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,
WWW home page: 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 resimulation, and a number of other features useful for taming the heterogeneity inherent in embedded control system designs. We describe concepts, elements, and progress 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. 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 these expectations.

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.

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

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. 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. Neither style of development supports comprehensive verification of certification requirements. 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 an 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 in other tools. We hope to describe a model-based approach to building model-based design and integration tools which has the potential to go far beyond the state of the art.

In the sequel we will describe the incremental development of these tools from the point of view of available functionality. After an overview of the project vision and some background, we will discuss the modeling concepts that enable integration of the various tool elements, and conclude with a simple case study to demonstrate the use of the modeling language.

2 Toolchain Vision

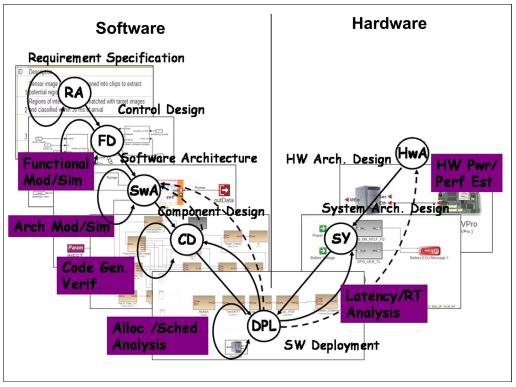

Fig. 1. Conceptual model of the toolchain: Development flow

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.

During development requirements analysis (RA) is followed by functional design (FD) of the controllers, typically enacted using a controller modeling tool like Simulink/Stateflow. Controller models typically include (synchronous) dataflow diagrams and some variant of Statechart models. Next the overall software architecture (SwA) and hardware architecture (HwA) layout follow. We assume that models capturing both SwA and HwA design aspects are available or producible. Controller models are organized during component design (CD) - this involves 'wrapping' code generated from block diagrams into integratable components on some run-time component framework. The platform design consists of (1) system architecture (SY) design (integration of hardware and software aspects), and (2) how software components must be deployed on hardware elements (DPL). Multiple, interlinked models are used in all design stages, and engineers work through the design flow via models to construct the system.

With this model-based design flow we aim to integrate a number of analysis activities. For example, platform models of operating system stacks may be used with code-level verification to check that the generated code will work with a bounded stack. Components allocated to hardware and software resources (e.g. 'tasks') in a system model could be used for schedulability analysis (assuming the problem is decidable and data for worst-case execution time is available).

2.1 Platform

The toolchain supports platform-based design [2] principles in order to reduce complexity. A platform is a component framework providing some common services used by all applications. Existing commercial real-time operating systems typically provide elementary services (task concurrency, synchronization, and communication, partitioning, etc.). Embedded systems require better software platforms with higher-level services, e.g. time-triggered execution, fault-tolerant communication, replica determinism, and others. The main goal of a component framework is the ability to construct systems from components so that properties of the system could be determined from properties of the components and composition structure. The composability property is tightly interwoven with the provided services of the framework, i.e. they cannot be separated.

Component frameworks for embedded systems are an active area of research, and there are numerous research prototypes as well as industrial implementations. High-confidence systems require platforms that provide services and guarantees for needed properties, e.g. fault containment, temporal firewalls, etc. These critical services (like partitioning) should be provided by the platform and not re-implemented from scratch by system developers.

Note that the platform, as a component framework, also defines a 'Model of Computation' [3]. An MoC governs how the concurrent objects of an application interact (i.e. synchronization and communication), and how these activities unfold in time. The MoC also determines how composition works and how system-level properties could be derived from component-level properties and the composition operation. The time-triggered architecture (TTA) [4] is a fault-tolerant MoC that provides precise timing and reliability guarantees in distributed processing systems. Time-triggered systems execute tasks and send messages according to strict, precomputed schedules. TTA enjoys some existing modeling tools [5] as well as commercial implementations [6]. TTA also provides a simplicity that scales down to very small resource-constrained systems.

