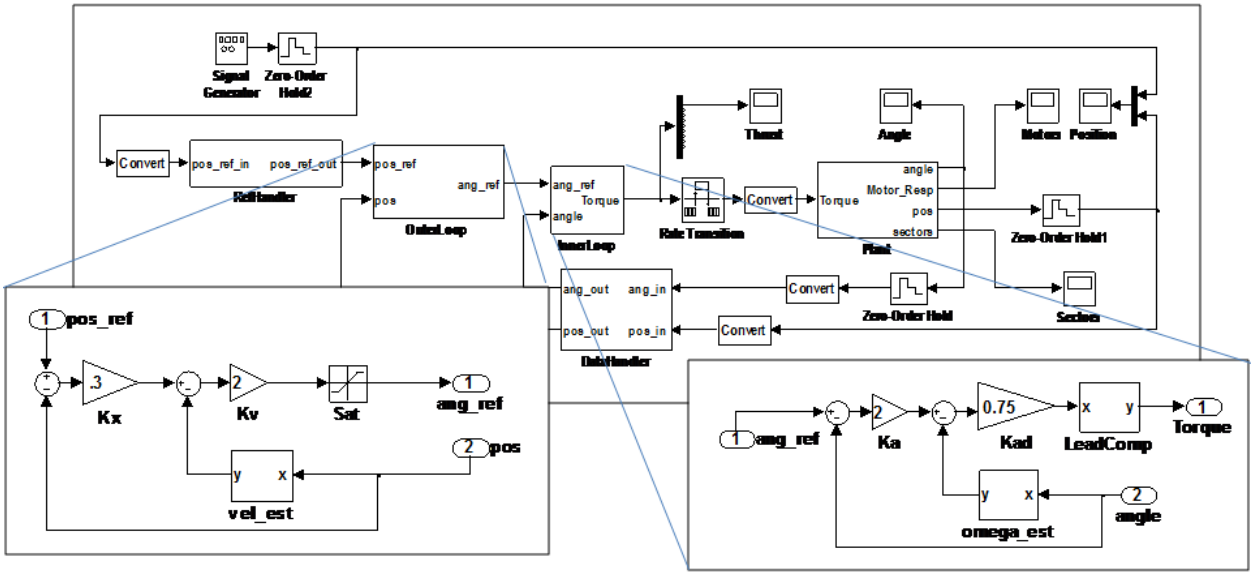


Figure 25: Simulink model of a simplified version of the quadrotor architecture.

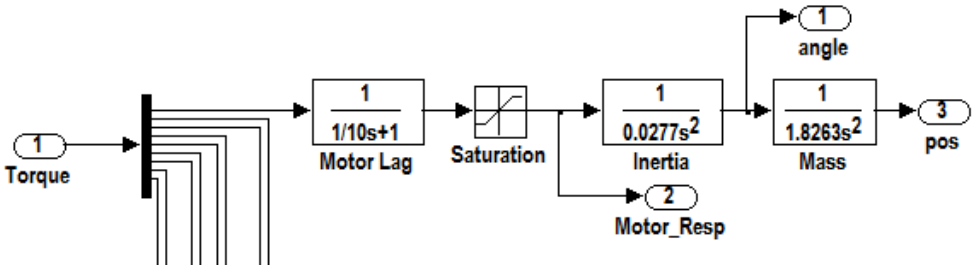


Figure 26: Simplified quadrotor plant dynamics. The signal lines leading off the picture are signal taps used for online stability analysis.

validating a control design in Simulink[36]. We propose to use the same structure to verify the deployed quadrotor control software online. This method is described more fully in Porter et al[34]. A few concepts make this approach appealing for our case:

1. For a given component, the sector measures behavior simultaneously over multiple inputs and outputs, so only one sector analyzer is required per control loop.

2. Our passive abstraction of the system design (described below) allowed us to use a sector analyzer for each control loop to quickly isolate problem components in the deployed design.

Passive control requires that controllers use energy received from inputs or stored previously, introducing no new energy into the environment[37]. If the plant dynamics were passive, we would have considerable freedom in setting gains and choosing control structures. The zero-order hold outputs can introduce small amounts of new energy to the environment during rapid velocity changes, so each of the control loops must mitigate small amounts of "active" behavior. The sector bound a quantifies the energy-generating behavior of each control loop. In our quadrotor system, we expect the bound a to be small and negative and choose the gains appropriately. The result from Kottenstette indicates that the condition $k < -1/a$ is sufficient to ensure stability in these situations (where k is the configured gain of the control loop)[36].

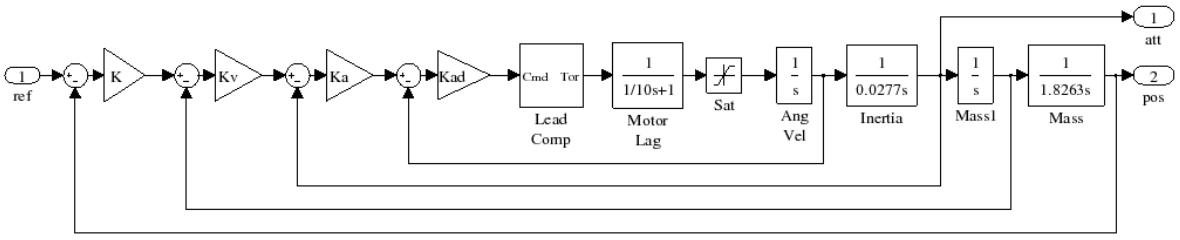


Figure 27: Conceptual nested loop structure of the controller.

