Viewpoints, Formalisms, Languages, and Tools for Cyber-Physical Systems

David Broman[1,2] Edward A. Lee[1] Stavros Tripakis[1] Martin Törngren[3]
{broman,eal,stavros}@eecs.berkeley.edu, martint@kth.se

[1]University of California, Berkeley, USA [2]Linköping University, Sweden [3]KTH, Sweden

ABSTRACT

Cyber-physical systems (CPS) are becoming indispensable in our modern way of life. As an application domain CPS is not new. As an intellectual discipline, however, it is. This paper focuses on CPS modeling, which is an essential activity in CPS design, with multiple challenges. In particular, stakeholders lack a systematic framework and guidelines to help them choose among the many available modeling languages and tools. We propose such a framework in this paper. Our framework consists of three elements: viewpoints, which capture the stakeholders' interests and concerns; concrete languages and tools, among which the stakeholders must make a selection when defining their CPS design environments; and abstract, mathematical formalisms, which are the "semantic glue" linking the two worlds. As part of the framework, we survey various formalisms, languages, and tools and explain how they are related. We also provide examples of viewpoints and discuss how they are related to formalisms.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*; F.1.2 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*

1. INTRODUCTION

Cyber-physical systems (CPS) are becoming an integral part of modern societies [11]. As an application domain CPS is not new. For example, early automotive embedded systems in the 1970s already combined closed-loop control of the brake and engine subsystems (physical parts) with the embedded computer systems (cyber parts). Since then, new requirements, functionalities and networking have dramatically increased the scope, capabilities and complexities of CPS. This has created needs to bridge the gaps between the separate CPS sub-disciplines (computer science, automatic control, mechanical engineering, etc.) and to establish CPS as an intellectual discipline in its own right. The development of a CPS involves many stakeholders who are interested in different aspects of the system. Consider for example the development of an embedded control system such as an *advanced driver assistance system* (ADAS) (e.g., adaptive cruise control). Building such a system naturally involves multiple engineering disciplines, dealing with requirements, control design, software development, hardware development, etc. In an actual organization, persons can take on one or more of these roles. Each stakeholder has a number of questions and decisions that need to be resolved. Modeling plays a key role in such resolution.

Reflecting the heterogeneity of CPS, many modeling languages and tools are typically employed. This leads to multiple models (of different aspects) captured in different modeling languages and tools, and with partly unclear relations among them. For example, to support control design, a control engineer might use continuous time models of the plant and of the controllers, to later refine the control system models to discrete-time models prepared for code generation. Software engineers may use various UML diagrams to design different aspects of the software, parts of which later will have to be integrated with the code generated controllers. System engineers will use yet other models to describe the system interfaces, reliability properties, etc. It is clear that developers would benefit from more systematic ways to make choices about the languages and tools they use. Among other benefits, this should also help avoid, as far as possible, what has been called the "tyranny of tools" [5]. A number of useful surveys of modeling languages and tools exist in the literature, for instance, see [6, 11, 13, 18, 30], or the report of the ARTIST project[1]. Some works also cover specific CPS modeling or design-space exploration challenges and how to address them, e.g. [11, 32]. While these works provide valuable insight into how to deal with specific modeling phenomena, they still leave a gap in how to select a set of modeling languages and tools that meet the needs of stakeholders in CPS design.

In this paper, we propose a framework that attempts to fill this gap. The framework consists of three elements and their relations, as illustrated in Figure 1: viewpoints (Section 2), capturing the stakeholders' interests and concerns; concrete modeling languages and tools (Section 4), which the stakeholders must choose at different stages of the CPS design flow; and abstract, mathematical formalisms (Section 3), which are the "semantic glue" linking the two worlds.

[1]http://www.artist-embedded.org/artist/ARTIST-Survey-of-Programming

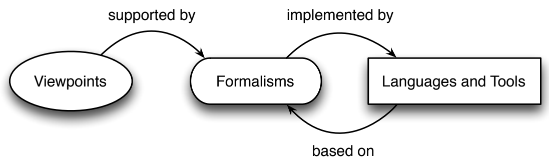

Figure 1: Framework for Viewpoints, Formalisms, Languages and Tools.

