# On Semantically Consistent Tool Integration in Model-Based Cyber-Physical Systems Design

JOSEPH PORTER, GRAHAM HEMINGWAY, HARMON NINE,
NICHOLAS KOTTENSTETTE, CHRIS VAN BUSKIRK, GABOR KARSAI
and JANOS SZTIPANOVITS
Vanderbilt University Institute for Software Integrated Systems
2015 Terrace Place, Nashville, TN 37203 USA
jporter@isis.vanderbilt.edu

Cyber-physical systems software is increasingly produced by generation from models. Integrating disparate tools and generators into a unified tool chain is problematic due to the difficulty of maintaining semantic consistency of design artifacts across translations to and from integrated tool domains. Our two-stage transformation approach provides abstraction at both the model design and tool integration points. Semantic consistency between domains is more easily maintained because the first stage (to an abstract intermediate language) contains all semantically relevant transformation details, reducing the second stage (generators) to syntax-only translations where possible. We illustrate the design flow with a simple control design example.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.2 [**Software Engineering**]: Design Tools and Techniques; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms: Design

Additional Key Words and Phrases: Cyber-Physical Systems, Model-Integrated Computing



**Submitted to the SCPS Special Issue**

## 1. INTRODUCTION

Cyber-physical systems (CPSs) can be difficult to design due to the heterogeneous nature of the domains in which they must operate. Safety-critical control applications also require a level of formal verification, to prove to certification authorities (and therefore indirectly to the consuming public) that use of these devices entails only minimal danger.

This work was sponsored (in part) by the Air Force Office of Scientific Research, USAF, under grant/contract number FA9550-06-0312. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Modern control design and embedded development approaches incorporate code synthesis techniques as well as integrating analysis tools into a model-based design flow. Each of these designs, assessments, or generations are typically made independently by analysts trained in either an engineering domain (mechanical, electrical, or control design) or in computer science (schedulability, componentization, or deployment concerns). Artifact generation from models occurs at different stages of the design process: generating functional code, analysis models, and possibly platform specific wrapper code and configuration. Each analysis tool may have its own modeling language with distinct semantics. Semantic differences can lead to inconsistencies in the understanding of the design, so details must be kept consistent across design specialization domains and between model interpretation stages.

In current practice, much of the reconciliation of design discrepancies is still done by hand. Individual designers or review teams discover design inconsistencies. Manual reconciliation of issues occurs as individual designers receive assignments to modify and correct the design. Several large modeling tool projects (for example, AADL [John Hudak and Peter Feiler 2007] and Topcased[Pontisso and Chemouil 2006]) 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 tools is a serious issue in such efforts.

Supporting such heterogeneity of analysis tools and domains requires everyone involved in the design process to have a consistent view of design details. Reconciling semantics between formalisms and tools is costly and time-consuming. Often the effort can not be justified outside of academic research unless the results are applicable to numerous designs.

In the model integrated computing approach, domain specific modeling languages represent different aspects of the design, with the promise of consistently integrating different tools and techniques. We present here a sketch of a formal model-based approach which promises to tame many of the difficulties involved in consistently integrating heterogeneous analysis tools into a single workflow. Our approach can be considered as an implementation of the tool integration ideas in [Pinto et al. 2006], 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 workflows[Porter et al. 2009].

(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 [Pinto et al. 2006] 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 [Eyisi et al. 2009] for preliminary work in this area).

(3) Both the front-end and intermediate languages support platform-based design [Carloni et al. 2005], separating component-level concerns and interaction con-

cerns where possible. The platform for this language is the time-triggered architecture[Kopetz and Bauer 2003], and the tool suite supports deployment to a time-triggered execution environment[Thibodeaux 2008] in order to preserve synchronous execution semantics.

(4) Generation of analysis models and code from the intermediate language using simple template generation techniques[Porter et al. 2009].

(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.

The described approach is part of the ESMoL modeling language and software design suite[Porter et al. 2009]. ESMoL is a research experiment in the implementation of modeling tools to support compositional analysis and design frameworks. In our design flow a modeler imports Simulink/Stateflow models [The MathWorks, Inc. ] into the ESMoL language, adds architecture elements, platform design, and deployment concepts, then performs automated analysis and synthesizes code for execution on a time-triggered platform. We aim to support iterative development, as analysis data can flow back to earlier stages for re-design or re-evaluation. The language and tools also include preliminary support for requirements modeling, by allowing specification of maximal latency bounds between computational tasks in the design. Table I gives an overview of our toolchain design goals and philosophy, considering details from three different semantic models.

Our tools enforce a single view of the interpretation of design details in a design model. Consistency is required in both the structure of the design models (i.e. the componentization and its relations) as well as the behavior represented by the design models (i.e. the model of computation in which tasks are executed and their behavior within that model).

The model-integrated computing approach (MIC) [Karsai et al. 2003] facilitates the essential part of the consistency effort. Figure 1 depicts a design flow that includes a user-facing modeling language for design and a language for explicit representation of semantics. In the Generic Modeling Environment (GME) [Ledeczi et al. 2001] a designer creates a software design from an imported Simulink control design. 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 second-stage transformations are simpler, and are isolated from changes to the user language.

## 2. DESIGN EXAMPLE

Our design example controls a continuous-time system whose model represents a simplified version of a quadrotor UAV. Figure 2 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. 3 shows a Simulink model containing the simplified dynamics.

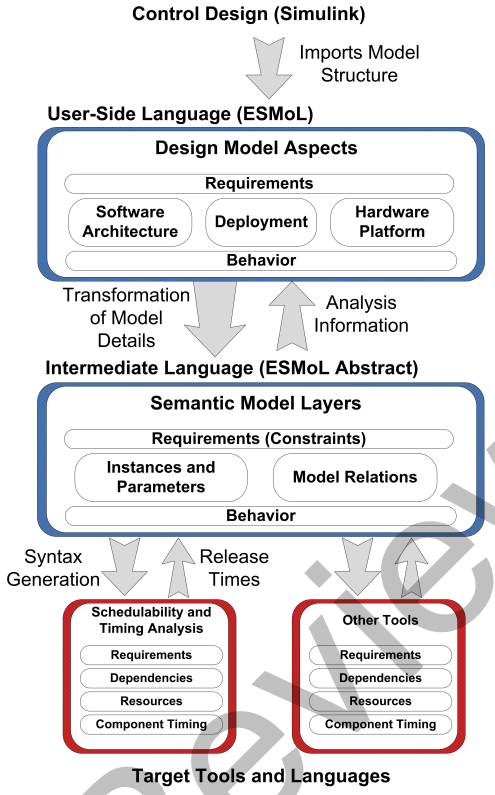


Fig. 1. Suggested flow of design models between design phases. During design iterations, analysis data can be transformed and fed back to designers.

