Towards Semantically Consistent Integration of Analysis Tools in Model-Based Embedded 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

1. INTRODUCTION

Distributed embedded control systems can be difficult to design due to the heterogeneous nature of the domains in which they must operate. Safety-critical 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.

Many tools exist to help embedded control system designers assess performance, stability, schedulability, and a host of other important properties. Each of these assessments are typically made by analysts trained in either an engineering domain (mechanical, electrical, or control design) or in computer science (schedulability, componentization, or deployment concerns). Additionally, each analysis tool may have its own modeling language with distinct semantics. Both situations can lead to inconsistencies in the understanding of the design.

In current practice, much of the reconciliation of design discrepancies is still done by hand. Design inconsistencies are discovered by individual designers or in review meetings. Manual reconciliation of issues occurs as individual designers receive assignments to modify and correct the design.

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 six main ideas:

—A front end graphical modeling language which integrates into existing embedded design workflows[Porter et al. 2009][Porter et al. 2009].

—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 (unambiguous, analyzable, and executable) behavioral semantics. We differ from the approach in [Pinto et al. 2006] by abstracting the dynamic behavior using passive control design techniques. We rely on the resulting robustness and compositionality of the passive approach to simplify design analysis and implementation.

—Both the front-end and intermediate languages support platform-based design [Carloni et al. 2005], separating component-level concerns and interaction concerns where possible.

—Generation of analysis models from the intermediate language using simple template generation techniques.

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

—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 embedded 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[Kopetz and Bauer 2003]. 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.

2. SEMANTIC CONSISTENCY

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. Fig. 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 an embedded system 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 simple abstract intermediate language whose elements are similar to those of the user

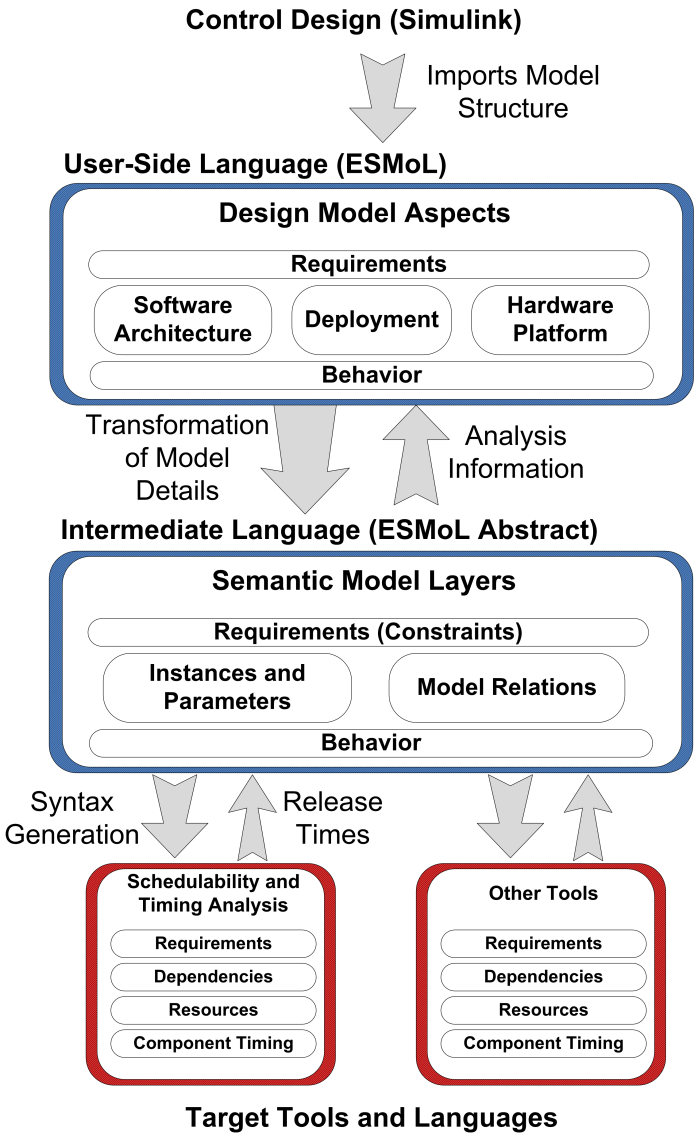

Fig. 1. Suggested flow of design models between design phases. During design iterations, analysis data can be transformed and fed back to designers.

language. Mappings from the intermediate language to analysis tools are one-to-one (if possible), and semantically insignificant relationships (i.e. those used for visualization) are removed. The first model transformation flattens the user model into the intermediate form, resolving parameters and special cases as needed. New elements are only added to the intermediate language as required by the analysis tools.

