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

The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, 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NOTICE: This is a draft of material collected from published (and unpublished) papers to cover the design philosophy and some of the details of the language and the tools. There are still some inconsistencies in the examples, but we are revising this material for publication. Please send comments, corrections, notes, etc... to jporter@isis.vanderbilt.edu. Thanks, -Joe

Chapter 1

High-Confidence Design Toolchain

The Embedded Systems Modeling Language (ESMoL) provides a mechanism for automating software development for high-confidence distributed control designs. We describe the language here, in a summary taken from previous papers: [1] gives an overview of the modeling language and tools, and [2] describes the time-triggered schedule calculation tool.

1.1 Overview

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 "'complete"' designs. 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 [3].

Software development uses UML (or similar) tools to capture concepts such as components, interactions, timing, fault handling, and deployment. Work flows focus on source code organization and management, followed by testing and debugging on target hardware. Physical and environmental constraints are not represented by the tools. At best such constraints may be provided as documentation to developers.

Complete systems rely on both aspects of a design. 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 functional dataflow model may fail to perform correctly when its functions are distributed over a shared network of processing nodes. Cost considerations may force the selection of platform hardware that limits timing accuracy. Neither aspect of development supports comprehensive validation of certification requirements to meet government safety standards.

1.1.1 High-Confidence Design Challenges

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 the implementation and engineering solutions based on incomplete or faulty understanding of design issues. Current state of the art resolves these problems by reviewing many of the details in meetings and personal discussions. In the worst cases serious incompatibilities are not discovered until very late in the design cycle, leading to project overruns and cancellations.

2. 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.

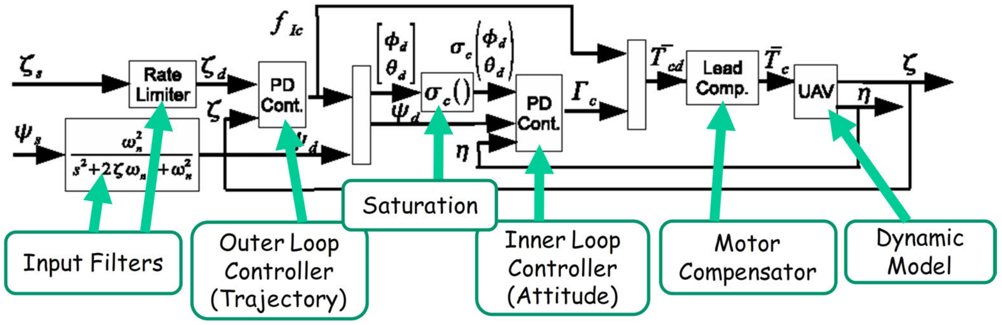

Figure 1.1: Basic architecture for the quadrotor control problem.

3. Long development, deployment, and test cycles limit the amount of iterative rework that can be done to get a correct design. 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.

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. 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. A successful suite of ESMoL tools will have numerous interpreters, so the maintenance difficulties will only grow larger.

1.1.2 Solution

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. 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 design environment so that modeling, analysis, simulation, and code generation artifacts are all clearly related to a single design model. The models directly relate Simulink control design structures with software and hardware design concepts.

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.

3. ESMoL's integrated analysis, simulation, and deployment capabilities can shorten design cycles.

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

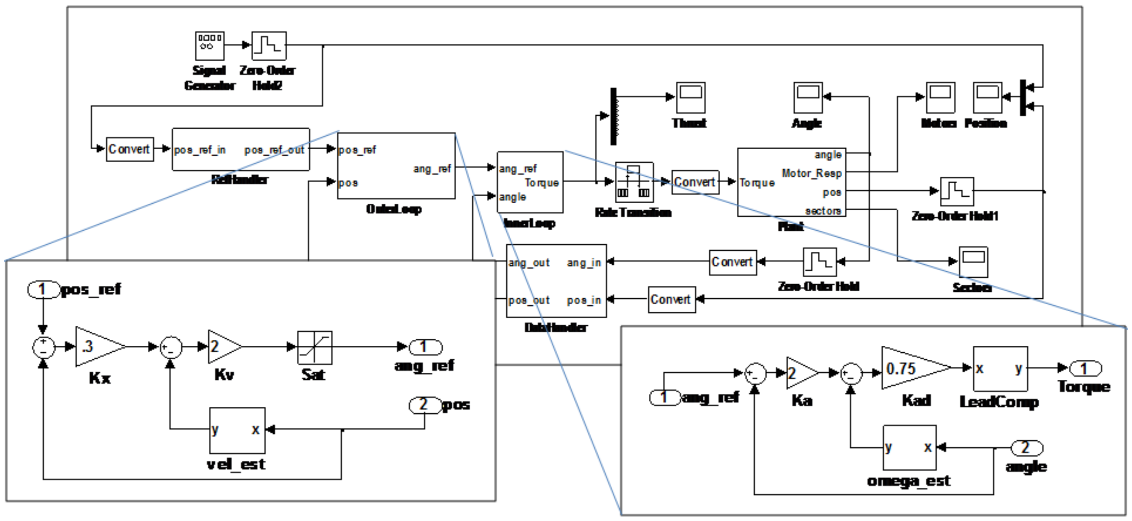

Figure 1.2: Simulink model of a simplified version of the quadrotor architecture.

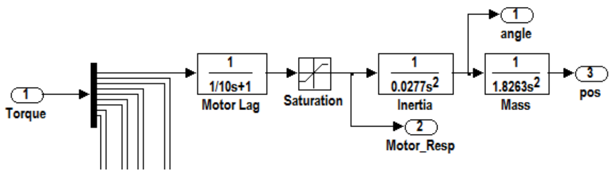

Figure 1.3: Simplified quadrotor plant dynamics. The rotational dynamics have been removed to facilitate easier study of the behavior. The full model includes all of the dynamics. The signals lines leading off the picture are signal taps used for online stability analysis.

a program under compilation. ESMoL itself tries to avoid explicitly representing tedious details in order to make the user experience 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 absorbed by the first stage transformation.

5. We will demonstrate the integration of a simple language for time-triggered scheduling analysis into the ESMoL tools using the two-stage architecture. The scheduling analysis tool input language and details are also covered here – the constraint models built by the analysis tool include timing adjustments for some of the platform effects represented by ESMoL model objects.

1.2 Design Example

Our design example controls a continuous-time system whose model represents a simplified version of a quadrotor UAV. Fig. 1.1 depicts the basic component architecture for the control design. Our example excludes the nonlinear rotational dynamics of the actual quadrotor for simplicity, but retains the difficult stability characteristics. Fig. 1.3 shows a Simulink model containing the simplified dynamics. For the fully detailed quadrotor model and a complete discussion of the control design philosophy, see [4]. The example

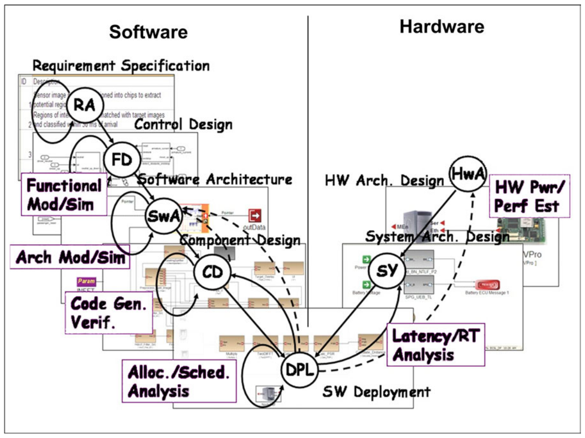

Figure 1.4: Conceptual development flow supported by the tool chain.

model controls a stack of four integrators (and motor lag) using two nested PD control loops. The Plant block contains the integrator models representing the vehicle dynamics. The two control loops (inner and outer, as shown in Fig. 1.2) are implemented on separate processors, and the execution of the components is controlled by a simple time-triggered virtual machine that releases tasks and messages at pre-calculated time instants. We refer to this example as the Quad Integrator.

