Structuring Simulink Models for Verification and Reuse*

Michael W. Whalen, Anitha Murugesan, Sanjai Rayadurgam, Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
200 Union Street, Minneapolis, Minnesota 55455
{whalen, anitha, rsanjai, heimdhal}@cs.umn.edu

ABSTRACT

Model-based development (MBD) tool suites such as Simulink and Stateflow offer powerful tools for design, development, and analysis of models. These models can be used for several purposes: for code generation, for prototyping, as descriptions of an environment (plant) that will be controlled by software, as oracles for a testing process, and many other aspects of software development. In addition, a goal of model-based development is to develop reusable models that can be easily managed in a version-controlled continuous integration process.

Although significant guidance exists for proper structuring of source code for these purposes, considerably less guidance exists for MBD approaches. In this paper, we discuss structuring issues in constructing models to support use (and reuse) of models for design and verification in critical software development projects. We illustrate our approach using a generic patient-controlled analgesia infusion pump (GPCA), a medical cyber-physical system.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Verification

Keywords

Model based development, Verification, Simulink design verifier

1. INTRODUCTION

Model-Based Development (MBD) refers to the use of domain-specific modeling notations such as Simulink [11] or SCADE [2] that can be analyzed for desired behavior before a digital system is built. The use of such modeling languages allows a system engineer to create a model of the desired system, early in the life cycle, that can be executed on the desktop, analyzed for desired behaviors, and then used to automatically generate code and test cases. The emphasis in MBD is to focus the engineering effort on the early life-cycle activities of modeling, simulation, and analysis, and to automate the late life-cycle activities of coding and testing. This reduces development costs by finding defects early in the life-cycle, avoiding rework that is otherwise necessary when errors are discovered during integration testing, and by automating coding and the creation of test cases. Thus, in domains for which the modeling notation is suitable, MBD can significantly reduce costs while also improving quality.

Recently, MBD has become popular in safety-critical domains such as avionics, automotive, and medical device systems. When used in these domains, support for verification and configuration management becomes an important part of model design. While guideline—notably the MathWorks Automotive Advisory Board guidelines [9]—exist that provide low-level design guidance for construction of control software using MBD tools, there is little, if any, higher-level guidance for the organization of model files and partitioning of models. This is a significant lacuna that we seek to address because the higher-level organization has a profound impact on several aspects of safety-critical software development. In particular, we would like a decomposition and file structure that support:

traceability – controller subsystems must be traceable to specific controller requirements;

configuration management – models should be organized into multiple files that can be easily managed by file-based version control systems;

parallel development – models should be organized into multiple files to support parallel development of systems by different developers or groups; and,

verification – models should be decomposed to best use automation and rigor in verification processes.

For traceability, proper structuring is important because of the different "roles" that models can play within the same notation. It is often the case that model-based development notations are used to describe *plant* models that represent the environment's behavior, *controller* models that represent the software controller's behavior, and *requirement* models that describe constraints on the allowable behavior of the controller given the plant. It is desirable to separate these models into separate files to support verification and reuse at different points in the software development life cycle.

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MiSE '14, June 2 - June 3, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2849-4/14/06 ...$15.00.

A sensible decomposition of models also assists in correct use of the variety of automated verification tools that are available in an MBD approach. The range of verification approaches that are supported under the umbrella of tools for Simulink models include, among others, proof (model-checking) using Simulink Design Verifier [12], automated test generation using Reactis [16], and static analysis using Polyspace [10]. In order to automate aspects of verification it is necessary to specify requirements in a fashion that supports automated verification.

In model-based development, *synchronous observers* [15] have become a standard way of representing requirements formalized as properties. In this representation, each requirement is encoded as a small model that runs in parallel with the controller and determines whether or not the controller is running correctly. However, care must be taken to ensure that design information and controller functionality do not creep into the property models: since the notation is the same, it is easy and even tempting to include internal details of the controller model in the specification of properties. But even small compromises can negatively impact several aspects of system development.

In this paper we suggest a structuring pattern in which environment models, verification models, and controller models are separated and hierarchically organized. We illustrate the pattern using Simulink/Stateflow, but believe that it is generally applicable to a variety of MBD notations including Esterel, SCADE, and UML tools. The properties within the verification model have access only to the input/output interfaces of the functional controller model, thereby avoiding the inclusion of internal controller model information in the properties. This approach:

- suitably shares information between functional controller model and the properties to be verified,
- allows independent and parallel development of functional model and properties,
- supports independent code generation both for the controller model to generate software controllers, and for property models as runtime monitors,
- supports traceability of properties (requirements) to appropriate subsystems within an architecture,
- integrates well with file-based configuration management systems, and
- supports independent reuse of requirements and controller subsystems across multiple projects.