The front end language, abstract intermediate language, and the transformation

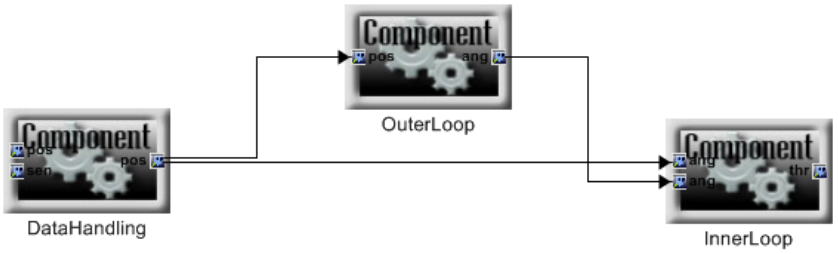

Fig. 2. Three components (blocks) with message transfer dependencies (arrows). The dependency connection ports on each component represent the message instance as seen by the component.

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.

Each analysis translation works from a single view of the design model, keeping tool-specific translations simple. As a simple example, consider Fig. 2, which comes from the user design model. Component DataHandling sends data messages to the other two components, as denoted by the dependency arrows. The behavioral semantics of this model are ambiguous at this point, as we have a dependencies but no timing specification. The deployment view (Fig. 3 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. 7). We still have some ambiguity. This is simply because different designers could make differing assumptions about the implementation of model details in their respective domains. Numerous message handling implementations would work adequately for the specified design. In the absence of other constraints (such as performance and efficiency), designers must choose implementation details. The message transfer could consist of a single broadcast to N receivers, or N individual message transfers, so these cases must be specified unambiguously.

The first stage transformation checks constraints to ensure that each object is used consistently 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 the message (provided by the scheduler, described below), including requested times to start transfers and duration of each message on the bus.

A more useful example of potential inconsistency appears in the interpretation of

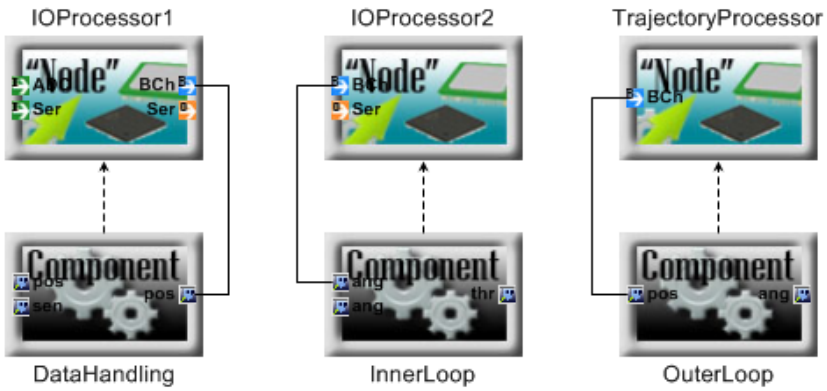

Fig. 3. Deployment model for the three components in the example. Dashed arrows represent assignment of components to their respective processor, and solid lines represent assignment of message instances (component ports) to communication channels (port objects) on the processor.

parameters. Assume that the designer of a code generator needs some way to flag a special case for generating task code, and so chooses a special interpretation of the task period parameter for the value zero. If the tools integrate schedulability analysis (and the scheduler is unaware of the special case, as often happens) then we have a condition that may introduce a subtle error into the design via a false positive result. In the two-stage framework, the interpretation of the special case would be represented in one place only. Its result would be to generate a different set of objects, relations, and/or parameters in the intermediate language, reducing opportunities for different interpreter writers to make inconsistent assumptions about message objects implementation.

The following sections (III, IV, and V) cover different aspects of a conceptual development process. We discuss semantics represented by the models, as well as information that could be provided back to the designers for further design iterations (see Fig. 1). We also discuss details of some of the intermediate language structures relevant to each design stage. While we do not have space to describe the full structural and behavioral semantics of the language, Table I gives a brief summary of our current correctness arguments within the full semantics. The front-end language, the intermediate language, and the first stage transformation work together to represent these assumptions and guarantees consistently.