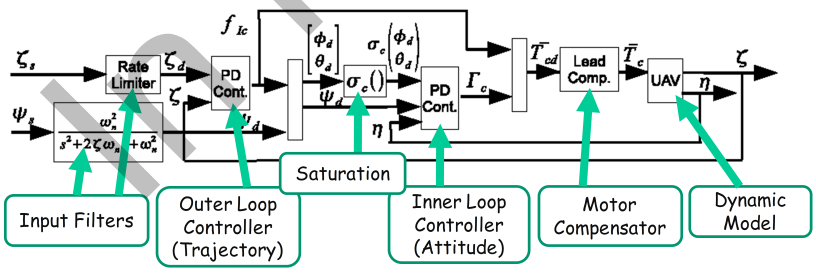


Fig. 2. Basic architecture for the quadrotor control problem.

For the fully detailed quadrotor model and a complete discussion of the control design philosophy, see [Kottenstette and Porter 2008]. The example model controls the stack of four integrators (and motor lag) using two nested PD control loops. The Plant block contains the integrator models. The two control loops (inner and outer, as shown) 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.

| Behavioral Consistency Goals | | |
|---|---|---|
| . | Execution Order Constraints | Timing Constraints |
| Dynamic Stability (Comp) | Passive controllers run synchronously at a fixed rate determined by control analysis and simulation. | Control functions execute in bounded time, measured for the given platform. Nominal sample rates are passive and stable. |
| Dynamic Stability (Intr) | Passive control design decreases the effects of delays from scheduling and platform timing jitter. Maximum acceptable sample delay is one possible abstraction for these effects. | Timing guarantees are not yet considered in our passive control design framework. |
| Scheduling (Comp) | We use a synchronous execution model to precompute component invocation order within tasks to eliminate local data hazards. | Each task has a known, bounded execution time (WCET/Deadline parameters are in the model). |
| Scheduling (Intr) | Message and task dependencies are translated to linear constraints, along with constraints to model resource utilization. | Scheduling achieves the desired sample rate and enforces latency bounds between tasks. A proposed delay abstraction could represent schedule slack, and latency could be used to budget slack. |
| Execution Environment (Comp) | Statically precomputed task release times are used to configure the generated tasks, and the VM enforces start times. | We assume bounded-time task execution. VM tasks can not be preempted, and so execute as quickly as possible to meet their deadline (WCET in this case). |
| Execution Environment (Intr) | Clocks on separate nodes are synchronized to support synchronous execution. Frame sync (null) messages are sent at the start of the common hyperperiod to keep nodes executing together. | Time-triggered schedules prevent collisions, so messages transfer deterministically. Measured transfer overhead is captured in the platform design, and used in scheduling calculations. |

Table I. Summary discussion of the behavioral consistency goals from the point of view of each design stage of our tools so far. The table covers relationships between three different semantic models: dynamic stability, schedulability, and deadlock freedom. The (Comp) notation refers to local (component/task) level concerns, and (Intr) refers to design concerns at the level of interactions between components, consistent with our platform abstraction. The ESMoL language constructs should embody the correct resolution of these design issues, and maintain the assumptions through all stages of design and execution.



Control design centers around the continuous-time abstraction of passive control [Kottenstette and Porter 2008]. Passivity provides a robust version of continuous-time dynamic stability which is insensitive to quantization effects [Fettweis 1986] and network delays [Chopra et al. 2008][Kottenstette et al. ] in digital control implementations. The passivity conditions help relax constraints on required component sample rates, increasing the system's tolerance to jitter and other timing variations.

Simulated stability analysis yields period parameters for the control components. For our discussion the exact nature of these parameters is not important, rather that they represent the same behaviors for all of the integrated tools. Passive design provides a guarantee of stable operation around a nominal sampling rate, as the

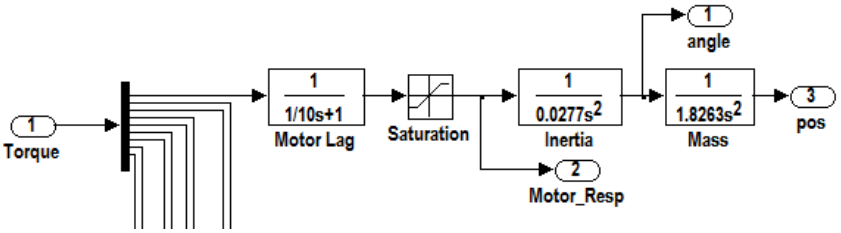


Fig. 3. Simplified quadrotor plant dynamics. The rotational dynamics have been removed to facilitate easier study of the behavior. The full model includes all of the dynamics. The signals lines leading off the picture are signal taps used to perform sector analysis to check stability (see Section 9).


Fig. 4. 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 deployment model.



system will tolerate a small number of lost or delayed data samples.

Figures 4 through 6 show different parts of the GME design model for the quad integrator controller. The model-integrated computing approach uses integrated sublanguages to represent models in the various aspects of the design.

—Fig. 4 shows an example of the component typing language. Message fields and their sizes are specified here, as well as component implementations and interfaces. 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).

—Fig. 5 portrays logical data dependencies between software component instances, independent of their distribution over different processors.

—Fig. 6 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. RS (Robostix) is an 8 bit

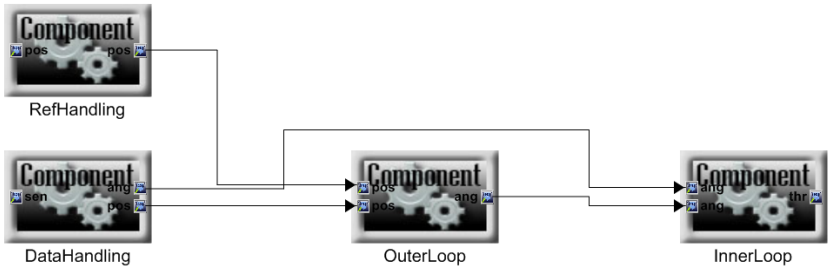


Fig. 5. Deployment: The logical architecture model specifies data dependencies between software component instances. The ports on the blocks represent data messaging interfaces into and out of the component.