2.2 Modeling language

A modeling language to support the development flow described above should have several desired properties: (1) ability to "understand" (and import) functional models from existing design tools, (2) ability to capture the relevant aspects of the system architecture and hardware, (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 capabilities that are highly compatible with AADL [7]. In fact, AADL examples are often used to test modeling language changes and features. The chief differences are that EsMoL aims for a simpler graphical entry language and for a wider range of execution semantics, in addition to the integration projects described below. It is likely that importing AADL models directly into EsMoL would be straightforward to do.

The 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 time-triggered messages. Extensions to heterogeneous MoC-s is an active area of research.

2.3 Code generators and synthesis tools

Model-based development is a high-level activity, i.e. programming with models instead of an algorithmic language. Therefore we need to generate executable

code from the models. With proper infrastructure the generator could be more than just a model-to-code translator; it could be a code 'synthesizer' yielding low-level code from higher-level specifications. Synthesis differs from translation as it may search during the code generation process. For instance, on platforms with fixed-point arithmetic, the synthesizer could determine appropriate scaling factors for each operational step in a dataflow model based on known value ranges for inputs and outputs and the available fixed-point precision.

Generating code from dataflow and Statechart models is a well-defined and solved problem, and there are many actual implementations. Code synthesis from higher-level models, however, is an active area of research with many open questions regarding code efficiency and correctness.

2.4 Verification and analysis

Verification and analysis are an inseparable part of the development process for high-confidence systems. There are many well-known verification techniques and tools; however, we must stress two concerns here: (1) verification tools often operate on models (i.e. abstractions of the system), not on detailed code and (2) verification results are meaningful only if model transformations from design models into analysis models or into code are correct: Properties must be carried over by the transformation with no addition of any artifacts extending behavior of the generated code beyond that of the model.

Verification of model translators and construction of verification-based tools is an active area of research. Correctness of a model transformations is a very complex problem, but some early results indicate promising directions [8]: instead of proving correctness for a transformation in general, one can show that a particular instance of a transformation preserves the properties of interest and conclude that the properties hold for the input of the transformation.

Another promising direction is a connection between models and the generated code. We are working on extending the code generator to carry forward model-level information to the generated code (as annotations) that provide help for the code-level verifier to check model-level properties on the code. This connection could potentially be used to improve performance, as the verifier could reason using higher-level abstractions than those immediately available from the code. Other relevant work includes distributed modeling and verification tools like BIP [9], which strives for separation of concerns using a layered behavior model.

3 Stage 1: From Simulink to TTA

Fig. 2 shows the tool flow for the first stage in the development of our tool chain. 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. Platform-independent and platform-dependent code generators can be created separately, as the language enables the separation of those concerns.

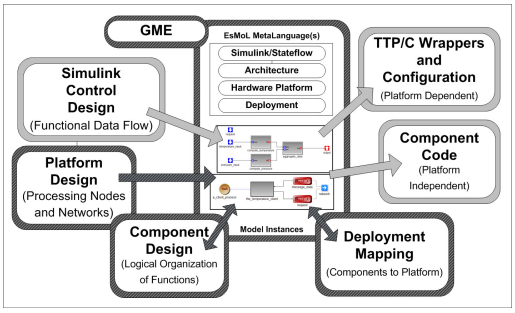

Fig. 2. Stage 1. Model interpreters automatically import Simulink and Stateflow control designs and synthesize code. Users directly enter and edit platform models, design software architecture, and create deployments using the Generic Modeling Environment (GME).