This particular design must be evaluated from the innermost loop to the outermost loop in order to make sense of the gain constraints. Fig. 27 shows the nested loop structure of the design. The actual design and implementation are complicated by the physical architecture of the digital realization:

1. Sensors acquire digital attitude and position information only, so velocities must be estimated.
2. The controller components are deployed to different processors in the digital implementation, as described previously. Components on the two processors exchange data messages using a time-triggered protocol.
3. Motor thrust commands are issued periodically using a zero-order hold. As discussed previously the hold introduces additional energy back into the environment, violating the passivity condition.

The sector blocks are attached around each controller, so input and output ports are oriented from the point of view of the control element. The output of the controller (input to the rest of the system) is connected to the sector analyzer input port. The signal controlled by the controller (before the error term is formed) is part of the input to the controller, but from our point of view it is the output of the system, so it connects to the sector analyzer output port. Fig. 28 displays the connection of the sector search block around the position control gain for our example. K_x is the proportional gain for the outer loop PD controller, and K_v is the derivative gain.

Signal	**Original Bound**	**Simulated Sector**	**Measured Sector**	**Delta**	**New Bound**	**New Sector**
Angular Velocity	-1.333	-1.2292	-1.2963	-0.0671	-2.667	-1.4568
Angle	-0.5	-0.0295	-0.4831	-0.4536	-1.0	-0.0068
Velocity	-0.5	-0.1856	-0.4830	-0.2974	-1.0	-0.9324
Position	-3.333	-.7757	-3.8811	-3.1054	-6.667	-1.6081

Table 10: Sector value comparisons for simulation and execution on the actual platform.

For this test we selected a square wave reference input near the highest frequency admissible by the controller. Platform effects caused a significant deviation from our ideal sector estimates and bounds, as illustrated by the sector bound changes in Table 10. Fig. 29 illustrates the evolution of the collected sector data over time. For each digital control signal the table records the following (by column):

1. Original Bound: the sector bound based on the original gain value ($-\frac{1}{k}$).
2. Simulated Sector: the sector value recorded in simulation.
3. Measured Sector: the initial sector value measured on the platform.

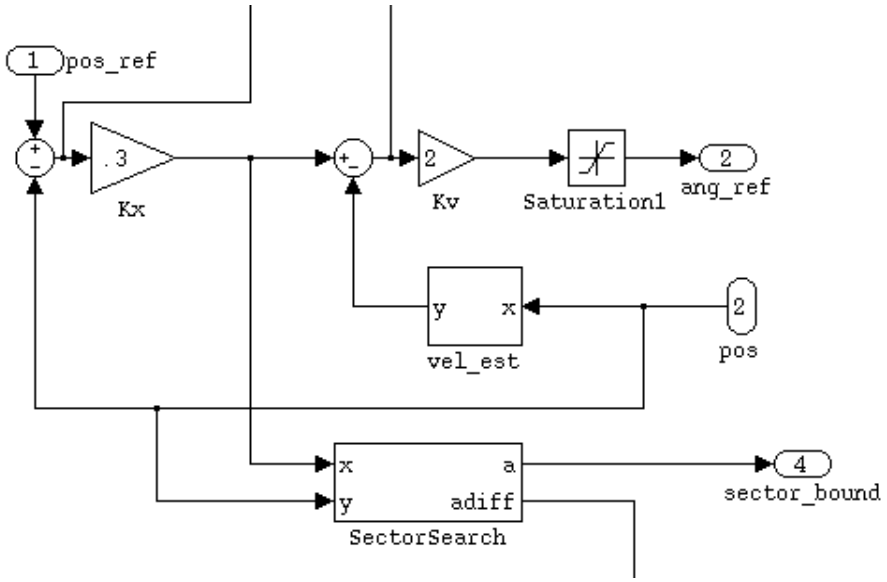


Figure 28: Sector analysis block (*SectorSearch*) connection around the position controller.