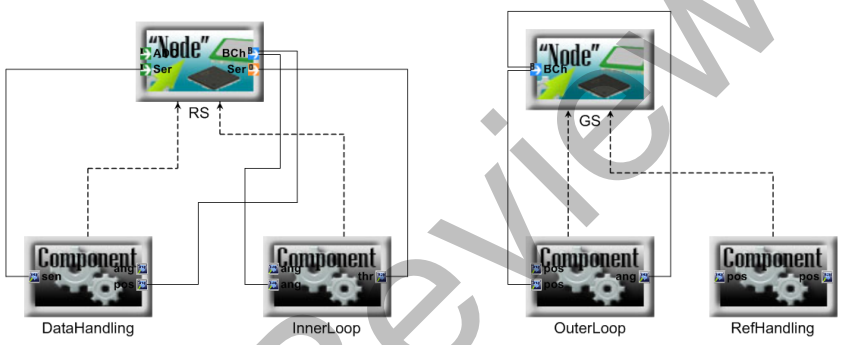


Fig. 6. Deployment: The platform mapping model specifies the hardware components that will support the transfer of data to and from each processing node, the software components that will send and receive the data, and the message instances in which the data will be carried. Dashed arrows represent assignment of components to their respective processor, and solid lines represent assignment of message instances (component ports) to communication channels (port objects) on the processor.



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

—An execution model (not shown) 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. Our example model assigns each component to its own task.

Behavior of the deployed components depends on execution timing of the functions on the platform, the calculated schedule, and coordination between distributed tasks. The calculated execution schedule can be used to simulate the control design with additional delays to assess the impact of the platform on performance. Fig. 7 shows simulated effects of additional delays in the control data paths. The top of the figure is the direct synchronous digital implementation of the control system.

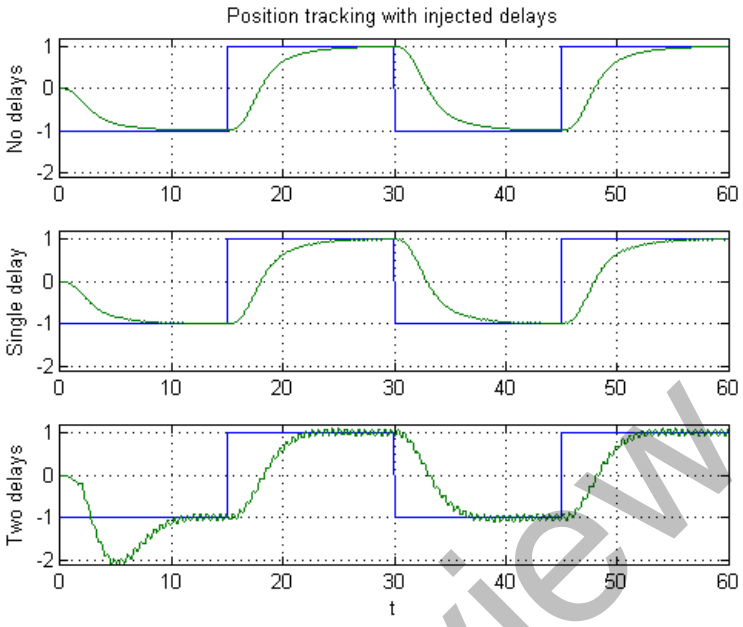


Fig. 7. Effects of increasing delays in the synchronous scheduling of the control nodes.



The two successive plots show the effects of adding first one and then two extra delay elements in each control path. The reference input frequency and amplitudes were chosen to lie on the edge of the stable operating point of the design. This is not meant to be an exhaustive verification of the control design, rather an illustration of the gradual oscillation and overshoot degradation that occurs as delays increase.

Section 3 discusses our approach to semantic consistency. Sections 4 and 5 cover some details of the model transformations made by the tools to get to analysis and code artifacts. Sections 6 and 7 briefly describe our approach and tools for two of the three semantic models for the language (time-triggered scheduling and the execution environment). Section 8 discusses our approach to assessing passivity of the controller, and compares controller performance between the simulated design model and the generated controller. Sections 9 and 10 cover related work and future work, respectively. Section 11 concludes.

## 3. SEMANTIC CONSISTENCY

The front end language, abstract intermediate language, and the transformation between them were designed and implemented together, in order to provide useful abstractions to the user and to the tool integrator. Both languages are specified as layered metamodels [Karsai et al. 2003] which include elements for requirements, design structures, and behavioral models. 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. While we do not have space to describe the full structural and behavioral semantics of the language,

Table I gives a brief summary of the goals of our correctness arguments within the full semantics of the language. The front-end language, the intermediate language, and the first stage transformation work together to represent these assumptions and guarantees consistently.

Each analysis translation works from a single view of the design model, keeping tool-specific translations simple. As a simple example, consider the model shown in Fig. 5. Component DataHandling sends data messages to the other two components, as denoted by the dependency arrows. The deployment view (Fig. 6) 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 actual 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. 13). 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, and then reduces this complex set of relations 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.

The passivity of the control components is an interface condition that must be maintained in order to ensure the proper behavior of the design. In the user language we select Simulink subsystems to be used as the specification of software components in the modeling language. Each will be implemented as a synchronous C function. The selected component objects translate directly to component instances in the semantic language. A component is an object with a unique name (i.e. InnerLoop), and information to find or generate its implementation in C (in this case, the filename and model path to the Simulink subsystem "'QuadIntegrator/InnerLoop"'). Though the robustness provided by the passive abstraction is not yet directly captured in our models, it would be a more complex concept. A useful abstraction of the behavior might be the maximum amount of tolerable time delay on each path (in synchronous time ticks) for the worst-case input for a particular control loop.

Note that the language does not currently encode or enforce passive control design structure. However, we require that Simulink models encapsulate functionality in subsystem blocks. As passivity is an interface condition, our preservation of the component structure and synchronous execution semantics should preserve the passive assumptions of the design, or help designers quickly discover where the abstraction has broken down. We will take a look at the validity of our approach in Section 9.

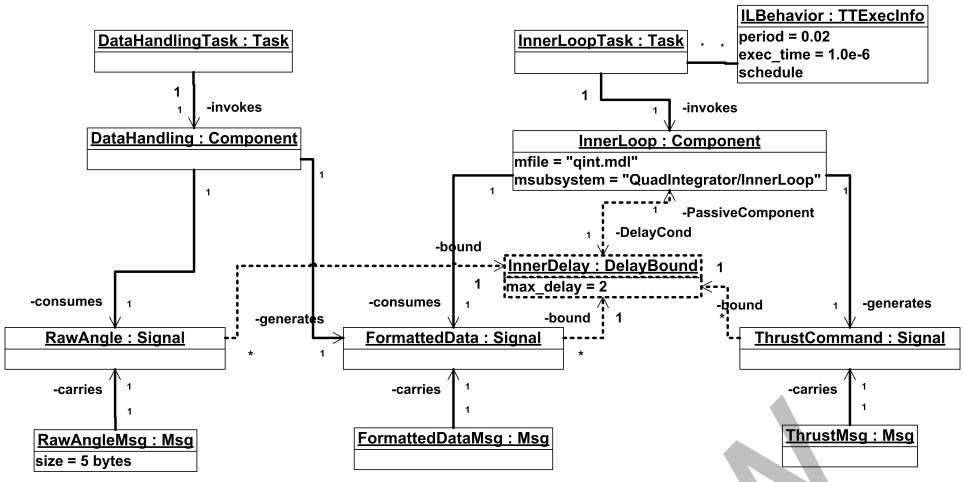


Fig. 8. Capturing a delay bound in the abstract model. The dashed object and connectors represent the concept to be represented. In practice each high-level design concept or bound is related to many other design objects and their parameters.