1.3 Vision

Our approach for creating high-confidence designs varies somewhat from the traditional V-diagram development model (see Fig. 1.4). 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 (via the dashed lines in the diagram) to facilitate rapid rework if necessary. The goal is that the overall project can rapidly move towards a correct implementation.

Consider the general class of control system designs for use in a flight control system. Sensors, actuators, and data networks are designed redundantly to mitigate faults. The underlying platform implements a variant of the time-triggered architecture (TTA) [5], 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. Software implementations of the control functions must pass strict certification requirements which impose constraints on the software as well as on the development process.

Figure 1.5 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) [6], configured to edit ESMoL models. The modeler then uses the dataflow models imported from Simulink to specify the functions of software components that will implement the controllers. These components are extended with interfaces defining their input and output message structures. The components are instantiated into multi-aspect models where logical dependencies, hardware deployment, and timing models can be specified for the software architecture. The completed model is transformed (via the Stage 1 transformation) 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 integrating using the Stage 2 transformation.

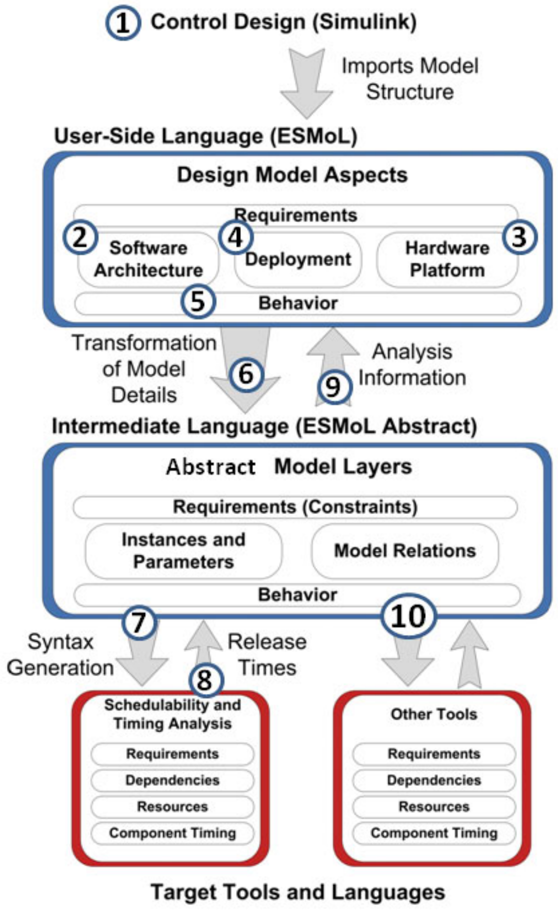

Figure 1.5: Flow of design models between design phases.

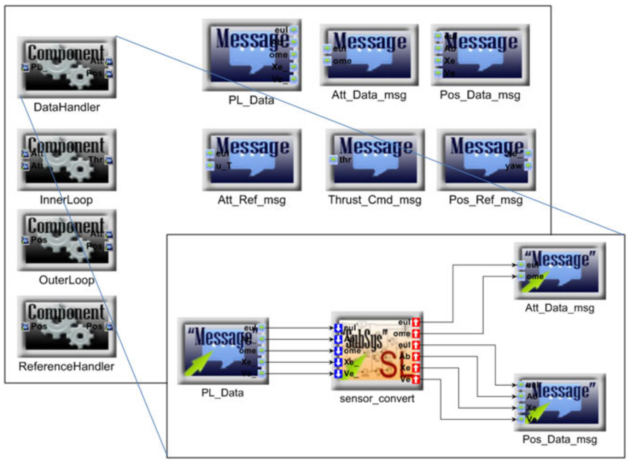

Figure 1.6: The component type model specifies message types and fields, as well as component types. Component types specify the mapping of Simulink functional designs to the fields in the messages that will carry the data. These components and messages are instantiated in the architecture and deployment models.

1.3.1 Requirements Analysis (RA)

This example has an informal set of requirements, though our modeling language currently supports the formalization of timing constraints between sensor and actuator tasks. Formal requirements modeling offers great promise, but in ESMoL requirements modeling is still in conceptual stages. We discuss latency modeling structures in a later example.

Informally, we require stability of the closed-loop system over the full range of possible inputs. Performance should allow reasonable tracking of a reference trajectory within the physical limits of the vehicle.

1.3.2 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.

Fig. 1.2 shows a simple top-level Simulink design for the quadrotor model. The imported ESMoL 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. A model import utility provides the illustrated function.

1.3.3 Component Design (CD)

In the component design phase (CD) we specify interfaces for the functions which will run in the distributed controller network. 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 other parts of the design (e.g., in the logical

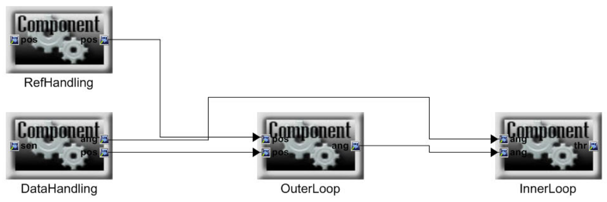

Figure 1.7: The logical architecture aspect specifies data dependencies between software component instances. The ports on the blocks represent data messaging interfaces into and out of the component.

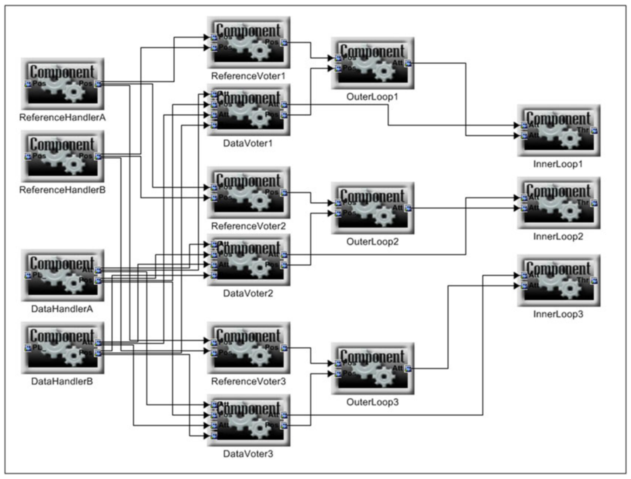

Figure 1.8: Triply-redundant version of the quadrotor logical architecture to illustrate instantiation in ESMoL design models. Each of the quadrotor design components is instantiated multiple times in this model.

architecture diagram described below) the message references specified here will appear as ports on the instance. Connections to and from those ports represent the transfer of that data message into or out of the component.

Fig. 1.6 shows an example of the component interface definition language. Message fields and their sizes are specified here, as well as component implementations and interfaces. These specifications are types in an ESMoL model, and they are instantiated in the architecture and deployment models. The quad integrator 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 in the figure shows the internals of the DataHandler component specification. The block in the center is a reference to the Simulink block that specifies its functional behavior. The blocks on the outer edges are references to messages defined at the top level of the system types model. 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.

In the component types sublanguage a component may be implemented by either a Simulink Subsystem or a C function. They are compatible at this level, because here their model elements represent the code that will finally implement the functions. They are assumed to execute synchronously. These units are modeled as blocks with ports, where the ports represent parameters passed into and out of C function calls.

