Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation

Joseph Porter, Gábor Karsai, Péter Völgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and János Sztipanovits

Institute for Software Integrated Systems,
Vanderbilt University,
Nashville TN 37203, USA
jporter@isis.vanderbilt.edu,
http://www.isis.vanderbilt.edu

Abstract. While design automation for hardware systems is quite advanced, this is not the case for practical embedded systems. The current state-of-the-art is to use a software modeling environment and integrated development environment for code development and debugging, but these rarely include the sort of automatic synthesis and verification capabilities available in the VLSI domain. We present a model-based integration environment which uses a graphical architecture description language (EsMoL) to pull together control design, code and configuration generation, platform-specific simulation, and a number of other features useful for taming the heterogeneity inherent in safety-critical embedded control system designs. We describe concepts, elements, and development status for this suite of tools.

1 Introduction

Embedded software often operates in environments critical to human life and subject to our direct expectations. We assume that a handheld MP3 player will perform reliably, or that the unseen aircraft control system aboard our flight will function safely and correctly. Safety-critical embedded environments require far more care than provided by the current best practices in software development. Embedded systems design challenges are well-documented [1], but industrial practice still falls short of expectations for many kinds of embedded systems.

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

Software development uses UML (or similar) tools to capture concepts such as components, interactions, timing, fault handling, and deployment. Workflows

M.R.V. Chaudron (Ed.): MODELS 2008 Workshops, LNCS 5421, pp. 20–34, 2009.

focus on source code organization and management, followed by testing and debugging on target hardware. Physical and environmental constraints are not represented by the tools. At best such constraints may be provided as documentation to developers.

Complete systems rely on both aspects of a design. Designers lack tools to model the interactions between the hardware, software, and the environment with the required fidelity. For example, software generated from a carefully simulated functional dataflow model may fail to perform correctly when its functions are distributed over a shared network of processing nodes. Cost considerations may force the selection of platform hardware that limits timing accuracy. Neither aspect of development supports comprehensive validation of certification requirements to meet government safety standards.

We propose a suite of tools that aim to address many of these challenges. Currently under development at Vanderbilt's Institute for Software Integrated Systems (ISIS), these tools use the Embedded Systems Modeling Language (ESMoL), which is a suite of domain-specific modeling languages (DSML) to integrate the disparate aspects of a safety-critical embedded systems design and maintain proper separation of concerns between engineering and software development teams. Many of the concepts and features presented here also exist separately in other tools. We describe a model-based approach to building a unified model-based design and integration tool suite which has the potential to go far beyond the state of the art.

We will provide an overview of the tool vision and describe features of these tools from the point of view of available functionality. Note that two development processes will be discussed – the development of a distributed control system implementation (by an assumed tool user), and our development of the tool suite itself. The initial vision section illustrates how the tools would be used to model and develop a control system. The final sections describe different parts of our tool-development process in decreasing order of maturity.

2 Toolchain Vision and Overview

In this work, we envision a sophisticated, end-to-end toolchain that supports not only construction but also the verification of the engineering artifacts (including software) for high-confidence applications. The development flow provided by the toolchain shall follow a variation of the classical V-model (with software and hardware development on the two branches), with some refinements added at the various stages. Fig. 1 illustrates this development flow.

Consider the general class of control system designs for use in a flight control system. Sensors, actuators, and data networks are designed redundantly to mitigate faults. The underlying platform implements a variant of the time-triggered architecture (TTA) [3], which provides precise timing and reliability guarantees. Safety-critical tasks and messages execute according to strict precomputed schedules to ensure synchronization between replicated components and provide fault mitigation and management. Software implementations of the control

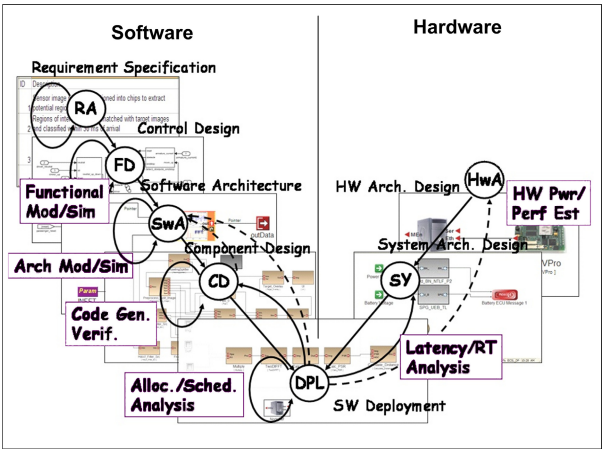

Fig. 1. Conceptual model of the toolchain: Development flow

functions must pass strict certification requirements which impose constraints on the software as well as on the development process.