## 4. TRANSFORMATION TO ABSTRACT SYNTAX GRAPH

We describe here the transformation 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 assumption of a single semantic message object for all participants in the data interchange. The Signal object (not shown) in the abstract graph represents the transfer of a single datum to or from the message.

The transformations shown 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. The actual transformation is implemented in C++ using the UDM modeling API[Magyari et al. 2003].

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).

### 4.1 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 ESMoL relations shown in table II are as follows:

—*CA* **ComponentAssignment**: (the dashed connection shown in Fig. 6) assigns a task to run on a particular processor ($id_N$).

—*AC* **AcquisitionConnection**: (the directed connection from processor ports to

| Specified ESMoL Relation Sets | Semantic Construct |
|---|---|
| $CA_{id_N} = \{(obj_{Node}, obj_{CompInst}) \mid id(obj_{Node}) = id_N\}$<br><br>$AC_{id_{Ch}} = \{(obj_{IChan}, obj_{MsgInst}) \mid id(obj_{IChan}) = id_{Ch}\}$<br><br>$NC_{id_N} = \{(obj_{Node}, obj_{IChan}) \mid id(obj_{Node}) = id_N \wedge parent(obj_{IChan}) = obj_{Node}\}$<br><br>$CC = \{(obj_{CompInst}, obj_{MsgInst}) \mid parent(obj_{MsgInst}) = obj_{CompInst}\}$ | $Acq = \{(obj_{MsgInst}, obj_{CompInst}, obj_N, obj_{Ch}) \mid (obj_N, obj_{CompInst}) \in CA_{id_N} \wedge (obj_{Ch}, obj_{MsgInst}) \in AC_{id_{Ch}} \wedge (obj_N, obj_{IChan}) \in NC_{id_N} \wedge (obj_{CompInst}, obj_{MsgInst}) \in CC$ |

Table II. Depiction of the transformation step that takes ESMoL relation objects on the left, and produces a single relation for each collection representing a data acquisition specfication.

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 II in a compact form (see Fig. 9), 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 fully support event-driven messaging and activation of sporadic tasks.



### 4.2 Actuation: From Data Back 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. Fig. 9 shows the resulting construct in the abstract language.

### 4.3 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 $CA_{id_N}$).

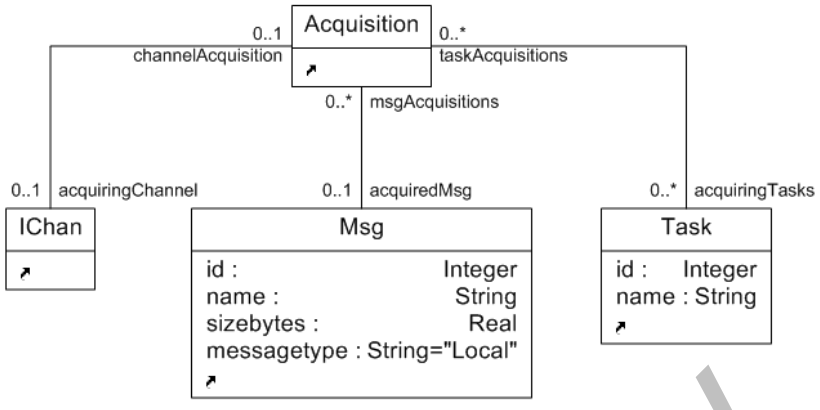


Fig. 9. Acquistion relation in ESMoL Abstract. Acquistion represents data arriving from the environment. The abstract syntax graph model collects all of the information scattered over multiple relations in the user-facing language, and stores them in one relation for consumption by the synthesis interpreters. Each of these communication relations supports one sender and multiple receivers.



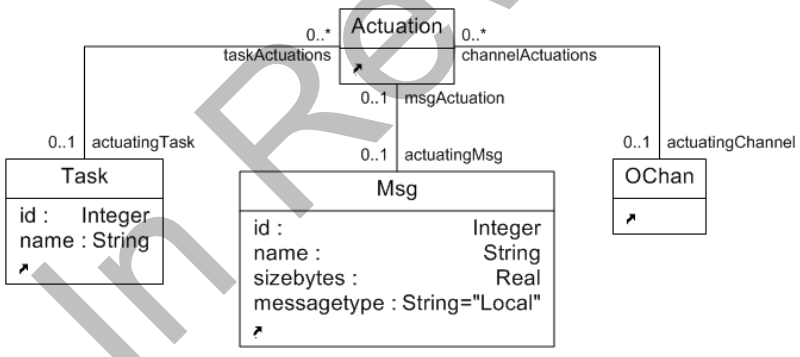


Fig. 10. Actuation relation in ESMoL Abstract. Actuation represents the flow of data back into the environment.

### 4.4 Bus Transfers: Data Movement Between Nodes

Bus transfers are slightly more complicated, as they involve two or more endpoints. Table IV along with Fig. 12 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. 13 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. 12.

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

Table III. Local (processor-local) data dependency relation.

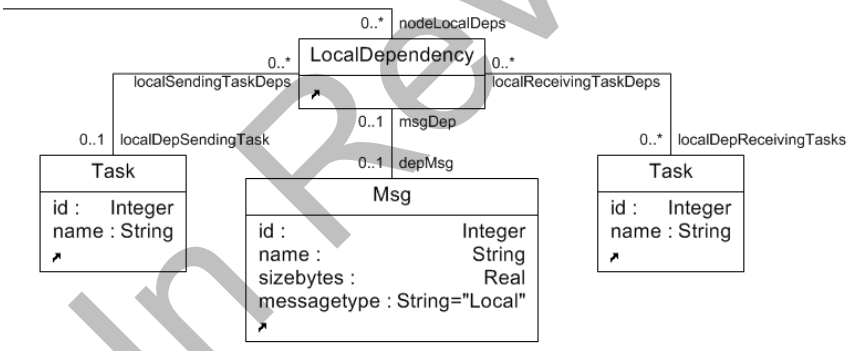


Fig. 11. Local dependency relation in ESMoL Abstract. Local data transfers take place between components on the same processing node.