4. Delta: the sector difference between the measured and simulated values.

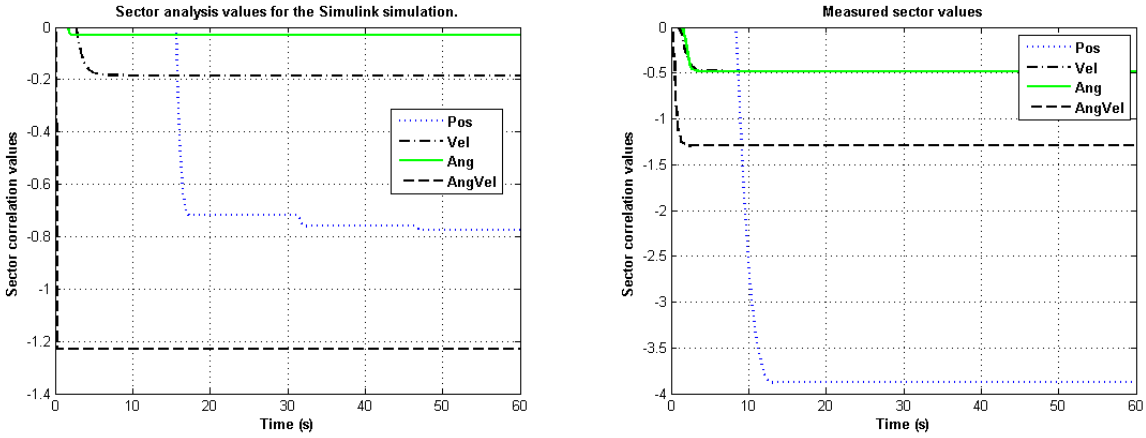
5. New Bound: the sector bound based on the newly adjusted gains.

6. New Sector: the sector value measured on the platform with the new gains.

Although the initial platform gains satisfied the sector stability conditions analytically and in simulation (comparing the Bound column to the Simulated column in the table), the overall system response when deployed to the target platform resulted in significant position overshoot. The measured sector value for position measured the farthest from the predicted value, and exceeded the gain bound for stability ($-1/k$), though no evidence of instability was visible in the plot of the output trajectory. As all of the gains moved right up to the edge of their bounds when deployed, we reduced all of the gains by $\frac{1}{2}$. Note that changing the gains changes the acceptable sector bound as well as the actual sector bounds themselves (as shown in Table 10). After adjusting the gains all of the sector values fell within the bounds.

On closer inspection we discovered that the most significant platform effect was a non-ideal position gain condition for signals with frequencies too close to the sampling rate. Fig. 30 shows a comparison of the ideal frequency response of the outer loop controller block with an empirically measured frequency response for the same controller block deployed on the target hardware. Note the spike at the right-hand side of the plot in Fig. 30(b). This is a nonlinear gain anomaly due to the effects of the saturation block, and which appears only for signals with frequencies right near the Nyquist sampling rate. The remedy was to add a simple input filter to cut off frequencies too close to the sampling rate. This effectively slows down the possible commands that can be issued to the system. The sector analysis blocks helped identify the position control component as the element whose behavior was farthest from predicted when deployed to the platform. Adding a rate limiter block to the reference input resolved the problem. Note that the full quadrotor model already included a similar (but more complex) rate limiter.

The Quad Integrator model simulation exposed a few interesting and unanticipated defects in our design, beyond the gain anomaly detected by the sector analysis. The most significant problem was the asynchronous arrival of the input sensor data. Since the input data transfers were not synchronized with the controller schedule, we had to add a double-buffer to the UART data handler in order to eliminate data corruption.



(a) Simulink simulation.

(b) Execution on hardware (including schedule effects).

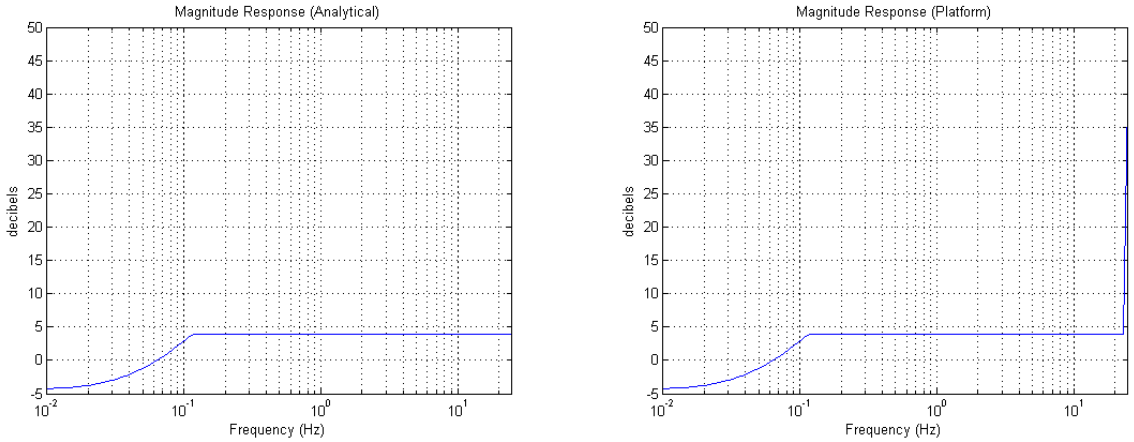
Figure 29: Sector value evolution over time for the quad integrator.

7.3 Quadrotor Model

The final development phase integrated the full dynamics of the quadrotor, comprised of the full data paths and nonlinear functions of the controllers. Figs. 31 - 33 show details from the full Simulink model for the quadrotor. In the top-level design model (Fig. 31), the *robo_stix* block (Fig. 32) contains the functional specifications for the *DataHandler* (*sensor_convert* block) and the *InnerLoop* (*inner_loop* block, also Fig. 33) software components. Likewise the *gum_stix* block and the *ref_data* block specify functions for the *OuterLoop* and *RefHandler* software components.