A modeling language to support this development flow must have several desired properties: (1) the ability to capture the relevant aspects of the system architecture and hardware, (2) ability to "understand" (and import) functional models from existing design tools, (3) support for componentization of functional models, and (4) ability to model the deployment of the software architecture onto the hardware architecture. The ability to import existing models from functional modeling tools is not a deeply justified requirement, it is merely pragmatic. EsMoL provides modeling concepts and capabilities that are highly compatible with AADL [4]. The chief differences are that EsMoL aims for a simpler graphical entry language, a wider range of execution semantics, and most important model-enabled integration to external tools as described below. Model exchange with AADL tools may be desirable in the future. A simple sample design will introduce key points of our model-based development flow and illustrate language concepts.

Our language design was influenced by two factors: (1) the MoC implemented by the platform and (2) the need for integration with legacy modeling and embedded systems tools. We have chosen Simulink/Stateflow as the supported "legacy" tool. As our chosen MoC relies on periodically scheduled time-triggered components, it was natural to use this concept as the basis for our modeling language and interpret the imported Simulink blocks as the implementation of these components. To clarify the use of this functionality, we import a Simulink design and select functional subsets which execute in discrete time, and then assign them to software components using a modeling language that has compatible (time-triggered) semantics. Communication links (signals) between Simulink blocks are mapped onto TTA messages passed between the tasks. The resulting language provides a componentized view of Simulink models that are scheduled periodically (with a fixed rate) and communicate using scheduled, time-triggered messages. Extensions to heterogeneous MoC-s is an active area of research.

2.1 Requirements Analysis (RA)

Our running example will model a data network implementing a single sensor/actuator loop with a distributed implementation. The sensors and actuators in the example are doubly-redundant, while the data network is triply-redundant. The common nomenclature for this type of design is TMR (triple modular redundancy). Unlike true safety-critical designs, we will deploy the same functions on all replicas rather than requiring multiple versions as is often done in practice [5]. The sensors and actuators close a single physical feedback loop. Specifying the physical system and particulars of the control functions are beyond the scope of this example as our focus is on modeling.

This example has an informal set of requirements, though our modeling language currently supports the formalization of timing constraints between sensor and actuator tasks. Formal requirements modeling offers great promise, but in ESMoL requirements modeling is still in conceptual stages. A simple sensor/actuator latency modeling example appears in a later section covering preliminary features for the language.

2.2 Functional Design (FD)

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

Fig. 2 shows a simple top-level Simulink design for our feedback loop along with the imported ESMoL model (Fig. 3). The ESMoL model is a structural replica of the original Simulink, only endowed with a richer software design