3.1 Integration Details

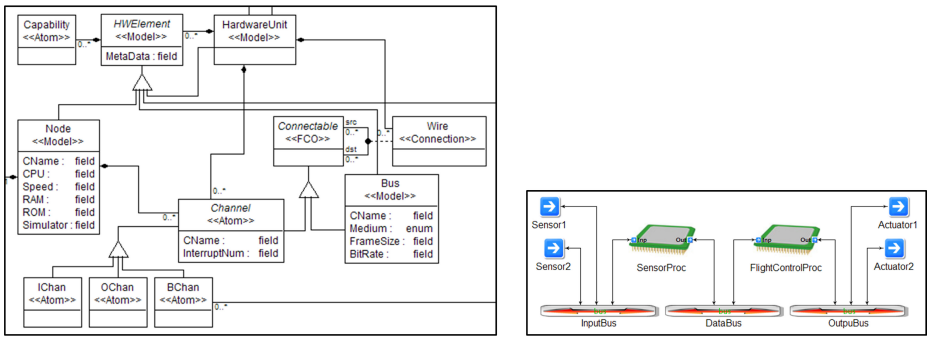

Fig. 3. Platforms. This metamodel describes a simple language for modeling the topology of a time-triggered processing network. A sample platform model is included.

The Simulink and Stateflow sublanguages of our modeling environment are described elsewhere [10], though the EsMoL language changes many of the other design concepts from the older ECSL-DP language. In the present section we will only cover language details relevant to the integration of the control design languages (Simulink and Stateflow) with a time-triggered platform (a TTTech distributed processing cluster [6]).

In our toolchain we created a number of code generators (no synthesizers yet). In the construction of 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 [11]. This approach relies on an assumption 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 graph transformation-based approach allows a higher-level representation of the translation process, which lends itself to algorithmic analysis of the transformations.

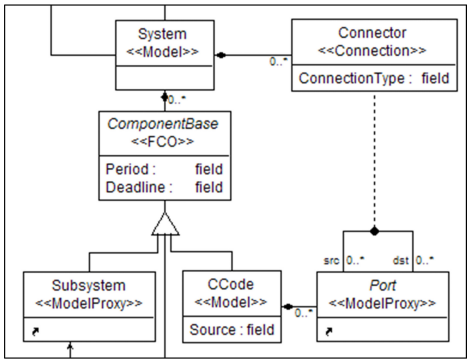

Fig. 4. Architecture Metamodel. Architecture models use Simulink subsystems as components. adding attributes for real-time execution. The Port class is also imported from the Simulink sublanguage to provide ports for components.

The simple platform definition language shown in Fig. 3 contains relationships and attributes for describing a time-triggered network. The GME metamodeling syntax may not be entirely familiar to the reader. The metamodel is interpreted by GME to create a graphical modeling environment representing these concepts. Class concepts such as inheritance can be read analogously to UML – e.g. peer subclasses can be used interchangeably. Class aggregation represents containment in the modeling environment, though an aggregate element can be flagged as a port object. In that case it 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. The figure also shows an example of a hierarchical model from a triple modular redundant (TMR) flight control design. The ports on either end of the model are visible at the next higher level of the hierarchy.

Similarly, Fig. 4 describes the software architecture using classes extracted from the Simulink language. As this metamodel describes a graphical syntax similar to the platform models (i.e. boxes and arrows), we will not cover those details. The Connector element models communication between components. Semantic details of communication interactions remain abstract in this logical architecture – the platform model must be specified and associated in order to completely specify the interactions.

Deployment models capture the assignment of Components (and Ports) from the Architecture to Platform Nodes (and Channels). Additional implementation

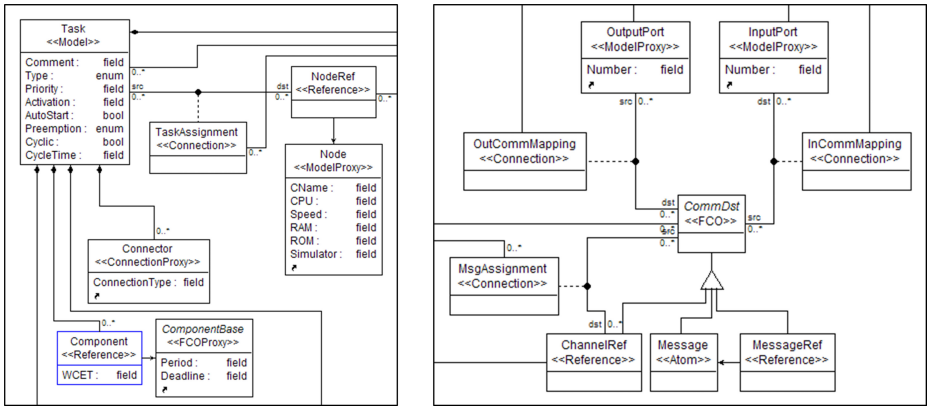

Fig. 5. Details from deployment sublanguage.