As such, it fills a void in terms of guidance for architectural structuring of MBD models. We illustrate our approach using a medical cyber-physical system.

The rest of the paper is organized as follows. In Section 2, we briefly explain the Simulink and Stateflow notation. In Section 3, we motivate the need of modeling guidelines. Section 4 describes our structuring approach, and Section 5 discusses benefits and drawbacks of the approach. Section 6 describes closely related work and Section 7 concludes.

2. SIMULINK AND STATEFLOW

In this paper, we describe structuring techniques for Simulink and Stateflow. Simulink is a data flow graphical language as well as a tool for modeling and simulating dynamic systems (both the language and the tool are generally referred to as Simulink). Stateflow is a state-based notation similar to David Harel's Statecharts notation [7] (again, Stateflow also refers to the tool). Both Simulink and Stateflow are tightly integrated in the MATLAB environment and can refer to other languages available in the environment. Although both Stateflow and Simulink, in our opinion, have many problems with their semantics (such as the lack of proper type systems, the event semantics in Stateflow, the distinction between transition actions and condition actions, etc.), they are by far the most widely used notations in industry and suit our modeling needs well.

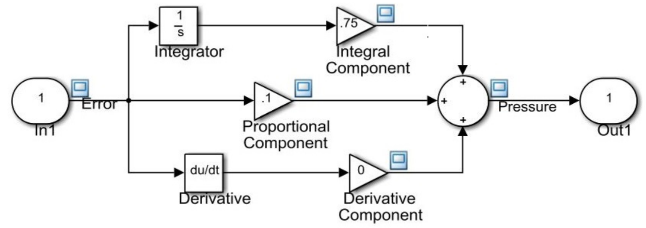

Figure 1: Pump Controller Block Diagram

We illustrate the notations with an example of a Generic Patient Controlled Analgesia (GPCA) infusion system [3]. The GPCA is a medical device used to accurately infuse liquids into a patient's bloodstream. One component of the GPCA is the pump controller, shown in Figure 1. This component implements a PID controller and is implemented as a Simulink block diagram. Simulink block diagrams have a dataflow semantics in which blocks can compute their outputs as soon as their inputs have been computed. Each block in the diagram is either a *basic block* provided by the Simulink library or a *subsystem* that itself is described by another block diagram.

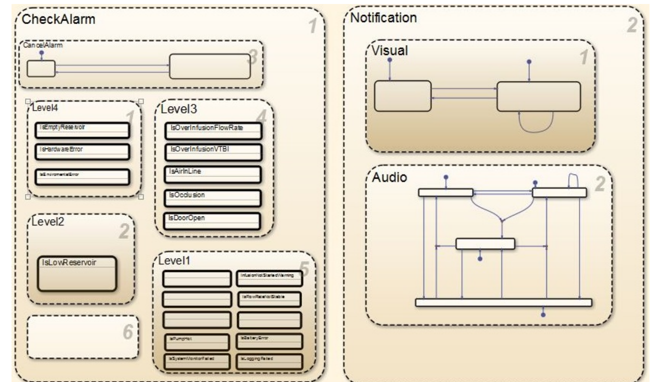

Figure 2: Alarm Component State Machine

An snapshot of the Stateflow model of ALARM, one of the sub-systems of GPCA that is responsible for detecting exceptional conditions and notifying the clinician, is shown in Figure 2. Stateflow is a variation of StateCharts, which augment traditional finite state automata with hierarchical states and parallelism. Stateflow diagrams are useful for naturally describing complex modal behavior.

Simulink and Stateflow are supported by a number of Mathworks and third-party tools that can be used for analysis and verification of models. The Simulink Design Verifier (SLDV) tool [12] is a tool that supports proofs about the behavior of Simulink models, and can provide very high confidence in the correctness of design models.

2.1 Synchronous Observers

The SLDV tool and many other tools can check conformance to requirements specified as *synchronous observers*, which are small Simulink/Stateflow models that execute in parallel with a controller model. Synchronous observers are simply Boolean signals within the model that are expected to be invariantly true.