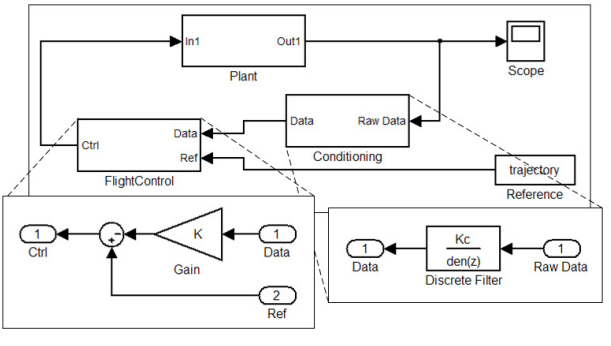

Fig. 2. Simulink design of a basic signal conditioner and controller

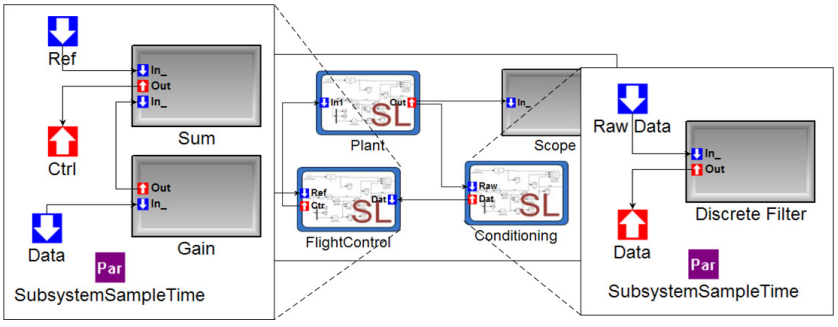

Fig. 3. ESMoL-imported functional models of the Simulink design

environment and tool-provided APIs for navigating and manipulating the model structure in code. A model import utility provides the illustrated function.

2.3 Software Architecture (SwA)

The software architecture model describes the logical interconnection of functional blocks. In the architecture language a component may be implemented by either a Simulink Subsystem or a C function. They are compatible at this level, because here their model elements represent the code that will finally implement the functions. These units are modeled as blocks with ports, where the ports represent parameters passed into and out of C function calls. Semantics for SwA Connections are those of task-local synchronous function invocations as well as message transfers between tasks using time-triggered communication.

Fig. 4 shows the architecture diagram for our TMR model. Instances of the functional blocks from the Simulink model are augmented with C code implementing replicated data voting.

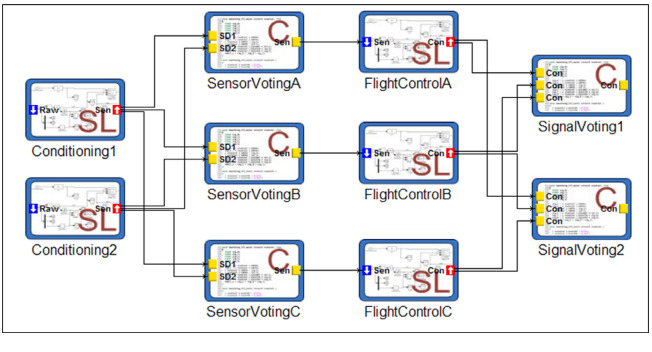

Fig. 4. The architecture diagram defines logical interconnections, and gives finer control over instantiation of functional units

2.4 Hardware Architecture (HwA)

Hardware configurations are explicitly modeled in the platform language. Platforms are defined hierarchically as hardware units with ports for interconnections. Primitive components include processing nodes and communication buses. Behavioral semantics for these networks come from the underlying time-triggered architecture. The platform provides services such as deterministic execution of replicated components and timed message-passing. Model attributes for hardware also capture timing resolution, overhead parameters for data transfers, and task context switching times.

Figs. 5 and 6 show model details for redundant hardware elements. Each controller unit is a private network with two nodes and three independent data buses. Sensor voting and flight control instances are deployed to the controller unit networks.