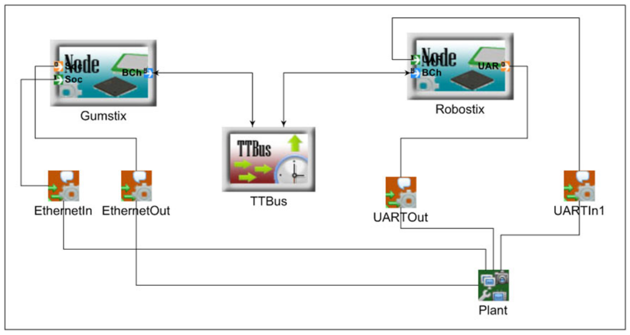

Figure 1.9: Overall hardware layout for the quadrotor example.

1.3.4 Logical Software Architecture (SwA)

Fig. 1.7 portrays logical data dependencies between software component instances, independent of their distribution over different processors. The software architecture model describes the logical interconnection of component instances. Semantics for SwA Connections are those of task-local synchronous function invocations as well as message transfers between tasks using time-triggered communication. The 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.

Creation of a (GME) reference object to one of the component types corresponds to instantiation. The architecture design model aspects (logical architecture, deployment, and timing) allow us to perform software design with components whose functions are specified by Simulink dataflow diagrams. Fig. 1.8 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. This assumes that the hardware support is all available, and could be specified in the hardware and deployment models – this particular model diagram is only shown to illustrate the instantiation mechanism.

1.3.5 Hardware Architecture (HwA)

Hardware configurations are explicitly modeled in the platform language. 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.

Fig. 1.9 illustrates an example of a platform model. The quadrotor architecture is deployed to a small embedded processor assembly manufactured by Gumstix, Inc. The 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 on 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 I2C bus which runs a software emulated time-triggered protocol.

1.3.6 Deployment Models (SY, DPL)

Fig. 1.10 shows 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.

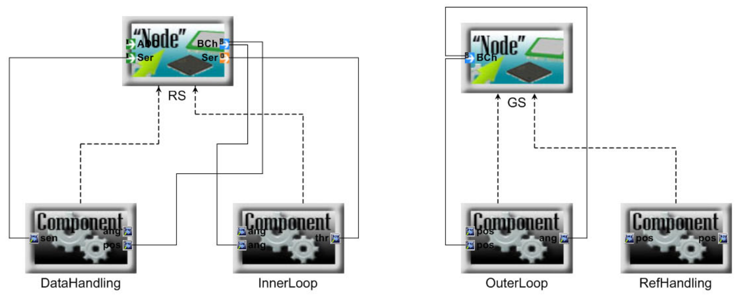

Figure 1.10: The platform deployment aspect specifies the hardware components that will support the transfer of data to and from each processing node, the software components that will send and receive the data, and the message instances in which the data will be carried. Dashed arrows represent assignment of components to their respective processors, and solid lines represent assignment of message instances (component ports) to communication channels (port objects) on the processor.

RS (Robostix) is an 8 bit ATMega128 CPU, and GS (Gumstix) is an Intel PXA ARM-based CPU.

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 takes the form of messages in the model. Whether delivered locally (same processing node) or remotely, all inter-task messages are pre-scheduled for delivery. ESMoL uses logical execution time semantics found in time-triggered languages such as Giotto [7] – message delivery is scheduled after the deadline of the sending task, but before the release of the receiving tasks. In the TT model message receivers assume that required data is already available at task release time. Tasks never block, but execute with whatever data is available each period.

Deployment concepts – tasks running on processing nodes and messages sent over data buses – are modeled as shown in Fig. 1.10. 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 objects on the node can also be connected to message objects on a component. These connections represent the flow of data from the physical environment 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 analysis artifacts.

1.3.7 Timing Models

Fig. 1.11 shows a timing and execution model for the quad integrator, which allows the designer to attach timing parameter blocks to components and messages. For the time-triggered case the configuration parameters include execution period and worst-case execution time. The quad integrator model runs all of the blocks at a frequency of 50Hz. 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. Our example model assigns each component to its own task. For calculated static schedules start times are also stored in the timing objects after the analysis has been performed.

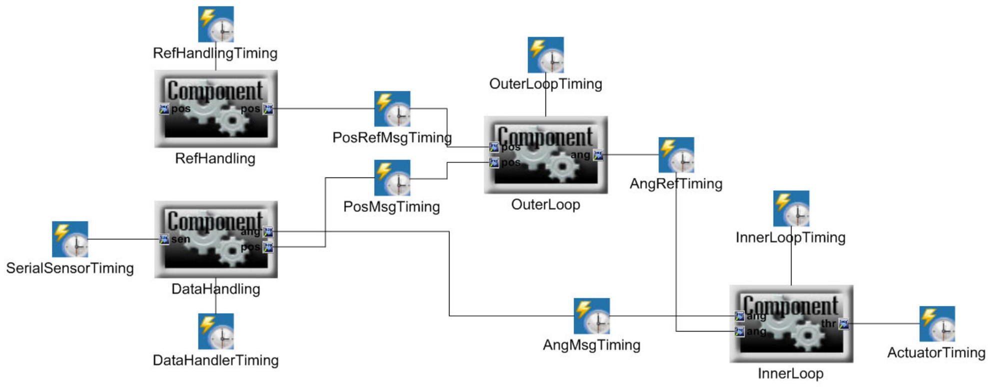

Figure 1.11: The timing/execution model indicates which components and messages will be scheduled independently. In the case of processor-local data transfers transfer time is neglected, as reads and writes occur in locally shared memory. The time order of the message writer and readers are enforced by the static schedule. The locality of a message transfer is specified in the architecture and deployment models. GME provides a separate window (not shown) for editing object parameters and attributes.

1.4 Key Techniques and Technologies

The models in the example and the metamodels described below were created using the ISIS Generic Modeling Environment tool (GME) [6]. 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 approach[8] 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.

The Simulink and Stateflow sublanguages of our modeling environment are described elsewhere, though the ESMoL language changes many of the other design concepts from previously developed languages described by Neema [9].

Another key technology used in the ESMoL tool suite is the GReAT model transformation language (and its associated code generation tools)[10]. ESMoL 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 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 [1], Thibodeaux [11], and Hemingway et al [12] for details).

High-confidence systems require platforms providing services and guarantees for properties such as fault containment, temporal firewalls, and partitioning. System developers should not re-implement such critical services from scratch [3]. The platform also defines a model of computation (MoC) [13]. A MoC governs how the concurrent objects of an application interact (i.e. synchronization and communication). Our chosen MoC is the time-triggered architecture (TTA)[5]. The time-triggered architecture provides deterministic synchronous execution in order to ensure the consistent behaviors of distributed replicas of controller components.