## 5. SYNTHESIS OF CODE AND ANALYSIS MODELS

### 5.1 Functional Code

[Neema and Karsai 2004] describes a much earlier version of the modeling language and tools created to support automotive software development. The Simulink model importer and graph transformation-based code generation for the control functions are documented in [Neema and Karsai 2004]. The tools still support the importing of Simulink functions, and generation from Simulink subsystem blocks to synchronous C code. The main advantage over the Simulink Real-Time Workshop code generator [The MathWorks, Inc. ] is the easy customizability of our genera-

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

Table IV. 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 differnt. 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.

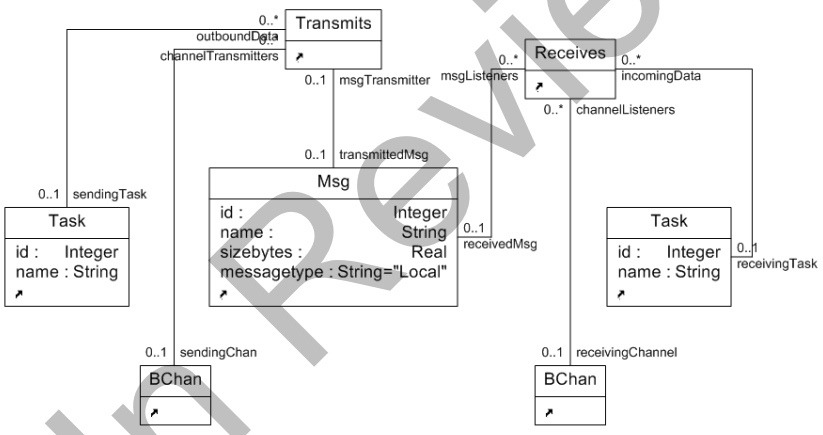


Fig. 12. Transmit and receive relations in ESMoL Abstract. These represent the endpoints of data transfers between nodes.



tor. Generation is based on simple code templates for each functional block which can be created new or customized by the user. Transformation details are also open to inspection and modification.

## 5.2 Scheduler

The scheduling tool is a good example of the second-stage generation process. We will consider one example of an analysis artifact. For generation other than the functional code, we use the CTemplate engine[Google ] called from C++ code.

The host section of the template appears below, followed by an example generated from the quad integrator model:

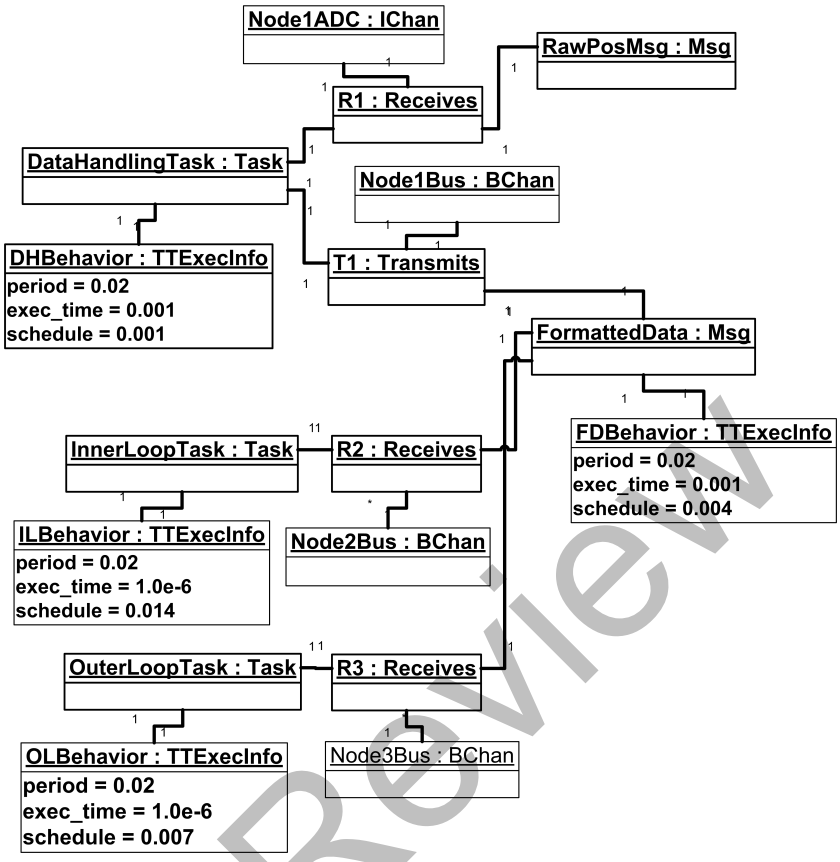


Fig. 13. Abstract semantic model of the message structure from Figs. 5 and 6. Object diagram showing relation objects of type "'Transmits"' and "'Receives"' associate message objects (and their associated behavior objects) to sending and receiving tasks, and to node-specific communication channel configurations used to transmit the message. This relation allows us to assemble behavioral information provided by the tasks, messages, and platform into a single model. The schedule interpreter uses information from all of these semantic elements to create input for the schedule solver.



```
{{#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}}

---------------------------------------------------------------------

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
```

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. Our model has two processors (Proc lines, above), each of which runs multiple components (Comp) and has at least one local data dependency (Msg).

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 Fig. 13 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 III) and distilled into the right relationships. This requires complex traversal code. To write a 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 configuration file.

### 5.3 Platform-Specific Code

The template description for the platform-specific code generator is shown here:

```
////////////////////////////////// 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}}
```

```
------------------------------------------------------------------------

//////////////////////////////// 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 }
}
```



The FRODO virtual machine generation template brings together all of the semantic 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 configuration 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).