3. DESIGN EXAMPLE

Our design example controls a continuous-time system whose model represents a simplified version of a quadrotor UAV. We exclude the nonlinear rotational dynamics of the actual quadrotor, but retain the stability characteristics. For the fully detailed quadrotor model, see [Kottenstette and Porter 2008]. The example model controls a stack of four integrators using two nested PD control loops, as shown in Fig. 4. The UAV block contains the integrator models. The two control loops (inner and outer, as shown) are implemented on separate processors, and the exe-

Behavioral Consistency Concepts		
.	Execution Order Constraints	Timing Constraints
Dynamic Stability (Comp)	Passive controllers execute 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. Below we propose one abstraction for these effects.	Timing guarantees are not yet considered in our passive control design framework.
Scheduling (Comp)	We use a synchronous data flow 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 concerns from the point of view of each design stage of our tools so far. The (Comp) notation refers to component (task) level concerns, and (Intr) refers to global interaction concerns.

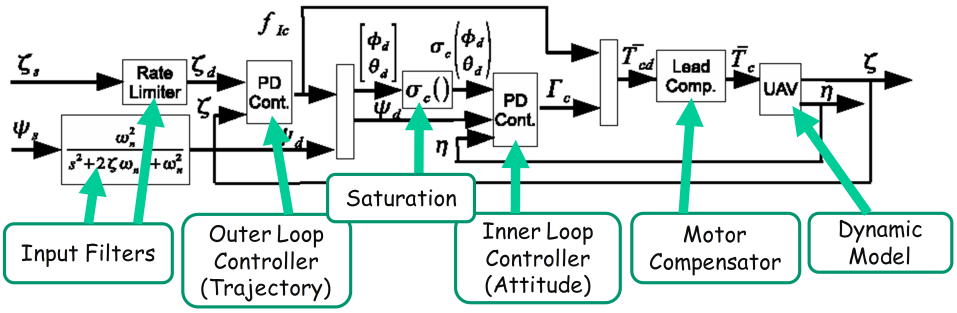

Fig. 4. Basic architecture for the quadrotor control problem.

cution of the components is controlled by a simple time-triggered virtual machine that releases tasks and messages at pre-calculated time instants.

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. Note that control performance requirements could create additional constraints, but we do not address those here.

Simulated stability analysis yields period parameters for the control components. Here they represent nominal sample periods, but conservative parameters could also be represented. 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 system will tolerate a small number of lost or delayed data samples.

Our modeling tools include explicit platform and deployment modeling[Porter et al. 2009]. 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. 5 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. 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. See [Kottenstette and Porter 2008] for details of the design and its validation.

The passivity of the control components is an interface condition ($Power_{in} \leq Power_{out}$) 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 specifi-

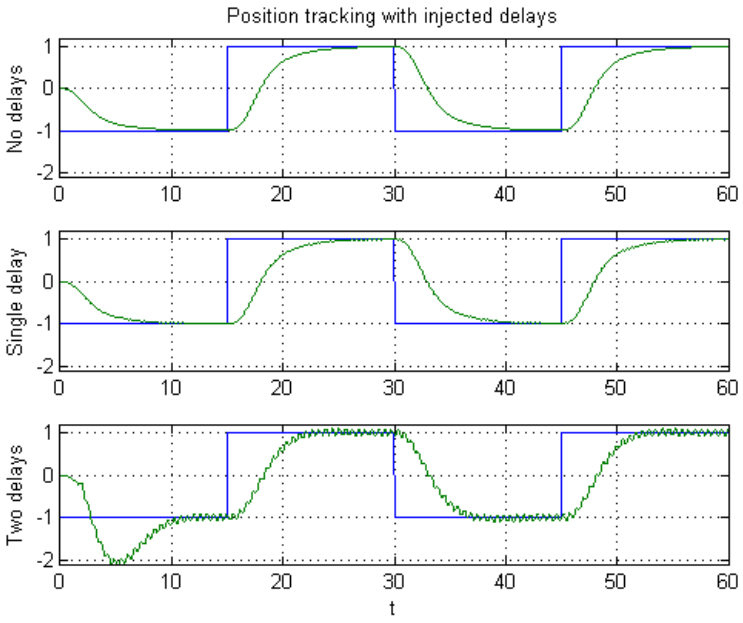

Fig. 5. Effects of increasing delays in the synchronous scheduling of the control nodes.