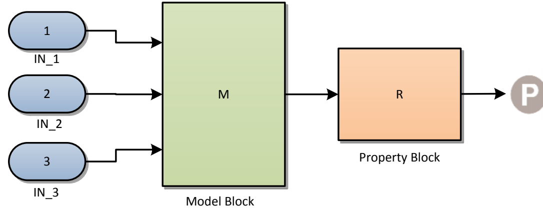

Figure 3: General proof outline

The idea is illustrated in Figure 3. The property to be proved is R for a function or model M. The proof objective block P returns 'true' if the property is currently satisfied by the model or function. For a concrete example, we consider a requirement on the ALARM component:

While ON and infusing, if reservoir volume < empty reservoir threshold then the system shall raise a critical alarm (level 4).

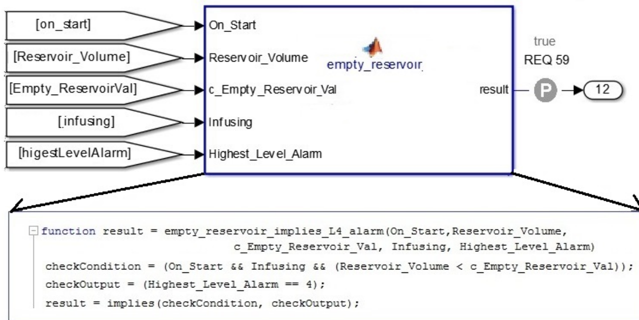

Figure 4: ALARM component property

This property for the ALARM component is depicted in Figure 4. The input signals to this property block are a subset of input signals from the ALARM controller relevant to this property, and the output signal of the ALARM behavioral model. The proof objective P (for property) is used to notify SLDV that this Boolean signal is expected to be invariantly true. In this case, the verification condition in Figure 4 is expressed using embedded MATLAB. Alternatively, the verification conditions could be captured using the Simulink or Stateflow notations.

3. ENGINEERING CONCERNS IN MDB

In model-based development, models play a central role throughout system development. Models are developed to understand the problem space, explore the design space, instantiate a solution and verify its sufficiency. Analysis, simulation, formal verification, code generation, test data generation and test oracle generation are some of the typical uses. There has indeed been a tremendous growth in recent decades in the technologies that underlie model-based development, in particular, for conducting early and up-front analyses and evaluation of crucial design aspects. However, there is a natural lag in the evolution of processes surrounding the use of those technologies for verification. As the technology gains adoption and sees widespread use, best practices learned through experience crystallize into guidelines and standards. Through our experience over the years in model-based development of safety-critical software intensive systems in multiple domains, we have come to value certain characteristics of the organization and decomposition of the models being built, because they decisively impact important engineering concerns. We will first describe a few key concerns that will provide the context for the subsequent modeling guidelines we propose.

Often developers of critical systems must demonstrate to the satisfaction of an independent agency—such as a regulatory body—that every requirement has been verified to hold for the system being assessed. Often the key to a successful demonstration is the ability to relate individual requirements to specific verification artifacts—good *traceability* is required. This traceability needs to be planned, constructed and maintained throughout the development process. Given that requirements inevitably see some changes over time, the verification artifacts necessary to establish traceability must be able to easily accommodate change; e.g., a change to a single requirement should not have an undue impact on the artifacts used model the requirement nor to the functional controller model over which the requirement is verified. Modularity and compositionality are, therefore, necessary principles in the construction of these models.

Another, closely related, engineering concern for system development is the ability to identify and control the evolution of the system throughout its life cycle so that one can reliably answer questions of what, when, why, and how changes were effected. This helps in demonstrating confidence in the development process, which is often mandated for critical systems. Key to this is the ability to identify, control and audit the artifacts at the appropriate level of granularity—*configuration management* is crucial. To achieve this for software artifacts version control systems that operate at the granularity of files in a file system are widely used. Thus, the models, which are software artifacts, must be manageable through decomposition into files at the level of granularity at which changes must be tracked. Therefore, (logical) granularity of the modeling abstractions is a significant consideration since this typically determines the (physical) granularity of modeling artifacts.

As the complexity of the systems being modeled grows and the capability of the analysis tools scale up, these models are more likely to be constructed and maintained by teams of engineers. Thus, division of labor and efficient integration of the resulting work products become important concerns. The granularity of model decomposition has a direct bearing on the division of work, while modularity and compositionality determine the efficiency of integration.