details (e.g. worst-case execution time) are represented here for platform-specific synthesis. Fig. 5 shows the relevant modeling concepts. Task execution semantics are periodic, with all Components in a single Task executing at the same rate. Simulink objects InputPort and OutputPort are assigned to Message objects, which marshal data to be sent on a Bus. The current communication semantics are limited to locally synchronous calls (i.e. among components in the same Task) or time-triggered messaging between Tasks (locally on one Node, or across a Bus to other Nodes). In the time-triggered MoC messages are sent only at predetermined times, trading flexibility for determinism. Heterogeneous execution and communication semantics are an active area of research.

3.2 Current and Future Work

Simulink/Stateflow model import and code generation are the most mature features of the toolchain, so the described features are fairly well supported. Ongoing work extends the number of Simulink blocks which can be synthesized by the code generator, and adds the ability to import Matlab functions embedded in the Simulink designs.

4 Stage 2: Platform-specific simulation, generic hardware, and scheduling

Fig. 6 shows the tool flow for the second development stage. 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 should allow the control functions to be re-simulated with appropriate platform effects.

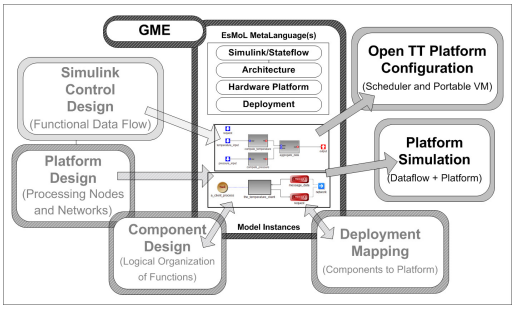

Fig. 6. Stage 2. Designers can synthesize simulations which include platform effects. Wrapper and configuration synthesis for TTP/C can also be easily adapted to generate code and configuration for an open time-triggered (TT) platform and related scheduling tools.

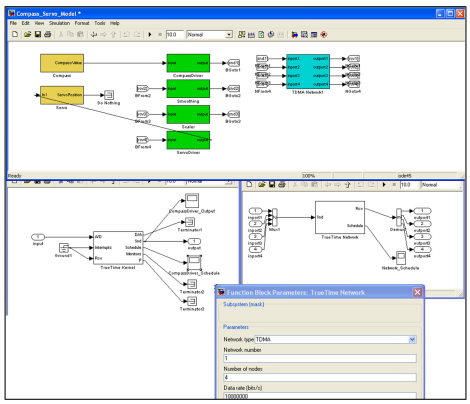

Fig. 7. TrueTime Simulink model with parameter window.

The TrueTime simulation environment [12] provides Simulink model blocks for processing nodes and communication links. 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. Fig. 7 shows an example TrueTime model.

After resimulation follows synthesis to a time-triggered platform. In order to use generic computing hardware with this modeling environment, we created a simple, portable time-triggered virtual machine to simulate the timed behavior of a TT cluster [13]. Since the commercial TT cluster and the open TT virtual machine both implement the same model of computation, synthesis dif-

ferences 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 and for many applications where timing and safety are less critical.

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 simple 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 UDM libraries [16] provide a portable C++ API for creating interpreters to navigate models and extract the relevant information. See Fig. 5 for the relevant associations. Model navigation in these intepreters 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.

4.2 Current and Future Work

This development stage has had working prototypes and demos for each of the described elements. The features are being adapted to recent language changes, integrated more fully, and polished.