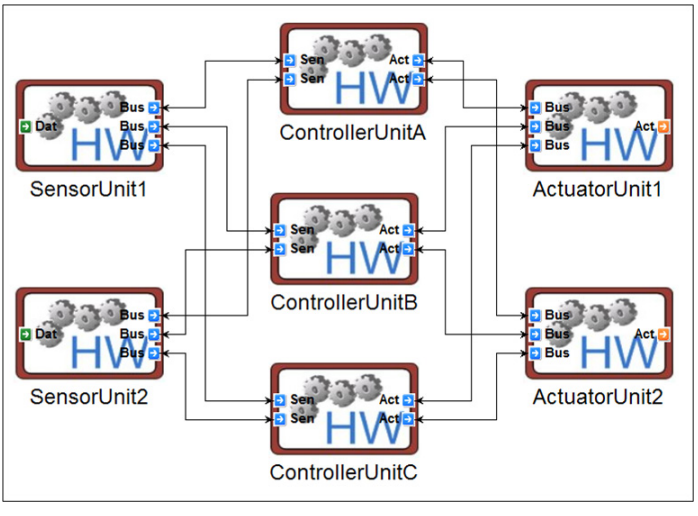

Fig. 5. Overall hardware layout for the TMR example

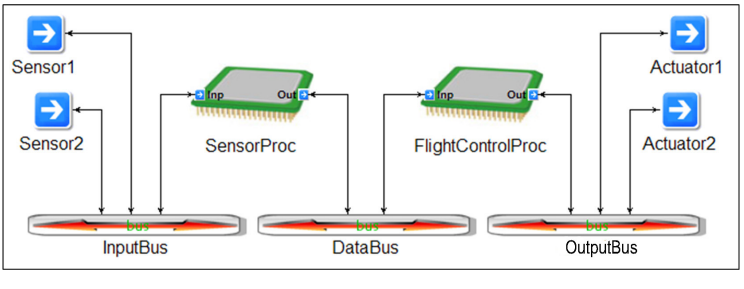

Fig. 6. Detail of hardware model for controller units

2.5 Deployment Models (CD, SY, DPL)

A common graphical language captures the grouping of architecture components into tasks. This language represents three of the design stages from the V-diagram (Fig. 1) – component design (CD), system architecture design (SY), and software deployment (DPL), though we will refer to it as the deployment language. In ESMoL a task executes on a processing node at a single periodic rate. All components within the task execute synchronously. Data sent between tasks takes the form of messages in the model. Whether delivered locally (same processing node) or remotely, all inter-task messages are pre-scheduled for delivery. ESMoL uses logical execution time semantics found in time-triggered languages such as Giotto [6] – message delivery is scheduled after the deadline of the sending task, but before the release of the receiving tasks. In the TT model message receivers assume that required data is already available at task release time. Tasks never block, but execute with whatever data is available each period.

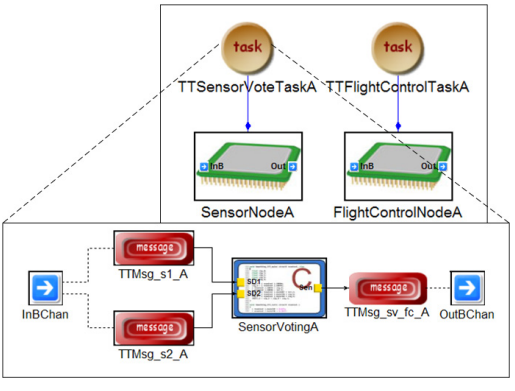

Fig. 7. Deployment model: task assignment to nodes and details of task definition

Deployment concepts – tasks running on processing nodes and messages sent over data buses – are modeled as shown in Fig. 7. Software components and bus channels are actually references to elements defined in architecture and platform models. Model interpreters use deployment models to generate platform-specific task wrapping and communication code as well as analysis artifacts.

3 Existing Tools: Simulink to TTA

Control designs in Simulink are integrated using a graphical modeling language describing software architecture. Components within the architecture are assigned to tasks, which run on nodes in the platform.

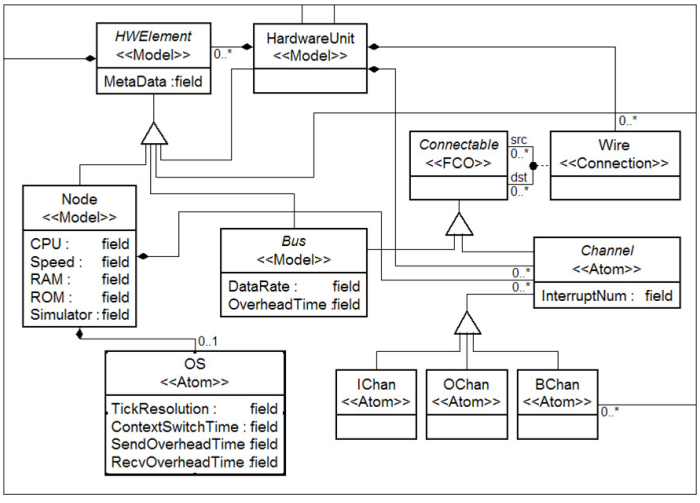

Fig. 8. Platforms. This metamodel describes a simple language for modeling the topology of a time-triggered processing network. A sample platform model is included.

3.1 Integration Details

The Simulink and Stateflow sublanguages of our modeling environment are described elsewhere, though the ESMoL language changes many of the other design concepts from previously developed languages described by Neema [7].