## 6. SCHEDULING

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. [Porter et al. 2009] describes the actual constraint model details, which are an extension of earlier work [Schild and Würtz 2000]. The Gecode constraint programming tool [Schulte et al. ] 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

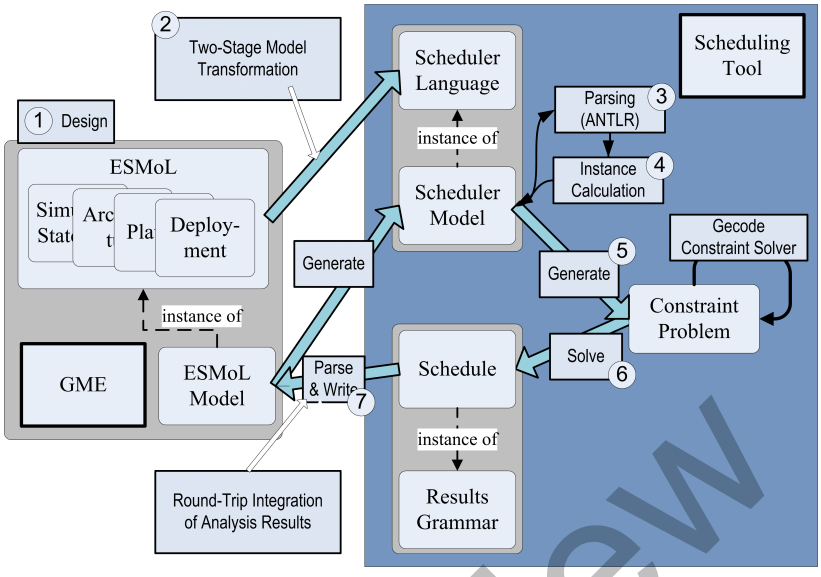


Fig. 14. Integration of the scheduling model by round-trip structural transformation between the language of the modeling tools and the analysis language.



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. 14 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.

In the example model, the configuration file created from the template is processed by the scheduling tool. The resulting start times are then important back into the model.

## 7. EXECUTION ENVIRONMENT

The tool suite includes a simple, portable time-triggered virtual machine[Thibodeaux 2008] 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[Kopetz and Bauer 2003] (see [Thibodeaux 2008] for details). Execution requires configuration with the computed cyclic schedule. Code generated for the virtual machine conforms to a particular structure – each task reads its input variables, invokes its component functions, and writes its output variables. Data structures describe the invocation times of each task and any other necessary parameters. The configuration also includes time-triggered messaging. Each message instance includes invocation time as well as local buffer addresses

where the virtual machine should store the computed cyclic schedule. Code generated for the virtual machine conforms to a particular structure – each task reads its input variables, invokes its component functions, and writes its output variables. Data structures describe the invocation times of each task and any other necessary parameters. The configuration also includes time-triggered messaging. Each message instance includes invocation time as well as local buffer addresses where the virtual machine should store data.

The virtual machine enforces timing constraints to ensure correct execution. The passive control design provides slack so that dynamic stability can tolerate imperfect execution of the tasks. The virtual machine must provide clock synchronization to preserve dependency ordering and avoid collisions.

The biggest limitations in our implementation are limited fault tolerance and lack of support for executing multiple components in a task. Our virtual machine can detect local deadline overruns, but at this stage it simply reports them. We use a single master frame message at the start of each hyperperiod to maintain synchronization, and so are not robust to loss of synchronization. Testing shows that we can constructively prevent some forms of input and output data corruption, but have not yet performed the formal analysis of those guards. The lack of support for multiple components in a task is not a technical issue, rather we have not yet implemented the synchronous data flow scheduling of those components.

## 8. EVALUATION

Passive control theory promises a constructive form of robust design with respect to dynamic stability, and in the presence of network delays and quantization errors [Fettweis 1986] [Chopra et al. 2008] [Kottenstette and Antsaklis 2008]. Passive control systems can be constructed from passive components with the application of a few simple interconnectivity rules. Previous work focused on constructive methods includes the approach of encoding passive interconnections in a modeling language [Eyisi et al. 2009] or enforcing passive interconnections at run time using transformations on the data variables and a data exchange system known as a power junction [Kottenstette et al. 2009].

Essentially, a passive component or system introduces no new energy into its environment. Output energy is limited to the amount of energy received as input plus any previously stored energy. Digital control systems with fast dynamics (such as the quadrotor) use a zero-order hold to convert control values produced at discrete time instants into step functions held over a continuous interval of time. For certain inputs and state trajectories, the hold process can introduce small amounts of new energy into the environment. The sector bounds analysis proposed by Zames [Zames 1966] can be used to assess the amount of "active" (energy-producing) behavior which we can expect from our design.

For linear (and some nonlinear) system models, sector bounds may be computed symbolically during system analysis. Each component is assigned a real interval ($[a, b]\ -\infty < a \leq b \leq \infty,\ b \geq 0$) representing a range of possible input/output behaviors. Components whose bounds fall in the interval $[0, \infty]$ are passive (and very close to stable). Zames also presents rules for computing sector bounds for systems based on calculated component bounds and different types of interconnec-

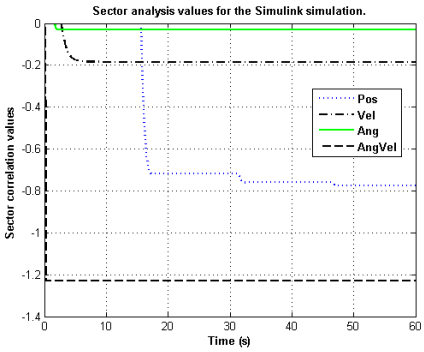


Fig. 15. Sector search values for the simulated quad integrator example.

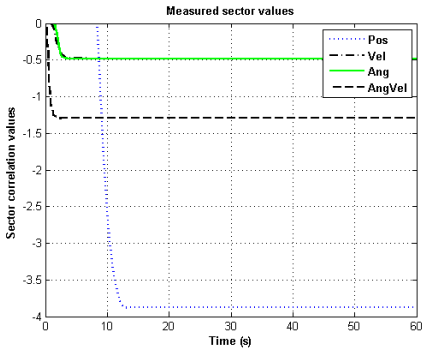


Fig. 16. Sector search values for execution on the control platform (including schedule effects).

tions between components [Zames 1966]. For our quadrotor system, we expect the bound $a$ to be small and negative, corresponding to the small amount of "active" behavior due to the hold effects. [Kottenstette and Porter 2008] shows a formula for selecting stabilizing gains based on knowledge of the value of $a$ ($k < \frac{-1}{a}\, a < 0$).

Empirical verification of the sector bounds is not sound (for all possible inputs or for discrete time), but can still be a useful technique nonetheless. Beyond a useful engineering measurement to see whether measured bounds lie close to the predicted values, if sectors are estimated conservatively measurements which exceed the bounds are significant. This indicates that the selected control gains may not have been sufficient to stabilize the system with the platform delay effects included in the closed-loop controller. Our test platform is a hardware-in-the-loop simulation environment that allows us to execute the plant model, and to interface the controller using actual serial links so that it appears to be a genuine plant with respect to the hardware interface, software interface, and timing delays.