We used the LogicPort probe to assess the correctness of the schedule. The configured schedule (Fig. 34) correlates with the schedule points measured by the LogicPort analyzer for the tasks and messages on the Robostix board (Fig. 35). Our experimental configuration did not provide a similar means for accurate measurement of the timing on the Gumstix board, though we can observe that message transfers start and end as predicted when task interference is absent. Task interference was only observed for misconfigured schedules, or when other non-controller Gumstix processes created heavy loads, delaying the controller. Both schedule-based and load-based interference were eliminated for nominal operation. Fig. 36 illustrates tracking behavior for the xPC-simulated Quadrotor, where the real-time controller implementation runs on the actual controller hardware. The dashed curves represent the commanded x, y, and z positions as shown, and the solid lines show the actual trajectory achieved by the HIL simulated helicopter using the deployed controller code.

Our first move to the full quadrotor model uncovered numerical problems with some of the emulated floating-point functions provided by the gcc ARM cross-compiler. This forced us to implement our own versions of the single-precision absolute value, signum, and minimum functions for the *OuterLoop* component. This problem was new to this phase of the evaluation because the rate limiter was not present in the Quad Integrator model.



(a) Analytically predicted response.

(b) Measured response.

Figure 30: Magnitude frequency responses for the quad integrator.

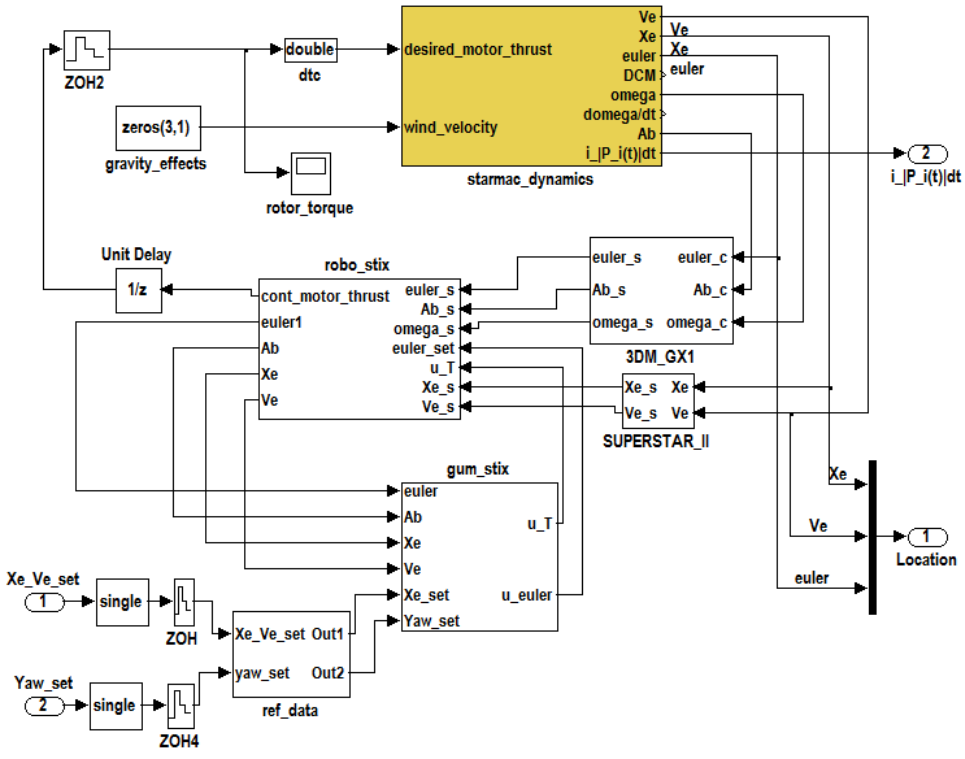


Figure 31: Simulink model of the Starmac quadrotor helicopter.

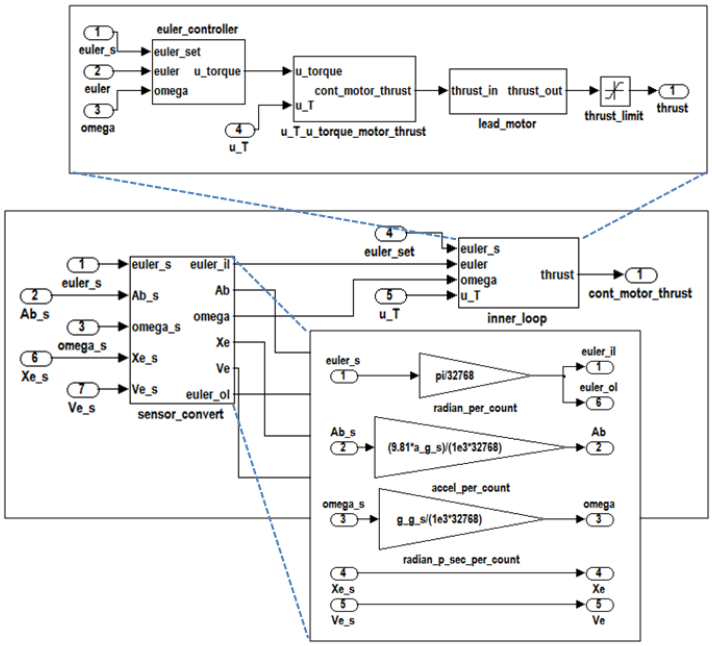


Figure 32: Detail of the Robostix block.