Finally, and most importantly, models, we assert, exist to address important *verification* concerns. The cost of verification, despite great strides in technology, continues to dominate overall development cost for critical systems. While model-based development pushes verification activities for the critical system aspects to early stages of development, thus reducing the cost of detecting and fixing defects, traditional verification techniques like testing are still required to demonstrate confidence in the delivered sys-

tem. However, models, used judiciously, can help automate the verification activities and thereby reducing cost while simultaneously supporting our confidence in the end product. Specifically, models in MBD straddle both the problem space (requirements) and the solution space (architecture and design). Since most models are intended for some form of analysis, in any given model these two aspects intermingle and coexist. Some notations make a distinction in representation (e.g., temporal logic for requirements and finite state machine for design), while others do not (e.g., Simulink requirements property monitors are notationally the same as Simulink design blocks). However, the two aspects take on different roles for different purposes. For example, one may need unit and integration testing of *design* models or test data may be generated from those. On the other hand *requirements* models may be used to synthesize test oracles. For purposes of model-checking, the *design* models represent how the system behaves, while the *requirements* models represent the constraints on what the system must do. These two aspects—when combined in the same model—must be carefully structured so that each can independently evolve, but yet integrate well to serve the purpose at hand. Thus, model structuring should enable *separation of concerns* between these two distinct aspects of the model, irrespective of the support from the underlying notational formalism.

4. MODEL STRUCTURING

It is crucial to structure the functional models and property models carefully taking into consideration their various uses and evolution. Neither The Mathworks' documentation nor related research discusses how to structure the models for verification and maintenance. In order to get maximal use out of our models, we partition them both horizontally and vertically. In this context, horizontal partitioning is determined by the role the model plays in the software development life cycle, and vertical partitioning corresponds to functional decomposition.

4.1 Horizontal Structuring

The idea of horizontal structuring is shown in Figure 5. We explain each of the different basic models (shown at left of Figure 5), and then describe how they can be composed for different verification purposes as shown at right.

Functional Models are those models that describe the behavior of the control software. These are the models that we would like to verify as fit-to-purpose. It is often the case in critical systems that functional models are used to generate software implementations. We must keep functional models separate from descriptions of the environment or requirements in order to generate minimal and efficient code.

Property Models describe representations of requirements (often "shall" statements) created in Simulink and Stateflow. These models emit a set of Boolean values, one for each requirement, which should always have the value true during execution. A false value signals a violation of a requirement.

Environment Models describe the behavior of the system to be controlled or *plant*. When performing simulations of a controller, it is often the case that one creates an environment model and then runs the functional model and environment model in parallel, in which the environment generates inputs to the controller and the controller outputs are fed back into the environment. The environment may be further decomposed into sub-models that describe the physics of the process being controlled as well as models of sensors and actuators that sample and affect the process, respectively.

Test Input Models describe vectors of inputs that constitute an open-loop test for the controller model.

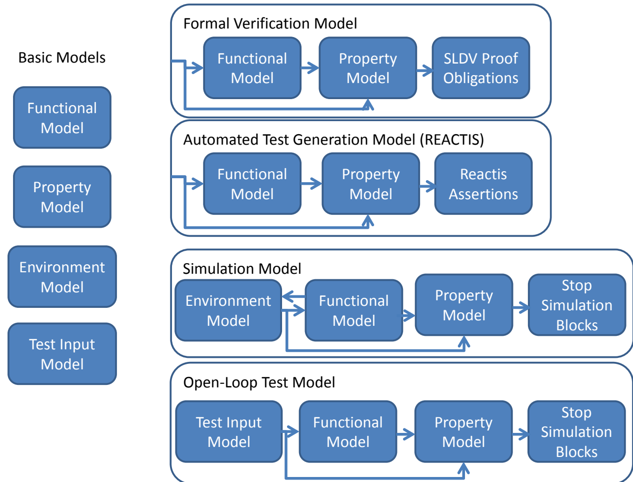

Figure 5: Horizontal Model Structuring

The horizontal decomposition supports parallel development via a separation of concerns; as long as modelers can agree on the input/output interface for the controller, teams can independently build property, test, environment, and controller models that can be straightforwardly integrated. In addition, the artifacts can be used individually and combined for various purposes. Note that the arrows in Figure 5 describe data flows between models. The formal verification model and automated test generation models are "open loop": they have system inputs that drive the analysis. The other models are "closed loop": the system inputs are provided by the environmental model or test input model.

Code generation: Since the functional model is isolated from verification and testing code, an implementation can straightforwardly be generated from it.

Runtime monitor generation: As the property model is written in Simulink/Stateflow, it can be compiled in the same way that the functional model is compiled. By doing so, it is possible to create efficient runtime monitors "for free" that can be used to check for errors in the controller during subsystem and system integration testing over the generated code.