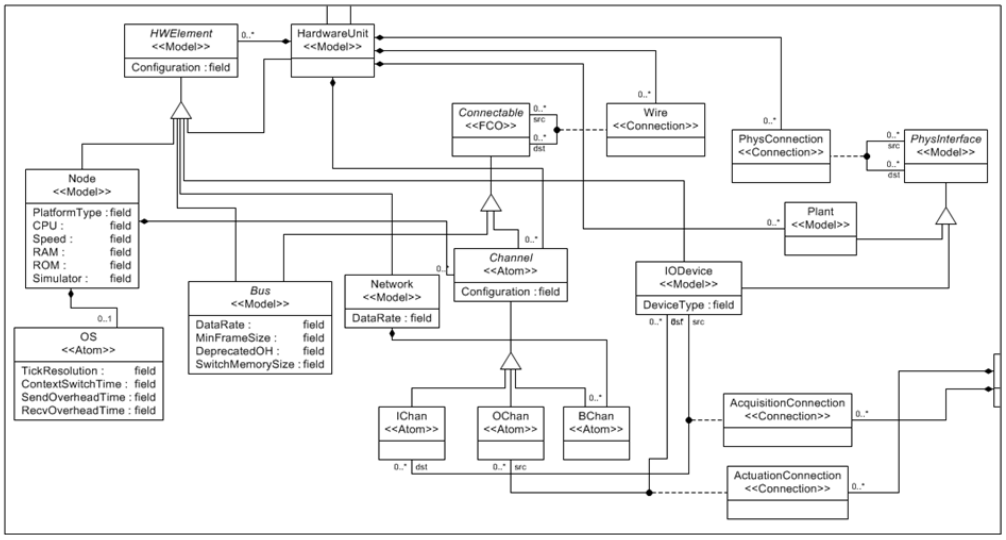

Figure 1.12: Platforms. This metamodel describes a simple language for modeling the topology of a time-triggered processing network. A sample platform model is included.

1.5 ESMoL Language Details

The GME metamodeling syntax may not be entirely familiar to the reader, but it is well-documented in Karsai et al [6]. Class concepts such as inheritance can be read analogously to UML. Class aggregation represents containment in the modeling environment, though an aggregate element can 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. One unique notation in MetaGME (the GME modeling language for creating modeling languages) is the dot used for relating an association class to its endpoint connection classes. For example, the dot between the Connectable class and the Wire class (Fig. 1.12) represents a line-style connection in the modeling environment.

A simple platform definition language (Fig. 1.12) 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.

Fig. 1.13 portrays the System Types sublanguage. Components of different types (here Simulink block references or C code blocks) specify the component functions. Message references define interfaces on the components, and ports on those message represent message data fields. Directed connections between the functional block and the message field ports represent marshaling and demarshaling of message data as it is consumed and produced by the specified component functions.

The Fig. 1.14 metamodel 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 instances in ESMoL, 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 object (node references) and connectors (ComponentAssignment and CommMapping) to identify the mapping of tasks and messages to the platform model.

The timing sublanguage (Fig. 1.15) 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. Components can also be grouped into a task. Messages are similarly annotated. The annotation objects contain

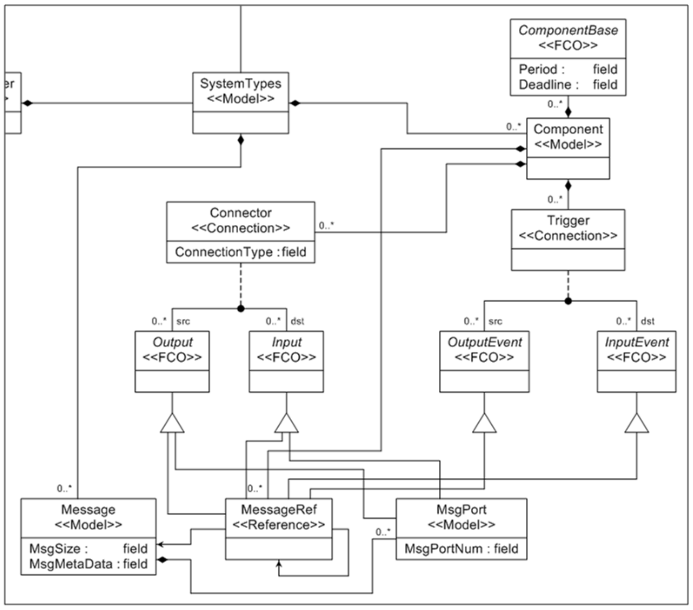

Figure 1.13: System Types Metamodel. Architecture models use instances of Simulink subsystems or C code functions to specify the functionality of software components, adding attributes for real-time execution. The Input and Output port classes are typed according to the implementation class to which they belong. The connections between the block reference and the MsgPort objects describe the details required to marshal and demarshal messages for use by the specified function.

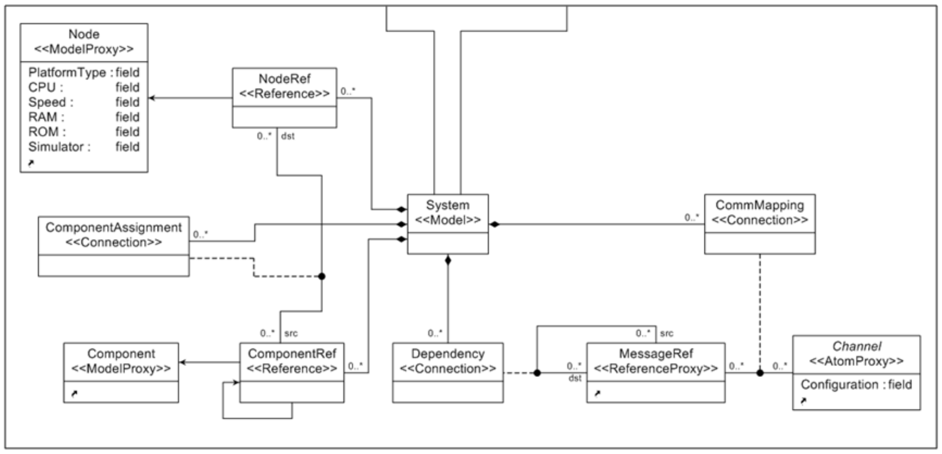

Figure 1.14: Details from deployment sublanguage.

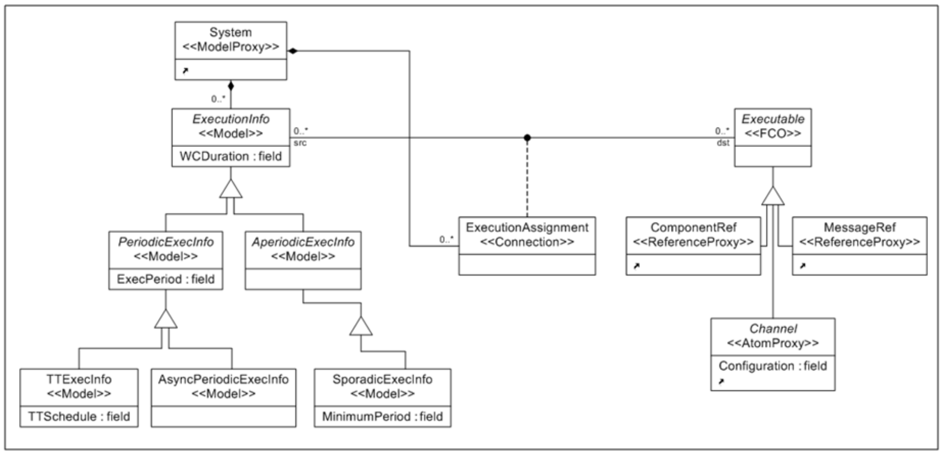

Figure 1.15: Details from timing sublanguage.

parameters such as period and worst-case execution time that must be given by the designer. Automated scheduling analysis fills in the schedule fields.

1.6 Interpreter Development Architecture

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 approach can be considered as an implementation of the tool integration ideas in [14], but with variations of the details included in the design language. Specifically we propose the following main ideas:

1. A front end graphical modeling language which integrates into existing design work flows[1]. This has already been described (ESMoL).