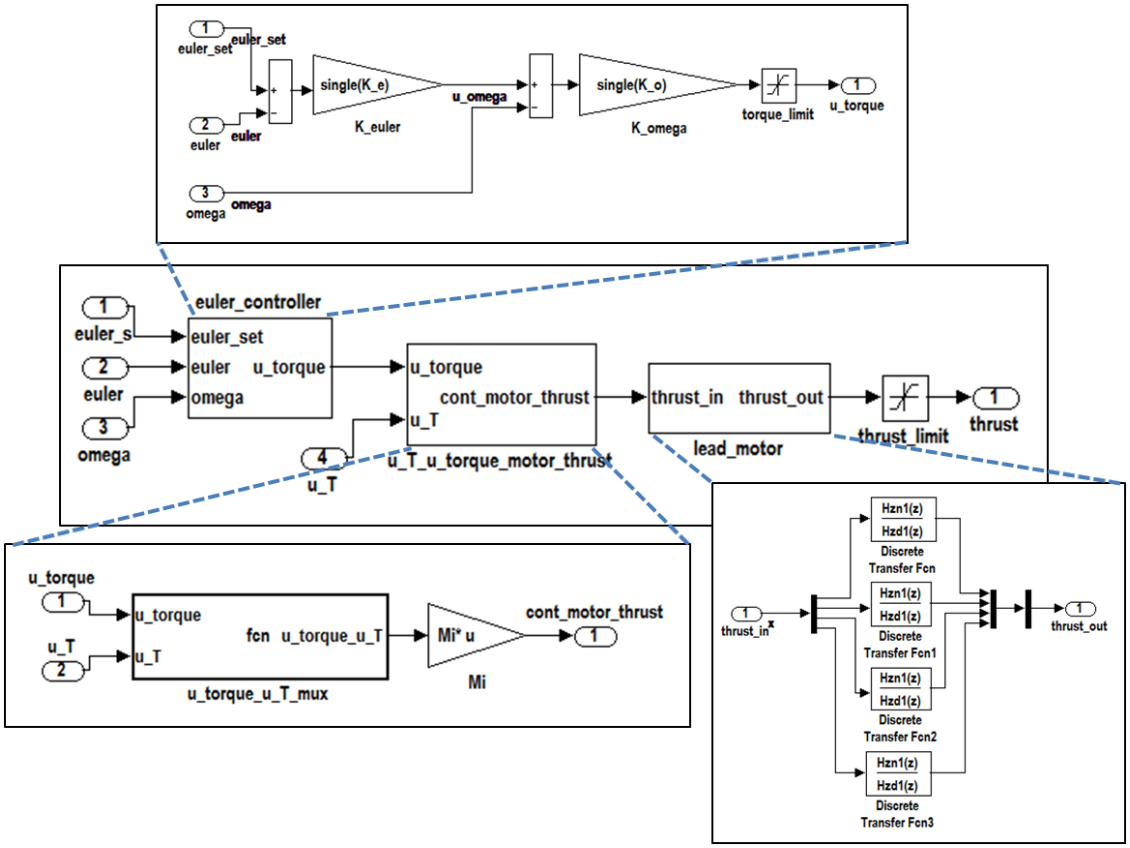


Figure 33: Detail of the inner loop block.