Proof: The MATLAB tool suite now contains a model checker called Design Verifier [12]. By combining the functional model with a property model, it is possible to perform proofs over certain types of models. Although there are scalability limits with Design Verifier, it is often possible to use it to perform unit proofs of subsystems that can be composed using other tools, as was performed in [14].

Simulation: It is straightforward to combine environment, controller, and requirements to perform simulation runs to explore the behavior of the controller.

Automated Test Generation: In situations where the model is too large to perform model checking or contains non-linear arithmetic, automatic test case generation can often provide a rigorous and low-cost way to check control models. There are many tools that perform test-case generation, but in the absence of an *oracle* that can determine correctness, the tests are not meaningful. In our structuring approach, the oracle is provided by the property model.

Automated Test Execution: Even with excellent automation, it will be necessary to write some tests by hand to create specific scenarios that illustrate the software performance and check for specific failures. These tests are usually purpose-built to check one aspect of the system. However, in our experience, it is often the case that they may expose additional defects beyond their scope; by using the property model as a monitor, it is possible to use existing tests to find more bugs.

Reuse: To reuse a software component in a critical environment, it is not enough to just have the code. The coding process is generally a small part of a safety-critical development effort; what requires the majority of the time is creating the requirements, test cases, and other required artifacts and establishing traceability between the different artifacts. The horizontal structuring approach packages together many of these required artifacts in a directory structure to facilitate reuse in a safety-critical environment.

Also, it is often the case that the *requirements* are more stable than the *software*, and even in cases where software is re-implemented, many of the component requirements can be reused. By separating out the formal requirements into their own file, it is often possible to reuse them even if a radically different controller implementation is pursued.

Model Conformance: Simulink and Stateflow are very rich languages, encompassing multiple notions of time (discrete and continuous) and an extremely wide range of basic blocks. Not all of these blocks are acceptable for control software due to efficiency or semantic concerns, but they can significantly simplify environmental descriptions or requirements models. By performing horizontal factoring, it is straightforward to build simple static analyses that ensure appropriate blocksets are used based on the model role.

4.2 Vertical Structuring

Along with separation of concerns, we would also like to have hierarchical structuring of our models to support understanding and tractable analysis. This is accomplished with vertical structuring of models. While in Simulink/Stateflow it is possible to perform vertical structuring within the context of a single model, it is not well-suited to our configuration management, traceability, and verification purposes. Instead, we would like to partition the system into sets of files for each layer of the architecture. By using multiple files for hierarchical decomposition, it is possible to independently version the files and to perform parallel development that is compatible with version control and configuration management systems which are primarily file-based.

This use of multiple files also allows us to perform horizontal structuring at multiple levels of the hierarchy. It is very often the case that requirements, as well as implementations, are hierarchically composed [17]. This decomposition allows us to create property, environment, and test models appropriate for each level of the requirements hierarchy.

File Management.

This profusion of model files, however, also leads to a file-management issue. We have structured our model files such that the directory structure corresponds to the abstraction hierarchy in the model: if a parent Simulink block diagram contains a child subsystem, then the directory containing the parent has a subdirectory with the same name as the child. The child directory contains the files for each of the horizontal development activities that are relevant to it. An example is shown in Figure 6, in which the top-level GPCA_SW system contains a *Functional* model for code generation, a *Functional Simulation* model for simulation, a *Properties* model for describing requirements, and a *Verification* model for performing proofs about the controller. The system also has subdirectories for each of the subsystems of the GPCA software: Alarms, Configuration management, Infusion management, Logging, System monitoring, System Status, and Top-Level Mode.

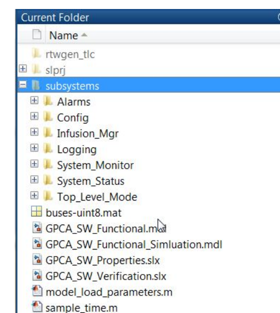

Figure 6: File Organization

Note that this directory structure does not immediately accommodate reuse, because each child "belongs" to exactly one parent. However, it is straightforward in MATLAB to reference models in arbitrary places within the file system, and we plan to define a convention for reusable and application-specific assets.

5. DISCUSSION

Initially, when we performed verification of Simulink controllers, we placed both the controller and properties into the same Simulink model. However, we quickly realized that this was not a good organization. We wanted to be able to generate component implementations, and for the implementation of the component and the specification of requirements for that component to be able to evolve in parallel.