2. A single transformation from front end models to an intermediate language explicitly representing a flattened semantic model, including parameters and objects to imply a precise behavioral semantics. We differ from the approach in [14] by abstracting the dynamic component interface behavior using passive control design techniques. We rely on the resulting robustness and compositionality of the passive approach to simplify design analysis and implementation, as our design language is specific to distributed embedded control systems. Note that we only have preliminary support for passive control design and analysis – this is a work in progress (see [15] for preliminary work in this area).

3. Both the front-end and intermediate languages support platform-based design [3], separating component-level concerns and interaction concerns where possible. The platform for this language is the time-triggered architecture[5], and the tool suite supports deployment to a time-triggered execution environment[11] in order to preserve synchronous execution semantics.

4. Generation of analysis models and code from the intermediate language using simple template generation techniques[2].

5. Representation of structural and behavioral concepts from the intersection of the semantic domains of integrated tools as primitives in the front end language.

6. Round-trip incorporation of calculated analysis results into the modeling environment to maintain consistency as models pass between design phases.

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 1.1: Acquisition relation: Details of the transformation step taking ESMoL relation objects (left column), and producing a single relation for each collection representing an ESMoL_Abstract data acquisition specification.

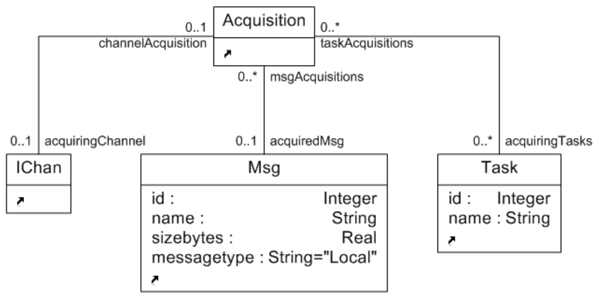

Figure 1.16: Acquisition relation in ESMoL Abstract. Acquisition represents data arriving from the environment. The abstract syntax graph model collects all of the information scattered over multiple relations in the user-facing language, and stores them in one relation for consumption by the synthesis interpreters. Each of these communication relations supports one sender and multiple receivers.

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.

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.

1.6.1 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. 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.

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 that both by constraint checking, and by the use of a single ESMoL_Abstract message 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.

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} is the set of objects of type $Type$, and obj_{Type} 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).

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 1.1 are as follows:

- CA **ComponentAssignment**: (the dashed connection shown in Fig. 1.10) assigns a task to run on a particular processor (id_N).
- AC **AcquisitionConnection**: (the directed connection from processor ports to component message ports) assigns a hardware input peripheral (modeled as an object of type $IChan$) to a message structure compatible with 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 1.1 in a compact form (see Fig. 1.16), so that all of the consumers of the input data get the same consistent structural view of the model. The modeling tools provide a programming interface for traversing, reading, and editing the models.

The collected relations are more easily processed by the synthesis interpreters, as they avoid extra traversals to gather the other relations. For acquisition and actuation (below), data transfers are timed in the current version of the execution environment. Work is currently underway to extend the language to fully support event-driven messaging and activation of sporadic tasks.

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. Table 1.2 gives the details of the transformation from relations in ESMoL to the actuation relation in ESMoL_Abstract. Fig. 1.17 shows the structure of the result.

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

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

Table 1.2: Actuation relation transformation.

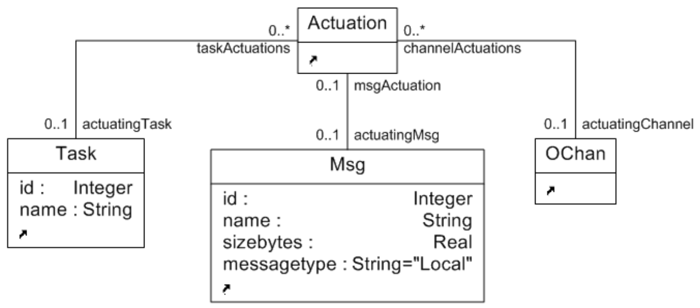

Figure 1.17: Actuation relation in ESMoL Abstract. Actuation represents the flow of data back into the environment.

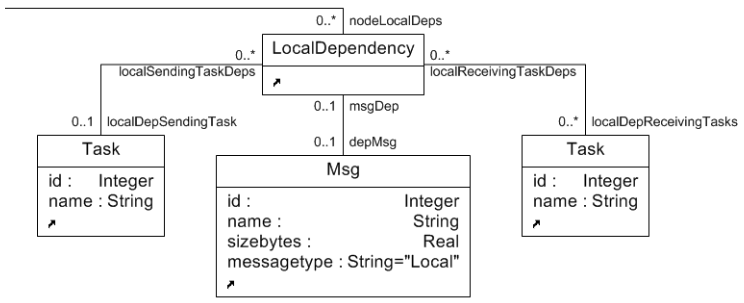

Figure 1.18: Local dependency relation in ESMoL Abstract. Local data transfers take place between components on the same processing node.

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

Table 1.3: Local (processor-local) data dependency relation.

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 1.3 gives the transformation details.

Bus Transfers: Data Movement Between Nodes

Bus transfers are slightly more complicated, as they involve two or more endpoints. Table 1.4 along with Fig. 1.19 show the details. The platform-specific code generators produce separate files for each processing node (especially as the nodes may be heterogeneous), so the send and receive relations are modeled separately. Fig. 1.20 shows an example of the objects and parameters based on our design example. The object network is an instance of the abstract language construct shown in Fig. 1.19.

1.6.2 Stage 2 Outputs: Analysis Models and Code

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.

The scheduling tool is a good example of the second-stage generation process, but we will cover it in greater detail in a later section. We use the CTemplate engine[16] called from C++ code to perform the generation tasks.

1.6.3 Execution Environment

The tool suite includes a simple, portable time-triggered virtual machine[11] which can run on generic Linux or FreeRTOS to implement timed execution of tasks and messages. The virtual machine is a lightweight implementation of the time-triggered architecture[5] (see [11] for details). 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

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 1.4: Transmit relation in the abstract graph. This represents the sender side of a remote data transfer between components. The Receives relation has an identical structure, but cardinality is different. Each Receives relation object represents a receiver in a remote data transfer between components. A single sender may have multiple receivers attached to the same message.

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 1.5: Receive relation in the abstract graph. Each *Receives* relation object represents a receivers in a remote data transfer between components. A single sender may have multiple receivers attached to the same message.

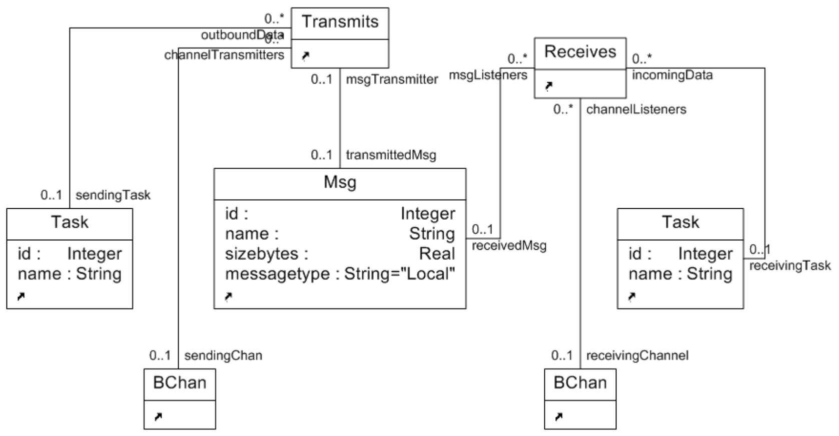

Figure 1.19: Transmit and receive relations in ESMoL Abstract. These represent the endpoints of data transfers between nodes. We use two separate relation objects to represent this concept to control cardinality: each message has exactly one sender (transmit relation), but possibly multiple receivers (receive relation).