Methodologically, we envision a process where stakeholders first identify a given viewpoint or set of viewpoints, then determine one or more formalisms that are most appropriate for these viewpoints, and finally choose one or more concrete languages and tools supporting these formalisms. Our contribution lies in introducing the framework. As part of the framework, we survey various formalisms, languages, and tools and explain how they are related. We also provide examples of viewpoints and discuss how they are related to formalisms.

2. VIEWPOINTS

We adopt the terminology of the ISO/IEEE standard 42010 [22] and apply and adapt it to CPS. We say that each stakeholder has concerns which can be captured (or framed) into viewpoints. For the advanced driver assistance system example mentioned above, the control designers are interested in control system performance and robustness, given constraints imposed by the plant, senors and actuators. A software engineer is another stakeholder. While both control and software stakeholders may have performance as a key concern, the interpretation of performance would be different, for example in terms of 'throughput' for the software engineer vs. 'bandwidth' or 'rise time' for the control engineer. Even when different stakeholders are interested in the same system parts and have same concerns (e.g., a software design engineer and a software tester are likely to both be interested in the software performance), their different roles will determine a slightly different emphasis of their work and how they develop and use related models. We therefore say that a viewpoint is characterized by one or more concerns, parts (interests) and the role of the stakeholder.

To elicit viewpoints, we thus identify stakeholders, their concerns and the parts they are interested in. This concept is depicted in Figure 2, illustrating three example viewpoints identified by a name, the involved concern(s) (such as e.g. robustness or performance), and the system parts/subsystems of interest (note that the parts dimension is not explicitly identified in [22]).

As illustrated in Figure 2, we use the term concern to refer to both functional and non-functional aspects of a system (these can be seen as a requirements dimension) whereas the parts refer to realization components/platforms (at some level of abstraction). In the example of the figure, the control performance viewpoint encompasses the control algorithm functionality and its performance (the concerns) and components corresponding to the controller, sensors, actuators and physical plant (the parts). The software viewpoint, dealing with controller realization, encompasses performance and control algorithm coding concerns as well as software and computing platform parts. The determination of appropriate viewpoints is up to each organization. For

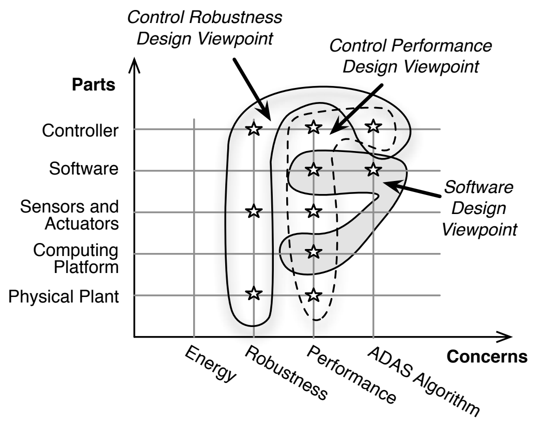

Figure 2: Example of a viewpoints matrix.

example, the control performance and control robustness viewpoints could well be merged into one viewpoint. The more stakeholders, the more complete the set of concerns and parts will be.

According to [22], establishing a viewpoint means defining guidelines and conventions such as recommended types of models, languages, design rules, modeling methods and analysis techniques. The modeling choices will thus be driven by the context of the design task at hand, including the stakeholder concerns. Our framework follows the same spirit, however, a major difference and contribution is that we identify a common ground in terms of formalisms.

3. FORMALISMS

In this section, we review some formalisms which are useful in modeling CPS. Our goal is by no means to be exhaustive, but merely to give examples of formalisms; in particular, those listed in Figure 3. Notable omissions include stochastic formalisms, as well as formalisms used in scheduling and real-time scheduling theory. The links between the viewpoints and formalisms shown in this figure are 'support' relations, loosely interpreted to mean formalisms which are suitable for modeling various aspects of the corresponding viewpoint. For instance, the 'Control Robustness Design' viewpoint is supported by the 'Timed and Hybrid Automata' and 'Differential Equations' formalisms. Again, we do not necessarily mean to be exhaustive in our description of such links. We also note that the formalisms presented below are not necessarily disjoint with each other in terms of expressiveness, e.g., hybrid automata subsume finite state machines or classes of differential equations.