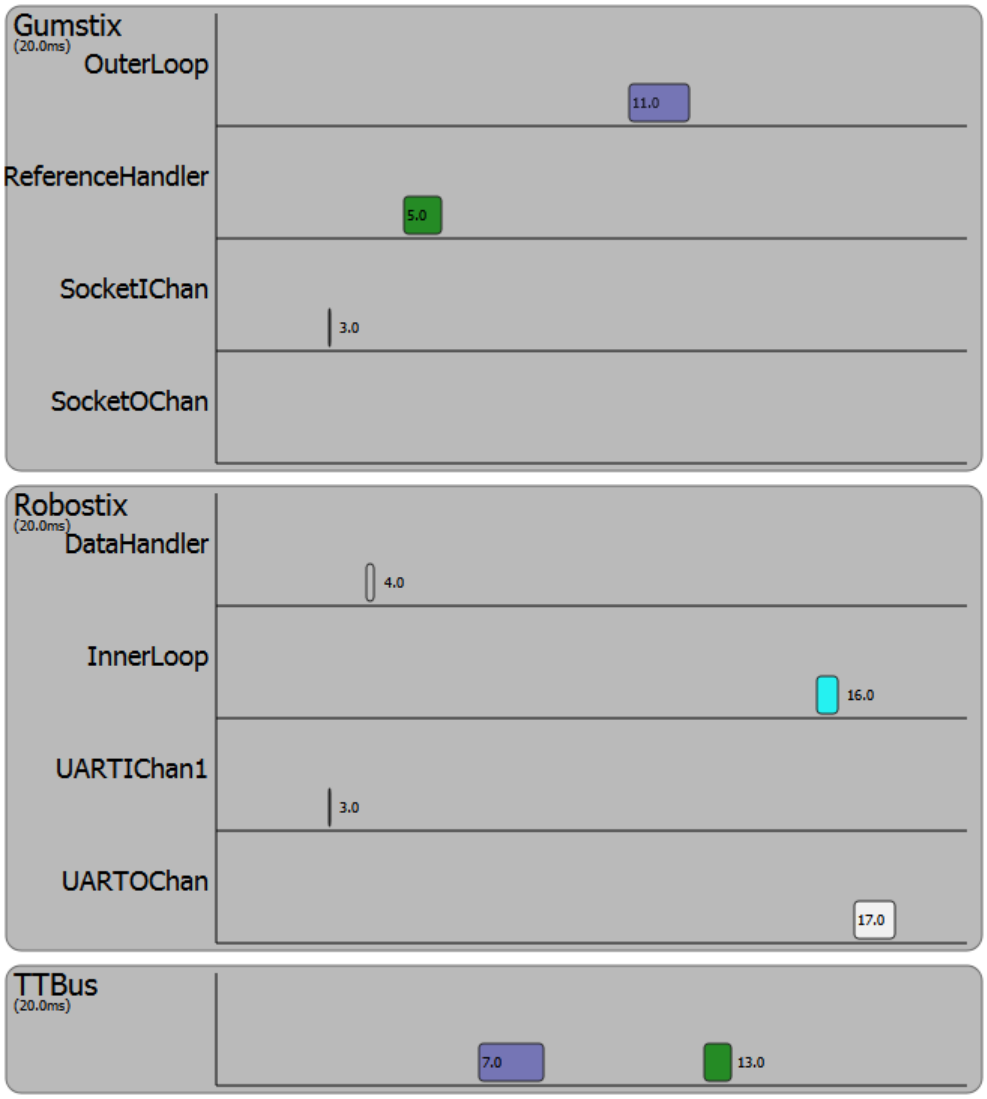


Figure 34: Schedule configuration for the quadrotor.

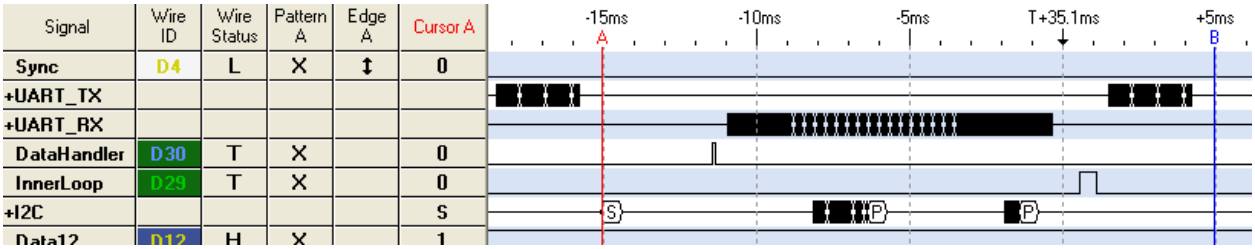


Figure 35: Timing diagram for the Robostix AVR running the inner loop controller.