cation 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 directly captured in our models (yet), 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. Fig. 6 depicts a UML object diagram showing objects and relations in the abstract model that could represent such a concept. The translation from a passive control component embedded in a design would have to visit all design model connections involved in the particular control interaction, and create the object and relations shown. The first-stage model transformation would encode this structurally for all interpreters that make use of this bound object, ensuring that all of them work with the same semantic model. Note that this is only a step in the right direction, as we have nothing yet in the language to ensure consistency at the level of equations that use these parameters.

4. SCHEDULING

The control design provides task period and execution time specifications for each component instance. Data transfer rates and overhead parameters are found in the platform model. Messages represent dependencies between the tasks. The semantic

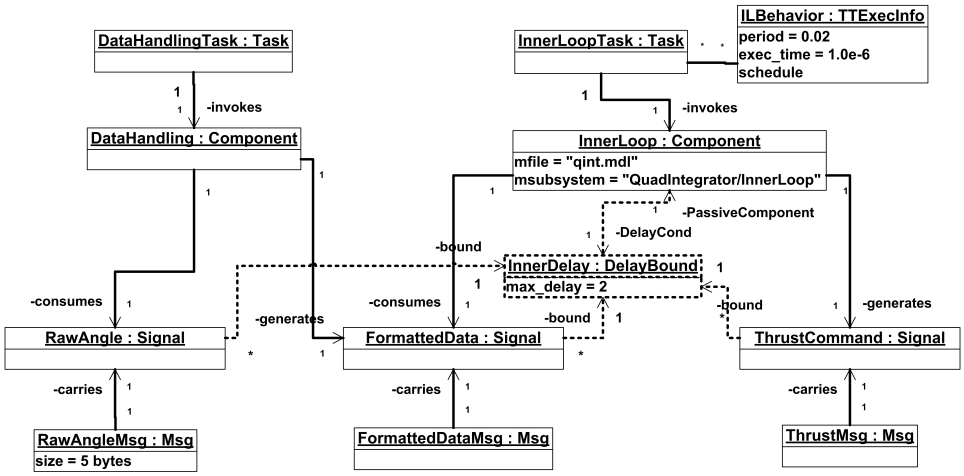

Fig. 6. Capturing a delay bound in the semantic model. The dashed object and connectors represent the concept to be represented. In practice each high-level design concept or bound is related to many other design object and their parameters.

model keeps a single set of relations describing the association of task, messages, and platform to get a single scheduling model. Task and message dependencies are given in the software architecture model. [Porter et al. 2009] describes the actual constraint model details. A constraint programming tool 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. 8 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.