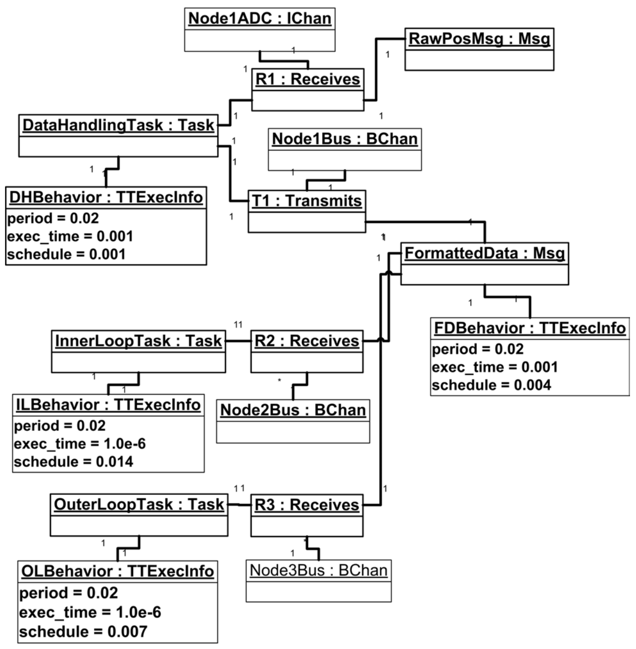

Figure 1.20: Abstract semantic model of a variant of the message structure from Figs. 1.7 and 1.10. The object diagram shows relation objects of type ”‘Transmits”’ and ”‘Receives”’, which associate message objects (and their associated behavior objects) to sending and receiving tasks, and to node-specific communication channel configurations used to transmit the message. This relation allows us to assemble behavioral information provided by the tasks, messages, and platform into a single model. For example, the schedule interpreter uses information from all of these elements to create an input specification for the schedule solver.

```
//////////////////////////////// SCHEDULE ////////////////////////////////

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 1.6: Template for the virtual machine task wrapper code. The Stage 2 FRODO interpreter invokes this template to create the wrapper code shown below (Table 1.7).

its output variables. Data structures describe the invocation times of each task and any other necessary parameters. The configuration also includes a schedule for time-triggered messaging. Each message instance includes invocation time as well as local buffer addresses where the virtual machine can store the computed cyclic schedule.

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.

The template description for the platform-specific code generator appears in Table 1.6, with the resulting generated code (for the Quad Integrator model) in Table 1.7. The message relations in ESMoL_Abstract (shown above) help in the collection of model objects for particular generation tasks, and also prevent ambiguous resolution of complex ESMoL relationships by different generators.

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. For the curious, the LocalDependency relations do not appear in this table. The scheduler creates constraints that must be satisfied for each local dependency. 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).

```
/////////////////////////////////// SCHEDULE ///////////////////////////////////

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 1.7: Generated code for the task wrappers and schedule structures of the Quad Integrator model.

1.7 Scheduling in Model-Based Embedded Design Tools

The control design provides task periods and profiling provides execution time specifications for each component instance. Data transfer rates and overhead parameters are stored in the platform model. [2] describes the actual constraint model details, which are an extension of earlier work [17]. The Gecode constraint programming tool [18] solves these constraints for task release times 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. The passive control design provides stability guarantees, even in the face of timing jitter or limited message loss. Hence (within the bounds of acceptable delays) the scheduler does not have to account for timing uncertainty beyond that inherent in the platform as long as the component abstraction is preserved in all interpretations that use the deployment information.

As the most mature analysis translator in our tools, the syntactic translation to the scheduling model provides the best conceptual illustration of the integration process. Fig. 1.21 shows a model transformation distilling details from the ESMoL tools and creating a scheduling problem model whose syntax represents the proper sets of behaviors. The task and message release time results (if the schedule is feasible) are fed back to the modeling framework as configuration parameters. We describe the steps indicated in the diagram here:

1. 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 (which is an instance of a model in the scheduling tool specification language).
3. The scheduling problem specification is parsed to import the model into the constraint generation environment.
4. The hyperperiod length determines the number of instances required for each task and message.

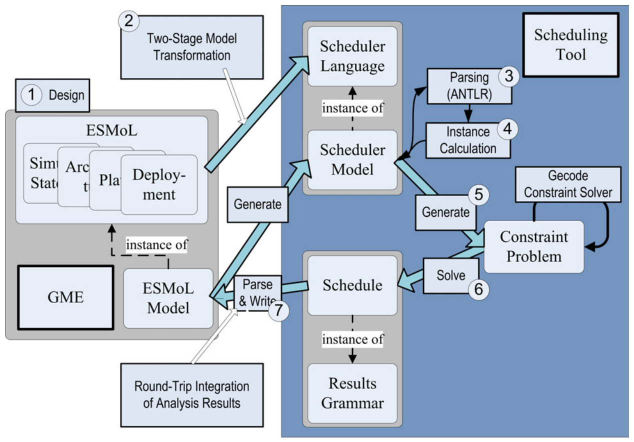

Figure 1.21: Integration of the scheduling model by round-trip structural transformation between the language of the modeling tools and the analysis language.

5. Instance relationships become constraints in Gecode after translation.
6. Gecode solves the constraint problem, possibly indicating infeasibility. If a valid schedule results, then it is written out to a file (which is an instance of a model in the very simple scheduling results language).
7. The results are imported into the ESMoL model and written to the appropriate objects.

1.7.1 Specification of Scheduling Problems

Listing 1.1 shows an example of a distributed schedule specification with 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.
- **Latency** specifies a timing constraint between two tasks in the system. This could model end-to-end timing constraints between a sensor and an actuator, for example. The maximum latency is given in seconds.

Listing 1.1: Scheduling problem specification.

```
Resolution 2us
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}}
Sync {{SYNCMSGNAME}} 1.25ms 0s
{{#MSG_SECTION}}Msg {{MSGNAME}} {{MSGSIZE}} {{SENDTASK}} {{RECVTASKS}}
{{MSG_SECTION}}
{{BUS_SECTION}}
{{#LATENCY_SECTION}}Latency {{LATENCY}} {{SENDTASK}} {{RECVTASK}}
{{LATENCY_SECTION}}
```

Table 1.8: Stage 2 Interpreter Template for the Scheduling Specification

```
Proc P1 100MHz 50us 10us
Task T1 =50Hz 8us
Task T2 =100Hz 10us

Proc P2 100MHz 40us 12us
Task T1 =50Hz 10us
Task T2 =100Hz 10us

Proc P3 100MHz 50us 12us
Task T1 =25Hz 10us
Task T2 =50Hz 5us

Bus B12 1Mb 0us
Msg M1 16B P1/T1 P2/T1

Bus B23 1Mb 0us
Msg M2 2B  P2/T1 P3/T1
Msg M3 4B  P3/T2 P2/T2

Latency 35us P1/T1 P2/T1
% Latency loop
Latency 100us P2/T1 P2/T2
```

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 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[3].

Scheduling specifications are created in the Stage 2 interpreter by the template shown in Table 1.8. 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. Table 1.9 contains the spec generated for the Quad Integrator example described above.