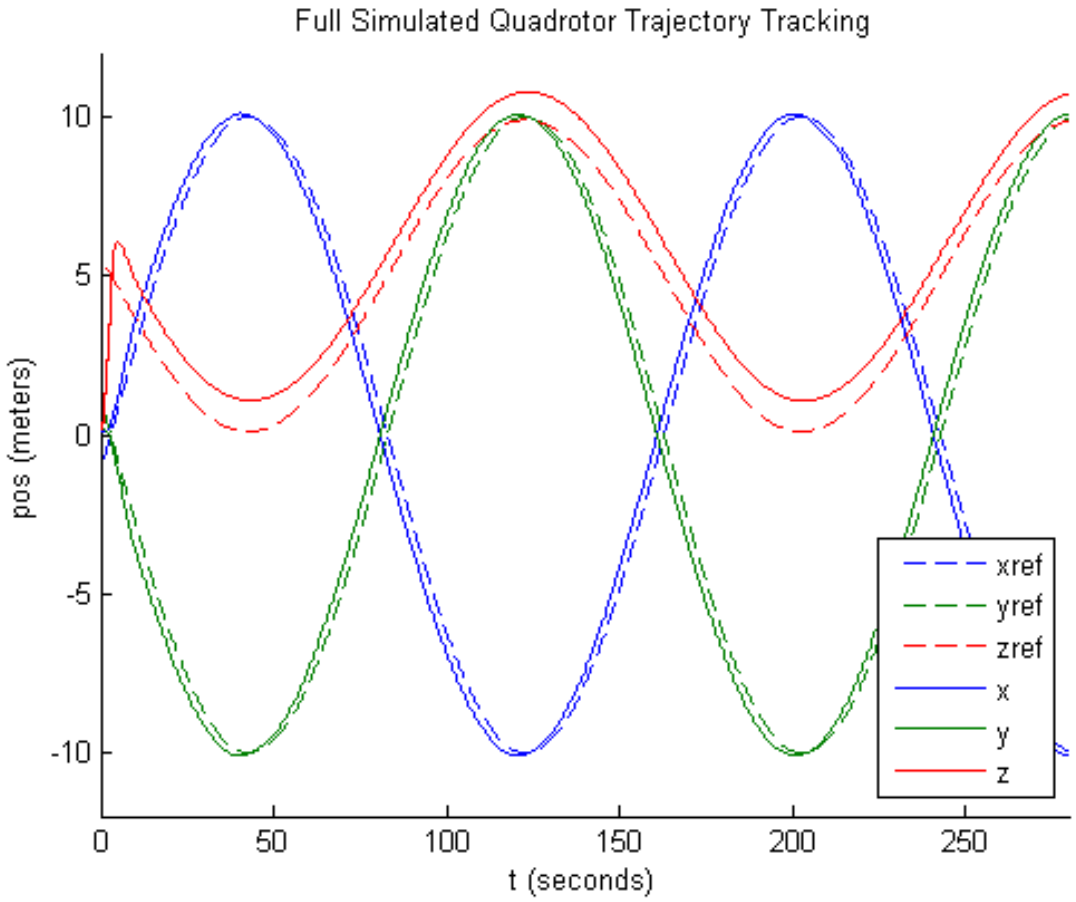


Figure 36: Trajectory tracking for the quadrotor implementation.

8 Lessons and Future Work

Probably the greatest difficulty in our work has been dealing with the large number of moving parts involved in the development of the modeling language and tools, the modeling and implementation of design examples, and the configuration of the development tools and execution environment for the target platform. Our MIC-based solution only covered a part of the entire problem. We only lightly addressed target system configuration, automated updates of the ESMoL model to track changes in the Simulink design, and runtime assessment (of both the simulator for plant dynamics and the target platform with the deployed code). We developed a technique for runtime assessment of controller stability as covered in Porter et al[34] (described partially in the Quad Integrator evaluation section), but it was difficult to automate due to the limited number of free data paths available for debugging in our chosen target system. The integration of third party libraries in the development of our tools, and variations in platform module behavior were not directly addressed by our techniques, though they consumed significant development and testing time.

The next frontier in ESMoL development should be control loop modeling and analysis. Control design formalisms abound, each with its own particular features and capabilities. Passivity and the more general sector analysis formalisms are good examples of compositional frameworks which could be encoded in modeling tools[38] and which could support incremental development.

References

[1] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proc. of the IEEE*, vol. 91, no. 1, pp. 112–126, Jan 2003.

[2] L. P. Carloni, F. D. Bernardinis, C. Pinello, A. L. Sangiovanni-Vincentelli, and M. Sgroi, "Platform-based design for embedded systems," in *The Embedded Systems Handbook*, R. Zurawski, Ed. CRC Press, 2005.

[3] S. McConnell, *Rapid Development: Taming Wild Software Schedules.* Redmond, WA: Microsoft Press, 1996.

[4] John Hudak and Peter Feiler, "Developing AADL Models for Control Systems: A Practitioner's Guide," CMU SEI, Tech. Rep. CMU/SEI-2007-TR-014, 2007.

[5] N. Pontisso and D. Chemouil, "Topcased combining formal methods with model-driven engineering," in *ASE '06: Proc. of the 21st IEEE/ACM International Conf. on Automated Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 359–360.

[6] J. Porter, G. Karsai, P. Volgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits, "Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation," in *Workshops and Symposia at MoDELS 2008 (ACES-MB), LNCS 5421.* Toulouse, France: Springer, 2009.

[7] G. Hemingway, J. Porter, N. Kottenstette, H. Nine, C. vanBuskirk, G. Karsai, and J. Sztipanovits, "Automated Synthesis of Time-Triggered Architecture-based TrueTime Models for Platform Effects Simulation and Analysis," in *RSP '10: 21st IEEE Intl. Symp. on Rapid Systems Prototyping*, Jun 2010.

[8] J. Porter, G. Karsai, and J. Sztipanovits, "Towards a time-triggered schedule calculation tool to support model-based embedded software design," in *EMSOFT '09: Proc. of ACM Intl. Conf. on Embedded Software*, Grenoble, France, Oct 2009.

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. T. IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," *Workshop on Intelligent Signal Processing*, May 2001.

[10] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and memory requirements analysis with AADL," *Ada Lett.*, vol. XXV, no. 4, pp. 1–10, 2005.

[11] "The Cheddar project : A free real time scheduling analyzer," http://beru.univ-brest.fr/ singhoff/cheddar.

[12] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. of the IEEE*, vol. 91, pp. 84–99, Jan 2003. [Online]. Available: http://www.gigascale.org/pubs/397.html

[13] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," in *Proc. of the 2005 ACM Conf. on Lang., Compilers, and Tools for Embedded Systems (LCTES '05)*. New York, NY: ACM Press, Jun 2005, pp. 31–39.