3.1 State Machines

State machines and automata are basic formalisms to describe discrete dynamical systems. State machines and automata come in many variants, therefore forming a class of formalisms rather than a single formalism. Finite-state machines [24] consist of finite sets of inputs, outputs, and states, an output function that describes how outputs are computed, and a transition function that describes how the system changes state. The model can be generalized so that states, inputs, or outputs are modeled by variables

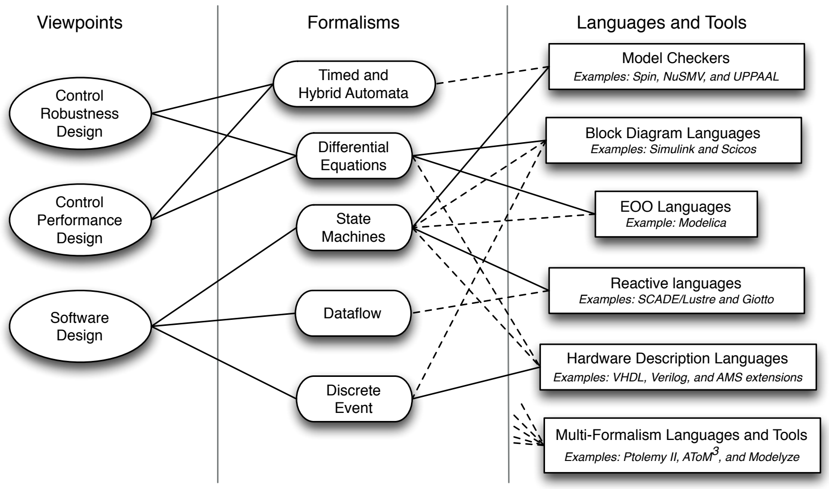
Viewpoints
Formalisms
Languages and Tools
Control Robustness Design
Control Performance Design
Software Design
Timed and Hybrid Automata
Differential Equations
State Machines
Dataflow
Discrete Event
Model Checkers
Examples: Spin, NuSMV, and UPPAAL
Block Diagram Languages
Examples: Simulink and Scicos
EOO Languages
Example: Modelica
Reactive languages
Examples: SCADE/Lustre and Giotto
Hardware Description Languages
Examples: VHDL, Verilog, and AMS extensions
Multi-Formalism Languages and Tools
Examples: Ptolemy II, AToM3, and Modelyze

Figure 3: Relationships between viewpoints, formalisms, languages, and tools. Solid and dashed lines represent strong and weak relationships, respectively.

with infinite domains (e.g., of type integer or real). In that sense, *difference equations* can also be seen as describing state machines. The model can also be generalized to *non-deterministic* machines, where the output and transition functions become relations. Non-determinism is useful when some parts of the system's dynamics are abstracted away (e.g., specific input values).

Finite-state machines are a natural model to describe hardware systems (e.g., digital circuits). State machines (possibly infinite-state) can also be used to describe embedded software systems (e.g., periodic controllers). State machines can even be used to capture systems with continuous dynamics at an abstract level. For example, the finite automaton shown in Figure 4 describes the transitions between three gears in a car. Even though this is a very simple automaton, it still provides some interesting information about the system: in particular, it states that the car cannot "jump" from gear 1 directly to gear 3, but must first pass from gear 2.

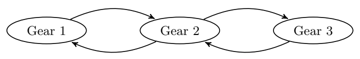

Figure 4: A simple finite-state automaton describing transitions between gears in a car.

Capturing large and complex systems with a single, "monolithic" state machine is not practical. Instead, such systems are typically built by combining a number of simpler components. It is natural to model each component separately, and then compose the components somehow. Two prominent composition paradigms for formalisms such as state machines are synchronous and asynchronous composition. In the *synchronous* composition paradigm all machines execute in "lockstep", that is, they execute simultaneously at the same rate, usually communicating via input-output variables. This is natural when modeling synchronous hardware, where components in a circuit are usually driven by a common clock. In *asynchronous* composition, each machine executes at its own pace. Communication is achieved via shared variables or message passing. This is natural when modeling concurrent processes or threads.