5. 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

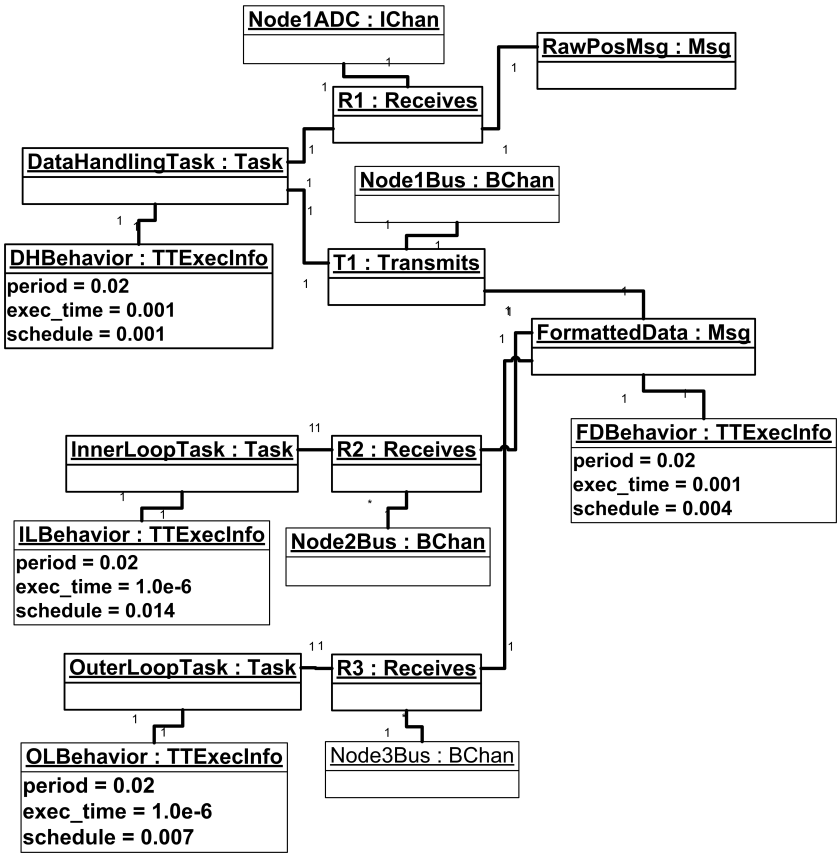

Fig. 7. Semantic model of the message structure from Figs. 2 and 3. Relation objects of type "'Transmits"' and "'Receives"' associate message objects (and their associated behavior objects) to sending and receiving tasks, and to node-specific communication channel configurations used to transmit the 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.

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.

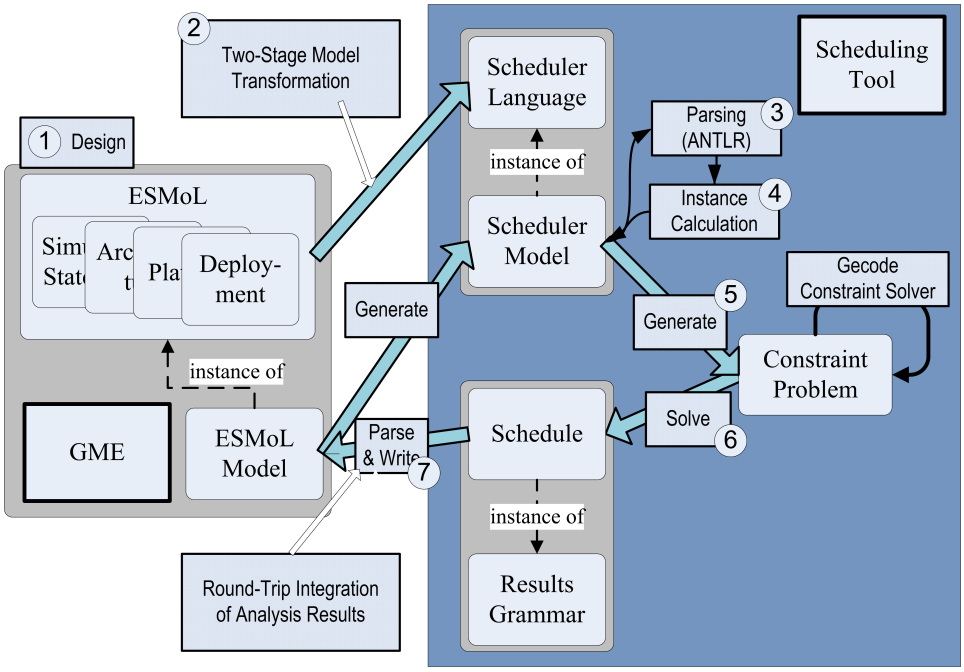

Fig. 8. Integration of the scheduling model by round-trip structural transformation between the language of the modeling tools and the analysis language.

The biggest limitations in our implementation are fault intolerance 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 are also not yet robust to data hazards such as floating point exceptions. 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.

6. 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; **?**] 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]. Interac-

tions 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.

7. RELATED WORK

A number of projects seek to bring together tools and techniques for embedded control system 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 projects also include integration 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, the SPIN model-checking tool, and other tools for schedule and timing analysis.

—The DECOS toolchain [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[mod] is a large tool integration effort that centers around the UML suite of software design languages.

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.

8. ACKNOWLEDGEMENTS

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.

REFERENCES

TOPCASED: The Open-Source Toolkit for Critical Systems. http://www.topcased.org/index.php.

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. *Control Systems Technology, IEEE Transactions on 16,* 2 (March), 304–313.

et al, W. H. 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.

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

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. *Proceedings of the IEEE 91,* 1 (January), 145–164.

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

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, Institute for Software Integrated Systems, Vanderbilt University.

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. *Proceedings of the IEEE 75,* 9, 1235–1245.

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.

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.

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 University.