In our toolchain we created a number of code generators. To construct the two main platform-independent code generators (one for Simulink-style models and another one for Stateflow-style models) we have used a higher-level approach based on graph transformations [8]. This approach relies on assumptions that (1) models are typed and attributed graphs with specific structure (governed by the metamodel of the language) and (2) executable code can be produced as an abstract syntax graph (which is then printed directly into source code). This transformation-based approach allows a higher-level representation of the translation process, which lends itself more easily to automatic verification.

The models in the example and the metamodels described below were created using the ISIS Generic Modeling Environment tool (GME) [9]. GME allows language designers to create stereotyped UML-style class diagrams defining metamodels. The metamodels are instantiated into a graphical language, and metamodel class stereotypes and attributes determine how the elements are presented and used by modelers. The GME metamodeling syntax may not be entirely familiar to the reader, but it is well-documented in Karsai et al [10]. Class concepts such as inheritance can be read analogously to UML. Class aggregation represents containment in the modeling environment, though an aggregate element can be flagged as a port object. In the modeling environment a port object will also be visible at the next higher level in the model hierarchy, and

available for connections. The dot between the Connectable class and the Wire class represents a line-style connector in the modeling environment.

High-confidence systems require platforms providing services and guarantees for properties such as fault containment, temporal firewalls, partitioning, etc. System developers should not re-implement such critical services from scratch [2]. Note that the platform also defines a 'Model of Computation' [11]. An MoC governs how the concurrent objects of an application interact (i.e. synchronization and communication), and how these activities unfold in time. The simple platform definition language shown in Fig. 8 contains relationships and attributes describing time-triggered networks.

Similarly, Fig. 9 shows the software architecture language. Connector elements model communication between components. Semantic details of such interactions remain abstract in this logical architecture – platform models must be defined

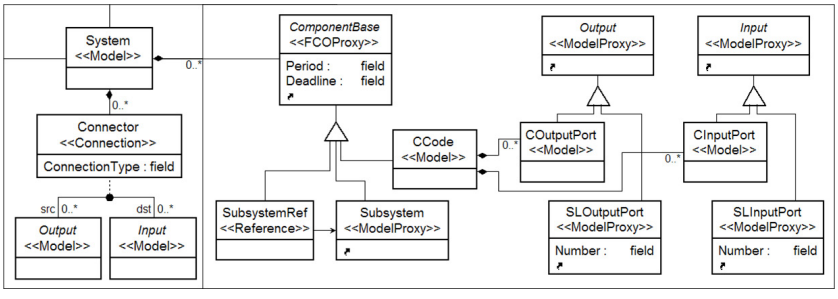

Fig. 9. Architecture Metamodel. Architecture models use Simulink subsystems or C code functions as components, adding attributes for real-time execution. The Input and Output port classes are typed according to the implementation class to which they belong.

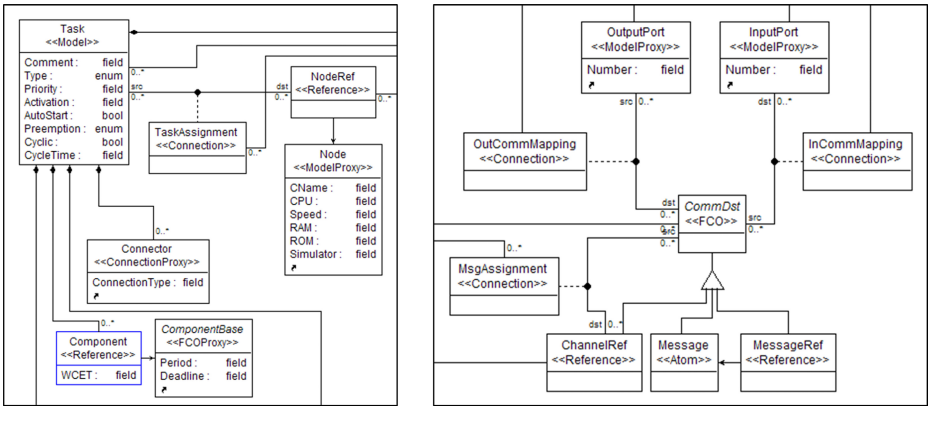

Fig. 10. Details from deployment sublanguage

and associated in order to completely specify interactions (though this version only offers synchronous or time-triggered communications).