5 Stage 3 (in progress): Requirements, external code, and model updates

Fig. 8 depicts features of the third development phase. 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 [17]) 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 [18] or Z notation [19]). Where possible, models should 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 could also be used for test generation and assessment of results.

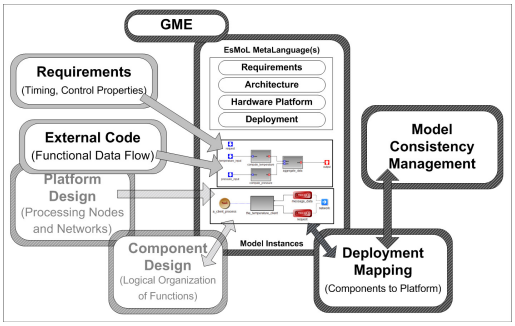

Fig. 8. Stage 3. Requirements modeling, integration of existing code, and automated management of deployment models when functional and platform models change.

Often embedded design projects must include preexisting code, whether from legacy systems or from libraries. External code should also be managed by the tool framework. Code for existing functions passes directly through the code/configuration generators, and is then incorporated with synthesized code consistent with the architecture models. For this particular tool chain the resimulation environment can incorporate existing C/C++ code, so wrappers may be generated for resimulation as well.

The final consideration at this stage is management of model updates. 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. The tool environment should help manage incompatible changes. Some portions of the dependent models will survive. 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. Fig. 9 shows an example with latency requirements between tasks. 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.

Integration of existing code occurs at the software architecture model (see Fig. 4). The CCode class is a peer of the Subsystem class, and Connectors attach to Port objects on the CCode class as well. The topology of the software architecture model then provides the necessary information for the integration of existing code with components implemented as data flows, by generating calls to the existing code at appropriate points.

To track model changes we propose to use the Simulink UserData field to store unique tags when the models are imported. During an update operation

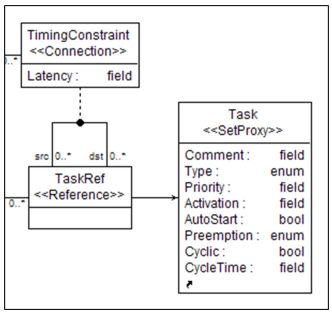

Fig. 9. Latencies are timing constraints between task execution times.

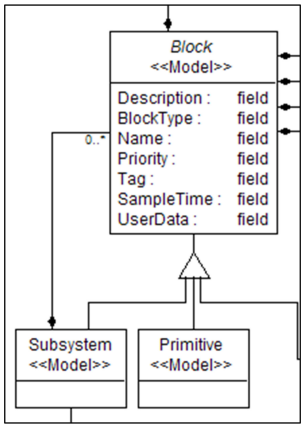

Fig. 10. Simulink's UserData field can help manage model changes occuring outside the design environment.

tags in the control design can be compared with previously imported tags in the model environment. Fig. 10 shows the UserData attribute which was added to the Simulink sublanguage. Successful updating relies on restriction of the hierarchy. Allowing the modeling environment to track changes at any depth in the hierarchy can allow some algorithmically difficult situations. With restricted depth lower level details are abstracted away, leaving only changes that would affect the software architecture or deployment relationships. This imposes some reasonable discipline on control designers, who must organize the top level of their Simulink designs into functional blocks. Another benefit of this structural requirement is added clarity of designs.

5.2 Current and Future Work

All of these language elements and features are in the early stages of design or prototype, though work has been done to flesh out many of the required details.

6 Stage 4 (wishlist): Expanded semantics, implementation generation, and verification

Many exciting possibilities loom on the horizon for this tool chain construction effort. We briefly describe some forward-looking concepts currently in discussion for the tools. The fourth stage of development is depicted in Fig. 11.

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.