Hierarchical state machines (HSMs), such as Statecharts [17], can be seen as another way of composing state machines. HSMs make it easier, compared to one-level FSMs, to model and organize large state machines, by composing states together in a hierarchical fashion. They are also useful in capturing fault- or exception-handling mechanisms. For instance, the top-level of a HSM could consist of two states representing a "normal" and a "faulty" mode: the former describes what the system should do when everything works as expected, using an internal machine within that mode; the faulty mode describes what the system should do when a fault occurs, using another internal machine.

3.2 Differential Equations

A differential equation is a mathematical equation containing functions and derivatives. Differential equations are in particular used for modeling the physical plant of a CPS. Differential equations can be broadly classified into *ordinary differential equations* (ODEs), *differential-algebraic equations* (DAEs), and *partial differential equations* (PDEs).

An ODE [15] is a differential equation with one independent variable. A first-order ODE has the general form $F(t, x, \dot{x}) = 0$ or explicit form $\dot{x} = f(t, x)$, where $x \in \mathbb{R}^n$ is the unknown state vector (a.k.a. dependent variables) and t the independent variable. In a CPS context, t typically represents time, and the system of equations is often augmented with an input signal $u(t)$. The *order* of a differential equation is the highest derivative of a dependent variable. For example, Newton's second law of motion $F = m\ddot{x}$ is a second-order differential equation, where $\ddot{x}$ is the acceleration. A *solution* to a differential equation is a function $x(t)$ that satisfies the ODE for a given interval. When simulating a physical system, it is desirable to find a *unique solution* by providing *initial conditions*. An ODE together with initial conditions is called an *initial value problem*: $\dot{x} = f(t, x)$,

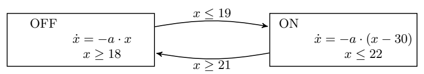

Figure 5: A thermostat as a hybrid automaton.

$x(t_0) = x_0$ where $x_0 \in \mathbb{R}^n$ is the initial conditions. Note that the dimensions of the vectors x_0 and x are equal. In a DAE [25], all dependent variables may not appear differentiated. The general form of a DAE is $F(t, x, \dot{x}, y) = 0$, where t is the independent variable of time, x a vector of variables that appear differentiated, and y a vector of algebraic variables. A PDE is a differential equation that contains multi-variable functions and partial derivatives. PDEs are important for physical modeling, but compared to ODEs are more difficult to solve. There exist various numerical techniques for solving PDEs, such as the finite element method (FEM) and method-of-line (MOL) [7].

3.3 Timed and Hybrid Automata

Discrete state machines can be used in capturing the cyber parts of a CPS, while ODEs and DAEs can be used for the physical parts. Timed and especially hybrid automata formalisms combine both worlds, therefore allowing to describe a CPS as a whole.

Timed automata (TA) [3] extend finite automata with special continuous variables called *clocks*, which measure time. The main strength of the TA formalism is that it allows to capture *quantitative continuous-time* properties. For example, the property "at most 5 time units elapse between events a and b" can be captured with a TA which resets a clock x to 0 when it receives event a and tests whether $x \leq 5$ when it receives b.

Hybrid automata [2] can be viewed as a generalization of timed automata, where the continuous variables can have more complex dynamics than $\dot{x} = 1$. In particular, they can have dynamics described by ODEs and DAEs. The difference from the normal ODE and DAE formalisms is that in hybrid automata the dynamics can be different in different automaton states (called *locations* or *modes*, in order to be distinguished from the true state of the automaton which includes the continuous variables). The behavior of a hybrid automaton is an alternation of continuous phases and discrete steps: in a continuous phase, time elapses and the continuous variables evolve according to the dynamics of the current location; in a discrete step, the automaton takes a transition to a new location (this is assumed to be instantaneous). As an example, Figure 5 shows a simple hybrid automaton modeling a thermostat (example taken from [28]). The automaton has two discrete states, labeled "OFF" and "ON". The dynamics of a variable x representing temperature are given as a differential equation at each state. Note that the combination of transition guards (e.g., $x \leq 19$) and mode invariants (e.g., $x \geq 18$) enable models with non-determinism, but with constraints on their behavior.