Deployment models capture the assignment of Components (and Ports) from the Architecture to Platform Nodes (and Channels). Additional implementation details (e.g. worst-case execution time) are represented here for platform-specific synthesis. Fig. 10 shows the relevant modeling concepts. Simulink objects SLInputPort and SLOutputPort are assigned to Message objects, which represent the marshaling of data to be sent on a Bus.

4 Under Development: Platform-Specific Simulation, Generic Hardware, and Scheduling

A control system designer initially uses simulation to check correctness of the design. Software engineers later take code implementing control functions and deploy it to distributed controllers. Concurrent execution and platform limitations may introduce new behaviors which degrade controller performance and introduce errors. Ideally, the tools could allow the control functions to be re-simulated with appropriate platform effects.

The TrueTime simulation environment [12] provides Simulink blocks modeling processing nodes and communication links. ESMoL tasks map directly to TrueTime tasks. In TrueTime, tasks can execute existing C code or invoke subsystems in Simulink models. Task execution follows configured real-time scheduling models, with communication over a selected medium and protocol. TrueTime models use a Matlab script to associate platform elements with function implementations. A platform-specific re-simulation requires this Matlab mapping function, and in our case also a periodic schedule for distributed time-triggered execution. Both of these can be obtained by synthesis from ESMoL models.

Resimulation precedes synthesis to a time-triggered platform. In order to use generic computing hardware with this modeling environment, we created a simple, open, portable time-triggered virtual machine (VM) to simulate the timed behavior of a TT cluster [13] on generic processing hardware. Since the commercial TT cluster and the open TT VM both implement the same model of computation, synthesis differences amount to management of structural details in the models. The open VM platform is limited to the timing precision of the underlying processor, operating system, and network, but it is useful for testing.

For both steps above the missing link is schedule generation. In commercial TTP platforms, associated software tools perform cluster analysis and schedule generation. For resimulation and deployment to an open platform, an open schedule generation tool is required. To this end we created a schedule generator using the Gecode constraint programming library [14]. The scheduling approach implements and extends the work of Schild and Würtz [15]. Configuration for the schedule generator is also generated by the modeling tools.

4.1 Integration Details

To configure TrueTime or the scheduler, the important details lie in the deployment model. Tasks and Messages must be associated with the proper processing nodes and bus channels in the model. The ISIS UDM libraries [16] provide a portable C++ API for creating model interpreters, navigating in models, and extracting required information. See Fig. 10 for the relevant associations. Model navigation in these interpreters must maintain the relationships between processors and tasks and between buses and messages. Scheduler configuration also requires extraction of all message sender and receiver dependencies in the model.

5 Designs in Progress: Requirements and Model Updates

Many types of requirements apply to real-time embedded control systems design. Embedded systems are heterogeneous, so requirements can include constraints on control performance, computational resources, mechanical design, and reliability, to name a few things. Formal safety standards (e.g. DO-178B [5]) impose constraints on the designs as well as on the development process itself. Accordingly, current research has produced many techniques for formalizing requirements (e.g. ground models in abstract state machines [17] or Z notation [18]). Models could be used to incorporate formal requirements into other aspects of the design process. During analysis, requirements may appear as constraints in synthesized optimization problems or conditions for model checking. Requirements can also be used for test generation and assessment of results.

Management of model updates is also essential. As designs evolve engineers and developers reassess and make modifications. Changes to either the platform model or functional aspects of the design may invalidate architecture and deployment models created earlier. Some portions of the dependent models will survive changes. Other parts needing changes must be identified. Where possible, updates should be automated.

5.1 Integration Details

The requirements sublanguage is in design, and so is light on details. As a simple example of the potential of such a language, Fig. 13 shows a model with latency requirements between tasks, and Fig. 11 shows the modeling language definition. This simple relationship can be quantified and passed directly to the schedule solver as a constraint. Ideally a more sophisticated requirements language could capture the syntax and semantics of an existing formal requirements tool. Some candidate languages and approaches are currently under consideration for inclusion in the framework.