We found that the horizontal and vertical structuring patterns had various merits. First, when the property model is a synchronous observer, it observes the behaviour or the functional model only using its interfaces. However, when we placed the properties within the functional model, we were tempted to use some internal model details such as intermediate variables in the property to simplify its specification. The drawback of using those internal model details is that the errors in computation of those internal variables may not be caught by verification.

Second, since only the interfaces were shared between the functional and property models, internal changes to either of the models did not affect the other. Hence, we were able to develop and modify the models in parallel, except in fairly

rare occasions when an interface needed changing. In the future, our aim is to automatically generate the properties starting from a compositional reasoning framework, as was done manually in [14]. The compositional framework allows us to prove properties of much larger systems than is possible using monolithic approaches.

In addition, for our GPCA model, the property model of the components has been reused for verification of other systems within an product family of infusion pump software with minimal changes, it appears that it is straightforward to reuse all of the models.

6. RELATED WORK

Synchronous observers were proposed as a means for verifying Reactive Systems by Nicolas Halbwachs in a sequence of papers (including [15, 5, 6, 4]) that describe the idea of synchronous observers, the expressive power of them, and uses for formal verification and automated testing. There is little discussion of structuring of models in this work, but the idea of parallel composition and strict use of inputs and outputs is introduced. More recently, [1] has proposed a multi-level refinement/verification approach for Simulink models using contracts; they discuss structuring in the abstract but do not describe file organization or runtime monitoring. The Mathworks Advisory Board has guidance for Simulink/Stateflow models documented in [9] for building software controllers. It discusses which blocksets and design paradigms are appropriate for autogenerating code. In addition, we have written about structuring Simulink models for proper specification of modes [13] and relating controller models to requirements [8], as well as compositional verification of AADL models [17].

7. CONCLUSION

Models play a central role in system engineering for critical domains, where model-based development methods have gained widespread adoption. However, there is not much in terms of guidance and standards that distill the best-practices for modeling, in stark contrast to coding. Considering that substantial software implementations are now likely to be auto-generated, this dissonance calls for urgent attention. Structuring models in an optimal way to serve several purposes can be challenging. In this paper we presented an approach based on horizontal and vertical structuring of models that allowed us to address this effectively for verification models in Simulink/Stateflow. While we illustrated it using an example from the medical device domain, we strongly believe this to be broadly applicable in domains where traceability, configuration management, parallel development and verification are among the prime concerns. Identification, documentation and dissemination of such successful strategies would eventually lead to a repertoire of best-practices and guidelines in the realm of control system modeling, which we believe, are sorely needed.

8. REFERENCES

[1] P. Boström. Contract-based verification of simulink models. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 291–306. Springer Berlin Heidelberg, 2011.

[2] Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.

[3] U. Food and D. Administration. White Paper: Infusion Pump Improvement Initiative. April 2010.

[4] N. Halbwachs. A synchronous language at work: the story of lustre. In *MEMOCODE 2005*, pages 3–11, July 2005.

[5] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language lustre. In *Sixth International Symp. on Lucid and Intensional Programming, ISLIP'93, Quebec*, April 1993.

[6] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: From formal verification to automatic testing. In P. Thiagarajan and R. Yap, editors, *Advances in Computing Science Ů ASIANŠ99*, volume 1742 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1999.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] M. Heimdahl, L. Duan, A. Murugesan, and S. Rayadurgam. Modeling and requirements on the physical side of cyber-physical systems. In *Second International Workshop on the Twin Peaks of Requirements and Architecture*, May 2013.

[9] Mathworks advisory board core modeling guidelines. http://www.mathworks.com/automotive/standards/maab.html.

[10] MathWorks Inc. Polyspace. Via the world-wide-web: http://www.mathworks.com/polyspace.

[11] MathWorks Inc. Simulink. http://www.mathworks.com/products/simulink.

[12] MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier.

[13] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Fifth International Workshop on Modeling in Software Engineering*, May 2013.

[14] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl. Compositional verification of a medical device system. In *ACM International Conference on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.

[15] P. R. Nicolas Halbwachs, Fabienne Lagnier. Synchronous Observers and the Verification of Reactive Systems. In *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing*, June 1993.

[16] Reactive systems inc. Reactis Product Description. http://www.reactive-systems.com/index.msp.

[17] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam. Your what is my how: Iteration and hierarchy in system design. *Software, IEEE*, 30(2):54–60, 2013.