Producing the Proc and Comp lines from the model API is straightforward, as these are simple traversals of the model, respecting the model hierarchy. Each generated line uses parameters from the respective model object. The parameters are shown only in the generated output, though the object diagram in In order to produce each Msg line, many relations must be collected (as shown in table 1.3) and distilled into the right relationships. This requires complex traversal code. To write a generator similar to this one, the developer

```
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 1.9: Generated scheduling spec for the Quad Integrator example.

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.

1.7.2 CLP Models for Scheduling

In time-triggered communications messages are pre-scheduled on the network, and schedule calculation must ensure that messages and their sending and receiving tasks appear at appropriate times in the schedule, as the protocol does not block for message transfers. For our testing and experimentation we make the following assumptions regarding system models to be scheduled:

Our scheduling model is probably most closely related to the approach described in [19]. Our framework does not support preemptive schedules for tasks, but we do allow message broadcasts on the bus (one message dependency for each transmission). In [19], dependencies are captured as binary decision variables, where we use global cumulative constraints to avoid explicitly representing exclusive access to resources. We share the approach that an explicit architectural mapping (functions to processors) should be characterized up front during design. Incorporating their task preemption modeling approach could allow us to extend the flexibility of our scheduling tool.

- All tasks are strictly periodic, or each task is released exactly on a periodic schedule. For controllers of plants with continuous-time dynamics this assumption may adversely affect schedulability and performance[20], but for deterministic replicated execution of plants having discrete dynamics (i.e. for DES or hybrid models) the assumption is still necessary to ensure consistency of the dynamic state between replicas[5].
- The worst-case execution time (WCET) is known for each task on a given processor.
- Tasks assigned to the same processor are not preemptive.
- Message buses are shared by multiple processors.

- Processors may connect to multiple communication buses.
- Messages are not strictly periodic (to ease scheduling), but must be delivered on time.
- Message transfer times are known.
- A feasible bus schedule does not allow preemption of message transfers.
- Messages are broadcast to all nodes on a bus.
- On task release input data are already available for use in input buffers. Tasks write output data to output buffers at completion times. The communication framework transfers data and updates these buffers.

All processing nodes execute tasks within the same hyperperiod (repetition window[17]). The hyperperiod is the least common multiple of the periods of tasks in the system. Let H be the hyperperiod length. Each task T_i has maximum duration (WCET) D_i and period P_i. Task T_i runs $\frac{H}{P_i}$ times during a full cycle of the schedule. The individual instances (or jobs) of T_i are denoted $T_{i,j}$ where $j \in [1, ...\frac{H}{P_i}]$. Each task instance $T_{i,j}$ has a start time $R_{i,j}$ which is to be determined given duration $D_{i,j}$. The same structure is required for messages. Message M_i has (transfer time) length L_i and instances $M_{i,k}$ having transfer times starting at $S_{i,k}$.

The basic concepts from finite-domain (FD) constraint programming needed for this exposition are covered here in summary only. A constraint problem is modeled as a set of integer variables and a collection of different types of constraints modeling the relationships between those variables. Constraints typically include many arithmetic equalities and inequalities. Any valid solution must satisfy all of the constraints. In our case the variables represent the start times of the various task and message instances ($R_{i,j}$ and $S_{i,k}$). The finiteness of FD constraints refers to the fact that all of these variables are positive and have an explicit upper bound. Here all variables initially range over the hyperperiod of the schedule. FD solvers search the solution space by propagating constraints and branching over assignment possibilities. We use the Gecode finite domain constraint solver library[18] within our scheduling tool. A thorough discussion of finite-domain modeling and propagation-based solvers can be found in [21].

1.7.3 Task Ordering

For each task T_i the instances are ordered by the following set of constraints, representing strict periodicity exactly as in [17]:

$$R_{i,j+1} = R_{i,j} + D_{i,j} \ \forall j \in [1, ...(\frac{H}{P_i} - 1)] \tag{1.1}$$

For all jobs on the same processor, a global serialization constraint is also imposed to represent non-preemption. The global serialization constraint forces all variable assignments from the given set to be distinct, and to differ by at least a constant value (duration). For efficiency (and model conciseness), global distinctness constraints are used instead of n^2 constraints representing the exclusivity of all possible pairs[22, 17]. These constraints provide a higher-order abstraction of resource utilization in scheduling models[22]. Constant durations are specified for each variable in the constraint, in a pair with the start time. For processor p, if I_p is the index set of all task instances on p, then we write

$$\forall i \in I_p, \forall j \in [1, ...\frac{H}{P_i}] \ ser(R_{i,j}, D_i). \tag{1.2}$$

Here the function *ser* represents the global constraint implementation of the set of distinctness constraints. *ser* enforces the constraint that no two tasks (or messages) may overlap on the same processor (bus). For our non-preemptive model, this represents the set of all permutations of tasks (messages) on the processor (bus). The additional ordering constraints reduce that search space.

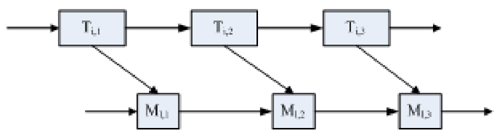

Figure 1.22: Dependency constraints for instances of a message and its sending task.

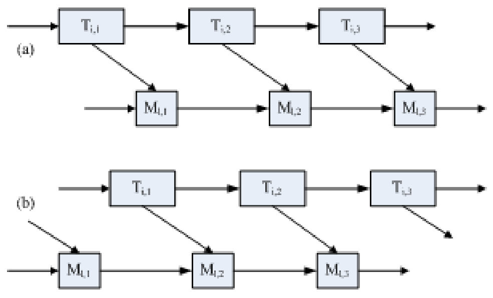

Figure 1.23: Two possible valid orderings of constraints for instances of a message and its sending task within one hyperperiod.

1.7.4 Message Ordering

The existing approach to message ordering constraints in [17] explicitly represents the relationships between sender and message instances. To insure that messages were sent between successive sender instances, constraints of the form

$$R_{i,j} + D_i \leqslant S_{l,j}, S_{l,j} + L_l \leqslant R_{i,j+1} \tag{1.3}$$

are used to capture the requirement that each message instance ($M_{l,j}$) begins after its sending task instance ($T_{i,j}$) and ends prior to the start of the next sending task instance ($T_{i,j+1}$). This models the logical execution time (LET) constraints as in [23, 24, 25]. For this description we have assumed that messages and their senders occur at the same rate, so both types of instances are indexed by j. The resulting ordering can be visualized as in Fig. 1.22. The matched rate constraint is not a requirement of the tool in general, and was only chosen for clarity of illustration.

Message instances are all serialized on a given bus, just as for tasks on a processor. No constraints are imposed on the relationship between message instances and their receivers, as this is handled by latency constraints. We propose the following alternative formulation for message ordering, which is more flexible than the approach presenting in [17] and at least as efficient for many useful cases. We will first describe the single-rate synchronous case, and then discuss constraints for multi-rate data transfers.

For task T_i sending message M_l (as above), let

$$S_{l,j+1} - S_{l,j} \geqslant P_i - D_i \tag{1.4}$$

and

$$(H - S_{l,\frac{H}{P_i}}) + S_{l,1} \geqslant P_i - D_i \tag{1.5}$$

represent the minimum distance between messages as discussed in [26]. The second expression (1.5) ensures the minimum distance between the last task in the cycle and the first task of the next cycle. To ensure the ordering as shown in the diagram we serialize all of the instances of both sender ($R_{i,j}$) and message ($S_{l,j}$) together using a single global serialization constraint:

$$\forall j \in [1, \ldots \frac{H}{P_i}],\ ser((R_{i,j}, D_i), (S_{l,j}, L_l)) \tag{1.6}$$

Notice that this approach represents both valid ways of placing the message instances with respect to the senders (see Fig. 1.23)).

Handling the multi-rate case requires a small modification to the new constraints. We assume that the receiver rate is slower than the sender rate (decimation). This is the only case that makes sense, as interpolation could be more efficiently performed by simply keeping and reusing messages on the receiving end. Suppose $j \in [1, \ldots J]$, $k \in [1, \ldots K]$, and that K divides J (written $K|J$). This condition is not strictly necessary, but we leave its relaxation as an exercise for the interested reader. Then we can re-state equations 1.4 and 1.5 as follows to include the decimation term:

$$S_{l,j+1} - S_{l,j} \geqslant (P_i \frac{J}{K} - D_i) \tag{1.7}$$

and

$$(H - S_{l,\frac{H}{P_i}}) + S_{l,1} \geqslant (P_i \frac{J}{K} - D_i) \tag{1.8}$$

Note that for the simple case $\frac{J}{K} = 2$, this combination of constraints now represents all four possible proper placements of message instances with respect to the sending tasks, increasing the flexibility of the multi-rate scheduling approach presented in [17]. The cost is an increase in the size of the search space, which we attempt to offset by our choice of heuristics.

Multiple buses are handled in a straightforward way – a task on a node that sends and receives messages on different buses has constraints for each of the attached communication media. While such constraints are easy to formulate, the likelihood of feasibility for those constraints must decrease as more buses add constraints to the start time for the same task. A more practical approach (if possible) would be to break the task into multiple tasks, each interacting only with a single bus. This relaxes some of the constraints on that task, while still ensuring that the task's processor resource is shared correctly.

1.7.5 Latency Constraints

Latency constraints enforce the maximum acceptable response time between the start of a task and the end of a (possibly) different task. Latencies were originally modeled in [17] as a disjunctive constraint for handling the possible wrap-around at the end of the message cycle. The disjunctive constraints apparently created problems for the constraint solver, leading to a two-stage approach to solving the problem [17]. In the two-step approach, the solver is run once without the latency constraints to capture a valid ordering of tasks and messages (to break symmetries). Then the task and message order determined by the solver is used to add the latency constraints.

Latency modeling is greatly complicated by multi-rate tasks and their dependencies. An initial ordering capture may lead to an infeasible schedule, without a clear path to backtrack to a different ordering. Our latency modeling approach does not sacrifice feasibility, but also may not lead to a valid solution. Implicit in selection of a timing bound for a pair of tasks is the set of chains of dependencies between t hem. These could be captured in the model, but path enumeration can easily lead to a combinatorial explosion of paths. Rather, our approach is to place constraints on the pairs of instances of the starting and ending tasks that try to get them within the specified distance. Then proper ordering must be checked post-facto (i.e. did all of the intervening tasks and messages occur in the proper order). Our argument is that this approach could be verified operationally. For example, re-simulating the control design with the new scheduling order may be a good approach to determine whether time delays introduced by schedule computation have adversely affected the controller performance. To converge on a working solution, the designers could iteratively generate a schedule for a given design, simulate the control loops running over that schedule, adjust rate parameters (or make other design changes), and repeat.

The latency model is constructed as follows:

1. Remove redundant constraints – latency specifications that are larger than the period of either the sending or receiving task.

2. For each remaining latency specification, create n^2 reified linear constraints, each requiring that a single sender instance and a single receiver instance fall within the specified time distance. Reified constraints have an explicit boolean control variable.

$$(S_{j,l} - S_{i,k} \leqslant l_m) \Leftrightarrow b_n \tag{1.9}$$

   In Eq. 1.9 we see sending task instance $T_{i,k}$, receiving task instance $T_{j,l}$, latency bound l_m, and boolean control variable b_n.

3. Add a constraint on the control variables, so that at least one of the conditions has to be true (Eq. 1.10).

$$\sum_n b_n > 0 \tag{1.10}$$

The size of the constraint problem for latencies is not ideal, but the final constraint generally ensures that search will be efficient. Many of the n^2 constraints are infeasible or redundant, so the number could be greatly reduced if necessary. Practically speaking, we will allow the solver to handle the elimination of redundant and infeasible subsets of those constraints until problem size or solution speed become an issue. Tests to date have not shown any problems (on medium-sized models).

1.7.6 Solution Concerns

The Gecode FD solver provides pre-programmed branching heuristics (as well as capabilities for building custom branchings). We branch first on the start time variables (end times are implied), subdividing intervals at the midpoint. This seems to give a good spacing for tasks and messages in the results. Latency branching comes second, starting with the maximum latency for each specification. No effort is made to optimize the schedule to minimize latencies – the first feasible latency is taken in each case.

1.7.7 Limitations

The scheduler has a number of limitations:

- We do not directly support mode switching. For specification languages that allow mode switching only at the end of the hyperperiod, individual modes can be formulated as separate schedule problems. Finer-grained switching is not supported.
- We do not support preemptive scheduling of tasks or messages.
- Time-triggered hardware protocols such as TTP/C or FlexRay subdivide hyperperiods into fixed-length slots and limit message transfers to ease schedule calculation. Compatibility with these protocols would require additional constraints to model the starting and ending times of the slots as well as communication limits (e.g., limiting a task to send a message only once per slot).
- 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.

- The scheduling model is discretized to a single resolution for everything in the system. In reality different processors and buses have different timing characteristics. We also do not account for uncertainty due to time synchronization, though it could be included in the overhead parameters. We also do not account for the inter-frame transmission gap of the underlying communication medium, though it would be straightforward to add to the constraints.
- 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 [27]).

1.8 Related Work

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[28]s. AADL projects also include integration with the Cheddar scheduling tool[29]. Cheddar is an extensible analysis framework which includes a number of classic real-time scheduling algorithms[30].
- Giotto[23] is a modeling language for time-triggered tasks running on a single processor. Giotto uses a simple greedy algorithm to compute schedules. ESMoL extends these capabilities with distributed architecture models and appropriate schedule calculation. Also worth noting, the TDL (Timing Definition Language) is the successor to Giotto, and extends the language and tools with the notion of modules (software components)[25]. 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[31].
- The Metropolis modeling framework[32] 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[33] 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. [34] and [35]) for deadlock and timing analysis.
- RTComposer[36] 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. Our work differs in that we translate requirements and timing information into constraint models rather than using formal automata models for determining schedulability.
- The DECOS toolchain [37] combines a number of existing 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. 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 as appropriate.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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_E%mbedded_C.pdf

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, “Extensible and scalable time triggered scheduling,” in *ACSD ’05: Proc. of the Fith Intl. Conf. on App. of Concurrency to System Design*, June 2005, pp. 132–141.

[20] A. Anta and P. Tabuada, “On the benefits of relaxing the periodicity assumption for networked control systems over CAN,” in *RTSS ’09: Real-Time Systems Symposium*, 2009.

[21] G. Tack, “Constraint propagation – models, techniques, implementation,” Ph.D. dissertation, Saarland University, Jan 2009.

[22] N. Beldiceanu and M. Carlsson, “A new multi-resource cumulatives constraint with negative heights,” in *CP*, ser. LNCS, P. van Hentenryck, Ed. Springer, 2002, pp. 63–79.

[23] 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

[24] S. Matic and T. A. Henzinger, “Trading end-to-end latency for composability,” in *Proc. of the 26th Annual Real-Time Systems Symposium (RTSS).* IEEE Computer Society Press, 2005, pp. 99–110.

[25] 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.

[26] C. Ekelin and J. Jonsson, “Solving embedded systems scheduling problems using constraint programming,” Chalmers Univ. of Technology, Tech. Rep. TR 00-12, 2000. [Online]. Available: http://www.ce.chalmers.se/~cekelin

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.