To track model changes we propose to use the Simulink UserData field to store a unique tag in each functional block when the models are imported. During an

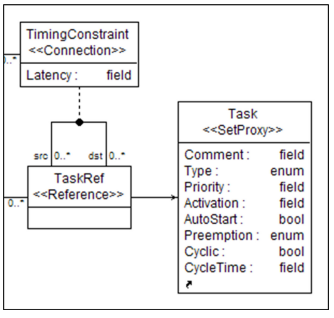

Fig. 11. Latencies are timing constraints between task execution times

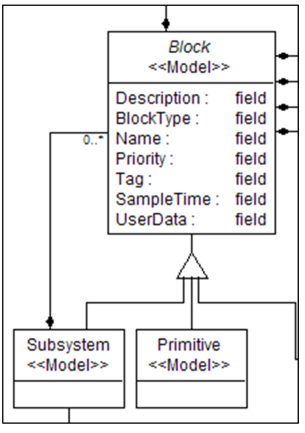

Fig. 12. Simulink's UserData field can help manage model changes occurring outside the design environment

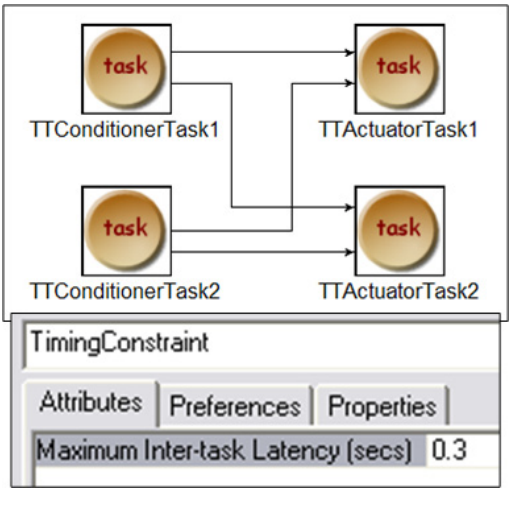

Fig. 13. Example of task latency spec for sample model, with detail of timing attribute value specified on model links

update operation tags in the control design can be compared with previously imported tags in the model environment. Fig. 12 shows the UserData attribute from our Simulink sublanguage, corresponding to the actual attribute in Simulink blocks. To handle issues arising from topology concerns during model evolution, we require control designers to group top-level functionality into subsystems and place a few restrictions on model hierarchy in deployment models.

6 Wishlist: Expanded Semantics, Implementation Generation, and Verification

Many exciting possibilities loom on the horizon for this tool chain construction effort. We briefly describe some concepts currently in discussion for the tools.

The current modeling languages describe systems which provide performance and reliability guarantees by implementing a time-triggered model of computation. This is not adequate for many physical processes and controller platforms. We also need provisions for event-triggered communication and components. Event-triggered component structures give rise to interesting and useful communication patterns common in practical systems (e.g. publish-subscribe, rendezvous, and broadcast). Several research projects have explored heterogeneous timed models of computation. Two notable examples are the Ptolemy project [19] and the DEVS formalism and associated implementations [20]. More general simulation and model-checking tools for timed systems and specifications include UPPAAL [21] and timed abstract state machines [22]. We aim to identify useful design idioms from event-triggered models and extend the semantics of the modeling language to incorporate them. Synthesis to analysis tools is also possible using model APIs.

Safe automation of controller implementation techniques is another focus. Control designs are often created and simulated in continuous time and arbitrary numerical precision, and then discretized in time for platforms with periodic sampling and in value for platforms with limited numeric precision. Recent work in optimization and control offers some techniques for building optimization problems which describe valid controller implementation possibilities [23] [24]. Early work on model interpreters aims to generate such optimization problems directly from the models. Other interesting problems include automated generation of fixed-point scaling for data flow designs. If integrated, tools like BIP [25] provide potential for automated verification of distributed computing properties (safety, liveness, etc...). Model representation of data flow functions, platform precision, and safety requirements could be used together for scaling calculation.

The addition of proper formal requirements modeling can enable synthesis of conditions for model checking and other verification tools. Executable semantics for these modeling languages can also provide the behavioral models to be checked (see Chen [26] [27], Gargantini [28], and Ouimet [29]). Other relevant work includes integration of code-level checking, as in the Java Pathfinder [30] or Saturn [31] tools. Synthesis to these tools must also be verified, an active area of research at ISIS [32].