## 8.1 Passivity, Stability, and Sector Search

| Control Signal | Simulated sector values | Measured sector values |
|---|---|---|
| Position | -.7757 | -3.8811 |
| Velocity | -0.1856 | -0.4830 |
| Angle | -0.0295 | -0.4831 |
| Angular Velocity | -1.2292 | -1.2963 |

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

Table V shows that we have not accounted for all of the effects introduced by the platform. Platform effects caused a significant deviation from our ideal sector estimates. Although the initial platform gains satisfied the sector stability conditions, the overall system response resulted in significant over-shoot. After adjusting the gains to minimize overshoot, the overall sector estimates were closer to those for the idealized case but still more negative as would be expected from the additional

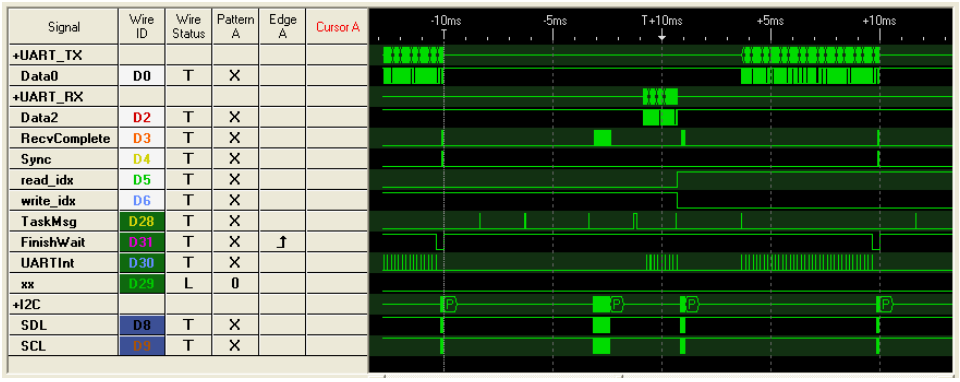


Fig. 17. Timing measurements for the computed schedule.

platform effects. We hope that such sector measures can be used to better pinpoint sub-platform effects and guide controller tuning.

A more careful review of the data showed that we had successfully mitigated the delay effects, but differences in the implementation of single-precision floating-point operations and libraries on three different platforms (simulation on x86 vs. actual ARM PXA and ATMega128 with floating-point emulation) led to some fairly significant differences in behavior. Section 8.3 below shows the tracking behavior of both the simulation and adjusted controller.

## 8.2 Schedulability

Fig. 17 shows measurements on the ATMega128 (inner loop controller) that confirm our schedule calculations. The running schedule releases tasks and transfers messages at the proper times, and that no tasks overrun deadlines. We briefly describe here the signals represented in the figure:

—**UART_TX and Data0** Show the transfer of data from the hardware-in-the loop simulator through a serial link to the controller. The LogicPort probe interprets the raw signal data (Data0) as an RS-232 exchange (UART_TX).

—**UART_RX and Data2** Show the transfer of data to the hardware simulator through the serial link.

—**RecvComplete, read_idx, write_idx, and UARTInt** Portray various debugging signals for tracking UART interrupts and double buffering indices.

—**Sync** Indicates the start of the hyperperiod. Each hyperperiod runs for 20 ms.

—**TaskMsg** Contains the start of the message transfers between processors as well as the run time of the control tasks. The tasks start at 4, 7, and 14 ms from the reference sync pulse.

—**I2C, SDL, and SCL** Show the data transfers between processors over the shared synchronous $I^2C$ bus. The I2C signal is a protocol-decoded version of the raw SDL and SCL data lines. The small 'P' indicates the end of a message transfer. Transfers occur at the start of each hyperperiod, and at offsets of 7 and 11 ms.

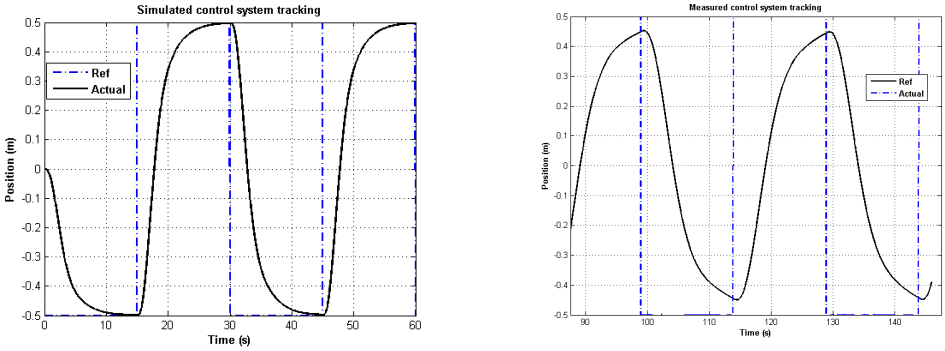


Fig. 18. Square wave input tracking for the simulated quad integrator example and for for a hardware-in-the-loop simulation of the actual control platform.

### 8.3 Execution Concerns: Reference Tracking

Fig. 18 depicts the simulated versus actual position tracking behavior of the generated control system. The figure on the right shows the tracking behavior with the reduced control gains described in Section 8.1. As expected, the platform tracking responds more slowly due to delays.

## 9. RELATED WORK

A number of projects seek to bring together tools and techniques which would be applicable to CPS design:

—AADL is a modeling language and standard for specifying deployments of control system designs in data networks[John Hudak and Peter Feiler 2007]. AADL also integrates with the Cheddar real-time scheduling tool[Singhoff et al. 2005].

—The Metropolis modeling framework[Balarin et al. 2003] has similar goals, and aims to give designers tools to create verifiable system models. Metropolis integrates with SystemC, SPIN model-checking, and tools for schedule and timing analysis.

—The DECOS toolchain [Herzner and et al 2007] combines a number of existing tools (e.g. the TTech tools, SCADE from Esterel Technologies, and others) but the hardware platform modeling and analysis aspects are not covered.

—Topcased[Pontisso and Chemouil 2006] 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. [Tripakis et al. 2005] and [Park 2007]).

—The Timing Definition Language (TDL) is the successor to the Giotto time-triggered modeling language, and inspired parts of our tool design [Farcas et al. 2005] [Naderlinger et al. 2007]. TDL does not directly model controller semantics.

Consistency of integrated tools is an issue for all of these efforts. We are creating a modeling language to experiment with design decoupling techniques, rapid integration of heterogeneous tools, and representation of formal semantics. Many of

the listed projects are too large to allow experimentation with the semantics of the entire toolchain, 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.

## 10. FUTURE WORK: DEADLOCK ANALYSIS USING BIP

Deadlock analysis of a model in our tools requires consideration of both the control system components and the inter-component communications controlled from within the virtual machine. Analysis of the control system components (within tasks) is simplified as we assume a synchronous dataflow execution paradigm. This guarantees deadlock free local execution as long as the overall network is schedulable [Lee and Messerschmitt 1987]. Of greater interest is deadlock analysis of the communications controllers and virtual machine execution when the control components have passed a message off for transfer.