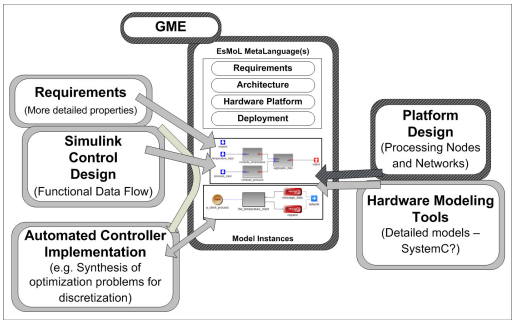

Fig. 11. Stage 4. Extension of formal requirements and platform models and automated synthesis of discretized controllers. These are wishlist items, though some preliminary experiments and designs have been made.

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 [20] and the DEVs formalism and associated implementations [21]. More general simulation and model-checking tools for timed systems and specifications include UPPAAL [22] and timed abstract state machines [23]. We aim to incorporate event-triggered behaviors into the semantics of the modeling language in a safe and realizable way. Synthesis to analysis tools is also possible.

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 [24] [25]. Early model interpreter work aims to generate such optimization problems directly from the models. Other interesting problems include automated generation of fixed-point scaling for data flow designs. 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.

7 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

1. Henzinger, T., Sifakis, J.: The embedded systems design challenge. In: FM: Formal Methods. Lecture Notes in Computer Science 4085. Springer (2006) 1–15
2. Sangiovanni-Vincentelli, A.: Defining Platform-based Design. EEDesign of EE-Times (February 2002)
3. 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)
4. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software (Oct 2001)
5. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE **91** (January 2003) 84–99
6. : TTTech TTP/C Cluster. http://www.tttech.com/
7. Committee, A..E.C.S.: Architecture analysis and design language. Technical Report AS5506, Society of Automotive Engineers (November 2004)
8. A. Narayanan and G. Karsai: Towards verifying model transformations. In R. Bruni and D. Varr, ed.: 5th International Workshop on Graph Transformation and Visual Modeling Techniques, 2006, Vienna, Austria. (Apr 2006) 185–194
9. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, IEEE Computer Society (2006) 3–12
10. 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)
11. Aditya Agrawal and Gabor Karsai and Sandeep Neema and Feng Shi and Attila Vizhanyo: The design of a language for model transformations. Journal on Software and System Modeling **5**(3) (Sep 2006) 261–288
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. : Generic Constraint Development Environment. http://www.gecode.org/
15. Schild, K., Würtz, J.: Scheduling of time-triggered real-time systems. Constraints **5**(4) (Oct. 2000) 335–357
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. RTCA, Inc. 1828 L St. NW, Ste. 805, Washington, D.C. 20036: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. (December 1992) Prepared by: RTCA SC-167.
18. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
19. ISO/IEC: Information Technology Z Formal Specification Notation Syntax, Type System and Semantics. (July 2002) 13568:2002.
20. : UCB Ptolemy II. `http://ptolemy.berkeley.edu/ptolemyII/`
21. Hwang, M.H.: DEVS++: C++ Open Source Library of DEVS Formalism, http://odevspp.sourceforge.net/. first edn. (May 2007)
22. : Uppaal. http://www.uppaal.com/ Integrated tool environment for modeling, validation and verification of real-time systems.
23. 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 '07), Nancy, France (March 2007)
24. Skaf, J., Boyd, S.: Controller coefficient truncation using lyapunov performance certificate. IEEE Transactions on Automatic Control (in review) (December 2006)
25. 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)
26. Chen, K., Sztipanovits, J., Abdelwahed, S.: A semantic unit for timed automata based modeling languages. In: Proceedings of RTAS'06. (2006) 347–360
27. Chen, K., Sztipanovits, J., Abdelwahed, S., Jackson, E.: Semantic anchoring with model transformations. In: Proceedings of European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA). Volume 3748 of Lecture Notes in Computer Science., Nuremberg, Germany, Springer-Verlag (November 2005) 115–129
28. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from asm specifications. In: Abstract State Machines 2003: Advances in Theory and Practice, 10th International Workshop. Volume 2589 of Lecture Notes in Computer Science., Springer (March 2003) 263–277
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 '07), Satellite of ETAPS '07, 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. (January 2005) 139–143