Acknowledgements

This work is sponsored in part by the National Science Foundation (grant NSF-CCF-0820088) and by the Air Force Office of Scientific Research, USAF (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

1. Henzinger, T., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
2. Sangiovanni-Vincentelli, A.: Defining Platform-based Design. EEDesign of EE-Times (February 2002)
3. Kopetz, H., Bauer, G.: The time-triggered architecture. In: Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software (October 2001)
4. AS-2 Embedded Computing Systems Committee: Architecture analysis and design language (AADL). Technical Report AS5506, Society of Automotive Engineers (November 2004)
5. RTCA, Inc. 1828 L St. NW, Ste. 805, Washington, D.C. 20036: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Prepared by: RTCA SC-167 (December 1992)
6. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001)
7. Neema, S., Karsai, G.: Embedded control systems language for distributed processing (ECSL-DP). Technical Report ISIS-04-505, Institute for Software Integrated Systems, Vanderbilt University (2004)
8. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Journal on Software and System Modeling 5(3), 261–288 (2006)
9. ISIS, V.U.: Generic Modeling Environment, `http://repo.isis.vanderbilt.edu/`
10. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE 91(1) (2003)
11. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A denotational framework for comparing models of computation. Technical Report UCB/ERL M97/11, EECS Department, University of California, Berkeley (1997)
12. Ohlin, M., Henriksson, D., Cervin, A.: TrueTime 1.5 Reference Manual. Dept. of Automatic Control, Lund University, Sweden (January 2007), `http://www.control.lth.se/truetime/`
13. Thibodeaux, R.: The specification and implementation of a model of computation. Master's thesis, Vanderbilt University (May 2008)
14. Schulte, C., Lagerkvist, M., Tack, G.: Gecode: Generic Constraint Development Environment, `http://www.gecode.org/`
15. Schild, K., Würtz, J.: Scheduling of time-triggered real-time systems. Constraints 5(4), 335–357 (2000)
16. Magyari, E., Bakay, A., Lang, A., et al.: Udm: An infrastructure for implementing domain-specific modeling languages. In: The 3rd OOPSLA Workshop on Domain-Specific Modeling (October 2003)
17. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
18. ISO/IEC: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. 13568:2002 (July 2002)

19. UCB: Ptolemy II, http://ptolemy.berkeley.edu/ptolemyII/
20. Hwang, M.H.: DEVS++: C++ Open Source Library of DEVS Formalism (May 2007), http://odevspp.sourceforge.net/
21. Basic Research in Computer Science (Aalborg Univ.) Dept. of Information Technology (Uppsala Univ.): Uppaal. Integrated tool environment for modeling, validation and verification of real-time systems, http://www.uppaal.com/
22. Ouimet, M., Lundqvist, K.: The timed abstract state machine language: An executable specification language for reactive real-time systems. In: Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS 2007), Nancy, France (March 2007)
23. Skaf, J., Boyd, S.: Controller coefficient truncation using lyapunov performance certificate. IEEE Transactions on Automatic Control (in review) (December 2006)
24. Bhave, A., Krogh, B.H.: Performance bounds on state-feedback controllers with network delay. In: IEEE Conference on Decision and Control 2008 (submitted) (December 2008)
25. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM 2006: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, pp. 3–12. IEEE Computer Society Press, Washington (2006)
26. Chen, K., Sztipanovits, J., Abdelwahed, S.: A semantic unit for timed automata based modeling languages. In: Proceedings of RTAS 2006, pp. 347–360 (2006)
27. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
28. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate testsfrom ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003)
29. Ouimet, M., Lundqvist, K.: Automated verification of completeness and consistency of abstract state machine specifications using a sat solver. In: 3rd International Workshop on Model-Based Testing (MBT 2007), Satellite of ETAPS 2007, Braga, Portugal (April 2007)
30. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal 10(2) (April 2003)
31. Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug detection. In: Proceedings of the 17th International Conference on Computer Aided Verification, pp. 139–143 (January 2005)
32. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: Bruni, R., Varró, D. (eds.) 5th International Workshop on Graph Transformation and Visual Modeling Techniques, 2006, Vienna, Austria, pp. 185–194 (April 2006)