[14] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ, "Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language (TDL)," in *Proc. of the 4th Intl. ICSE workshop on Software Eng. for Automotive Systems*, Minneapolis, May 2007.

[15] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, and A. L. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, Apr 2003.

[16] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time simulink to lustre," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.

[17] D. Park, *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Springer Netherlands, 2007, ch. Translation of Safety-Critical Software Requirements Specification to Lustre, pp. 157–162.

[18] R. Alur and G. Weiss, "RTComposer: a framework for real-time components with scheduling interfaces," in *EMSOFT '08: Proc. of the 8th ACM Intl. Conf. on Embedded software*. New York, NY, USA: ACM, 2008, pp. 159–168.

[19] W. Herzner and R. S. et al, "Model-Based Development of Distributed Embedded Real-Time Systems with the DECOS Tool-Chain," in *Proc. of SAE 2007 AeroTech Congress & Exhibition*, Los Angeles, CA, USA, Sep 2007.

[20] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan 2003.

[21] Aditya Agrawal and Gabor Karsai and Sandeep Neema and Feng Shi and Attila Vizhanyo, "The design of a language for model transformations," *Journal on Software and System Modeling*, vol. 5, no. 3, pp. 261–288, Sep 2006.

[22] E. Magyari, A. Bakay, A. Lang, and et al, "UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages," in *The 3rd OOPSLA Workshop on Domain-Specific Modeling*, Oct 2003.

[23] R. Thibodeaux, "The specification and implementation of a model of computation," Master's thesis, Vanderbilt Univ., May 2008.

[24] E. Lee and A. Sangiovanni-Vincentelli, "A unified framework for comparing models of computation," *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, December 1998.

[25] G. Hoffmann, D. G. Rajnarayan, S. L. Waslander, D. Dostal, J. S. Jang, and C. J. Tomlin, "The stanford testbed of autonomous rotorcraft for multi-agent control," in *the Digital Avionics System Conference 2004*, Salt Lake City, UT, November 2004. [Online]. Available: pubs/DASC_2004.pdf

[26] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Quadrotor helicopter flight dynamics and control: Theory and experiment," in *Proc. of the AIAA Guidance, Navigation, and Control Conf.*, Hilton Head, SC, August 2007, aIAA Paper Number 2007-6461. [Online]. Available: pubs/Quadrotor_Dynamics_GNC07.pdf

[27] S. Neema and G. Karsai, "Embedded control systems language for distributed processing (ECSL-DP)," Inst. for Software Integrated Sys., Vanderbilt Univ., Tech. Rep. ISIS-04-505, 2004. [Online]. Available: http://www.isis.vanderbilt.edu/publications/archive/Neema_S_5_12_2004_Embedded_C.pdf

[28] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree, "From control models to real-time code using giotto," *Control Systems Magazine*, vol. 2, no. 1, pp. 50–64, 2003.

[29] A. Pinto, L. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli, "Interchange formats for hybrid systems: Abstract semantics," in *Hybrid Systems: Computation and Control*, J. Hespanha and A. Tiwari, Eds., Mar 2006, pp. 491–506.

[30] Google, "CTemplate, A Simple but Powerful Language for C++," http://code.google.com/p/google-ctemplate.

[31] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems," *Constraints*, vol. 5, no. 4, pp. 335–357, Oct. 2000.

[32] C. Schulte, M. Lagerkvist, and G. Tack, "Gecode: Generic Constraint Development Environment," http://www.gecode.org/.

[33] K.-E. Arzen and B. B. et al, "Integrated control and scheduling," Dept. of Automatic Control, Lund Inst. of Technology, Sweden, Tech. Rep. ISRN LUTFD2/TFRT–7586–SE, Aug 1999.

[34] J. Porter, G. Hemingway, N. Kottenstette, G. Karsai, and J. Sztipanovits, "Online stability validation using sector analysis," in *EMSOFT '10: Proc. of ACM Intl. Conf. on Embedded Software*, Scottsdale, AZ, Oct 2010.

[35] G. Zames, "On the input-output stability of time-varying nonlinear feedback systems part one: Conditions derived using concepts of loop gain, conicity, and positivity," *Automatic Control, IEEE Trans. on*, vol. 11, no. 2, pp. 228–238, Apr 1966.

[36] N. Kottenstette and J. Porter, "Digital passive attitude and altitude control schemes for quadrotor aircraft," in *ICCA '09: 7th IEEE Intl. Conf. on Control and Automation*, ChristChurch, New Zealand, 2009.

[37] M. D. la Sen, "Links between dynamic physical systems and operator theory issues concerning energy balances and stability," *American Journal of Applied Sciences I*, vol. 3, pp. 248–254, 2004.

[38] E. Eyisi, J. Porter, J. Hall, N. Kottenstette, X. Koutsoukos, and J. Sztipanovits, "PaNeCS: A Modeling Language for Passivity-based Design of Networked Control Systems," in *2nd Workshop on the Arch. and Constr. of Emb. Sys – Model-Based (ACES-MB)*, Denver, Colorado, 2009.