The BIP [Basu 2008] [Sifakis 2005] tool chain includes facilities for performing deadlock analysis on componentized models. BIP (which stands for Behavior, Interaction, Priority) is built upon the labeled transition system formalism for component behaviors, allowing model checking with the IF Toolset [Bozga et al. 2004]. Interactions are used to compose components and are able to maintain properties such as deadlock freedom through composition. Priorities are applied between interactions to select from a group of enabled transitions.

The structure of the inter-component communications controllers and virtual machine do not vary from one model to another – their execution behavior is only altered by the schedule data. Once a valid schedule has been generated, the intermediate model contains all of the information necessary to translate into an equivalent BIP model. A model interpreter will generate the BIP output from the semantic model using templatized BIP-component models substituting scheduling information where necessary and creating interactions and priorities between components as appropriate. Significant initial effort must go into creating the templatized component models to accurately reflect the execution semantics of the communications controllers and virtual machine. This effort is the foundation upon which our BIP integration is based.

While we are in the early design stages of BIP integration, it already promises to be able to faithfully capture the semantics of our models. The LTS paradigm is a natural adaptation from the approach taken in the design of much of the virtual machine, and the available interaction structures are rich enough to represent our distributed environment. Additional parameters may need to be added to the component templates if other types of analysis beyond deadlock are required.

## 11. CONCLUSION

We have seen that even for a few of the design concerns of a cyber-physical system numerous details must be addressed. The two-stage model transformation approach seems to offer hope for organizing these details. Still, the first stage transformation may be prohibitively complex to maintain or extend. Our approach to control design correctness through passivity and time-triggered execution also

offers hope for constructively verifying designs. We also hope that extensions to include asynchronous execution semantics will be well-managed using the constructively verification approach provided by the BIP formalism. However, our results showed that we still have other issues to address before the promise of constructively correct model-based design can be realized. We must deal with the effects of quantization differences between platforms. All in all the approach seems sound, and future work will address the shortcomings.

## REFERENCES


Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Paserone, C., and Sangiovanni-Vincentelli, A. L. 2003. Metropolis: an integrated electronic system design environment. *IEEE Computer 36,* 4 (April).

Basu, A. 2008. Component-based Modeling of Heterogeneous Real Time Systems in BIP. Ph.D. thesis, VERIMAG, 2 avenue de Vignate 38610, Gieres, France.

Bozga, M., Graf, S., Ober, I., Ober, I., and Sifakis, J. 2004. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems*. LNCS, vol. 3185. 237–267.

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

Chopra, N., Berestesky, P., and Spong, M. 2008. Bilateral teleoperation over unreliable communication networks. *IEEE Trans. on Control Sys. Technology 16,* 2 (March), 304–313.

Eyisi, E., Porter, J., Hall, J., Kottenstette, N., Koutsoukos, X., and Sztipanovits, J. 2009. 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.

Farcas, E., Farcas, C., Pree, W., and Templ, J. 2005. 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)*. ACM Press, New York, NY, 31–39.

Fettweis, A. 1986. Wave digital filters: theory and practice. *Proc. of the IEEE 74,* 2, 270 – 327.

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

Herzner, W. and et al, R. S. 2007. 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.

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

Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T. 2003. Model-integrated development of embedded software. *Proc. of the IEEE 91,* 1 (January), 145–164.

Kopetz, H. and Bauer, G. 2003. The Time-Triggered Architecture. *Proc. of the IEEE 91,* 1 (Jan), 112–126.

Kottenstette, N. and Antsaklis, P. J. 2008. Time domain and frequency domain conditions for passivity. Tech. Rep. ISIS-2008-002, Inst. for Software Integrated Sys., Vanderbilt Univ. and Univ. of Notre Dame. November.

Kottenstette, N., Hall, J., Koutsoukos, X., Antsaklis, P., and Sztipanovits, J. 2009. Digital control of multiple discrete passive plants over networks. Tech. rep., Institute for Software Integrated Systems, Vanderbilt Univ. March.

Kottenstette, N., Koutsoukos, X., Hall, J., Antsaklis, P. J., and Sztipanovits, J. Passivity-based design of wireless networked control systems for robustness to time-varying delays. *IEEE RTSS 2008*, 15–24.

Kottenstette, N. and Porter, J. 2008. Digital passive attitude and altitude control schemes for quadrotor aircraft. Tech. Rep. ISIS-08-911, Inst. for Soft. Integrated Sys., Vanderbilt Univ.

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

Lee, E. A. and Messerschmitt, D. G. 1987. Synchronous data flow. *Proc. of the IEEE 75,* 9, 1235–1245.

Magyari, E., Bakay, A., Lang, A., and et al. 2003. Udm: An infrastructure for implementing domain-specific modeling languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling*.

Naderlinger, A., Pletzer, J., Pree, W., and Templ, J. 2007. Model-driven development of flexray-based systems with the timing definition language (tdl). In *Proc. of the 4th International ICSE workshop on Software Engineering for Automotive Systems*. Minneapolis.

Neema, S. and Karsai, G. 2004. Embedded control systems language for distributed processing (ECSL-DP). Tech. Rep. ISIS-04-505, Inst. for Software Integrated Sys., Vanderbilt Univ.

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

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

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

Porter, J., Karsai, G., and Sztipanovits, J. 2009. Towards a time-triggered schedule calculation tool to support model-based embedded software design. In *Proc. of ACM Intl. Conf. on Embedded Software (EMSOFT '09)*. Grenoble, France.

Porter, J., Karsai, G., Volgyesi, P., Nine, H., Humke, P., Hemingway, G., Thibodeaux, R., and Sztipanovits, J. 2009. 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*. Springer, Toulouse, France.

Schild, K. and Würtz, J. 2000. Scheduling of time-triggered real-time systems. *Constraints 5,* 4 (Oct.), 335–357.

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

Sifakis, J. 2005. A framework for component-based construction extended abstract. In *SEFM '05: 3rd IEEE Intl. Conf. on Software Eng. and Formal Methods*. IEEE Comp. Society, Washington, DC, 293–300.

Singhoff, F., Legrand, J., Nana, L., and Marcé, L. 2005. Scheduling and memory requirements analysis with AADL. *Ada Lett. XXV,* 4, 1–10.

The MathWorks, Inc. Simulink/Stateflow Tools. http://www.mathworks.com.

Thibodeaux, R. 2008. The specification and implementation of a model of computation. M.S. thesis, Vanderbilt Univ.

Tripakis, S., Sofronis, C., Caspi, P., and Curic, A. 2005. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst. 4,* 4, 779–818.

Zames, G. 1966. 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 Transactions on 11,* 2 (Apr), 228–238.