3.4 Dataflow Formalisms

Most CPS are concurrent systems, in the sense that they consist of multiple sub-systems operating at the same time and exchanging information in some way. *Dataflow* is an important paradigm in concurrency, where a number of concurrent agents (*actors*) operate on sequences of data (*streams*). Dataflow formalisms such as *static data flow* (SDF) [27][2] and its variants are natural in modeling signal-processing systems. In SDF, actors communicate by sending to each other streams of tokens. Tokens are stored conceptually in FIFO queues. When an actor has enough tokens at its input queues it can *fire*, thereupon producing tokens at its output queues. An important aspect of SDF models is their analyzability: many interesting properties such as absence of deadlocks (an actor "stuck" with not enough tokens to fire) or queue boundedness (can the model be executed with bounded queues?) are decidable. Moreover, timed extensions of SDF and related formalisms, where actor firings take time, can be used to model, analyze and optimize (via scheduling) properties such as throughput or latency [31], which are critical in signal-processing applications.

Dataflow formalisms are also fundamental in modeling distributed systems. In fact, SDF and its variants can be seen as subclasses of *Kahn process networks* (KPN) [23] a very general and elegant formalism, where concurrent processes are modeled as sequential programs communicating via FIFO queues. The main property of a KPN is that, despite the asynchronous execution of processes (non-deterministic interleaving), the streams of data produced at each queue in the network are uniquely defined, a property called *determinacy*. This is in contrast to many other models of concurrency, including the prevalent paradigm of threads communicating via shared memory.

3.5 Discrete-Event Formalisms

The term "discrete-event system" is used in many domains and communities to mean different things. We use it here to group together a number of formalisms which are based on concurrent actors manipulating streams of timed events, such as DEVS [34], real-time process networks [33], or actor theories [14].[3] The DE paradigm is prevalent in simulation frameworks in different application domains, from queueing systems and networks to circuits.

4. LANGUAGES AND TOOLS

Formalisms are mathematical objects consisting of an abstract syntax and a formal semantics. Languages are concrete implementations of formalisms. A language has a concrete syntax, may deviate slightly from the formalism in the semantics that it implements, or may implement multiple semantics (e.g., changing the type of numerical solver in a simulation tool may change the behavior of a model). Also, a language may implement more than one formalisms. Finally, a language is associated with tools to support it, for example parsers, simulators, model-checkers, and code generators. As an example of the distinction, *timed automata* [3] is a formalism, whereas *Uppaal timed automata* [26] and *Kronos timed automata* [9] are languages. In the rest of this section, we highlight some languages and tools (depicted in Figure 3), which are used for modeling, simulation, and verification of cyber-physical systems. Our categorization and list of example languages are in no way exhaustive; it merely serves the purpose of demonstrating the proposed framework.

[2] We use the term 'static' instead of 'synchronous' not to confuse between SDF and synchronous languages.

[3] In this sense, timed automata can also be seen as a discrete-event formalism. We categorize timed automata separately for historical reasons.

Model checkers are tools used to automatically verify whether a model fulfills a specification. For early model exploration, model-checkers typically also support simulation mechanisms, although their main strength is exhaustive verification. Spin [20] is a model-checker designed for modeling and verification of asynchronous software systems. Models are written in PROMELA (Process Meta Language). Correctness claims are specified as assertions or as Linear Temporal Logic (LTL) formulas. The latter are translated into Büchi automata for the verification task. NuSMV [8] is a model-checker primarily used for synchronous systems such as digital circuits. UPPAAL [26] is a model-checker based on timed automata extended with data variable types and other features. A user can specify and verify invariant, reachability and some liveness properties.

Block diagram languages may be used for modeling both the cyber and physical parts of a CPS. A model is defined as a directed graph, where each node, called a block, has as a set of input and output ports. A widely used block diagram environment is the commercial product Simulink®[4]. Simulink can describe both continuous-time and discrete-time systems using differential equations and discrete time difference equations, respectively. Its semantics are implicitly defined by its simulation engine. An open source alternative to Simulink is Scicos[5], a toolbox to the Scilab environment.

Equation-based object-oriented (EOO) languages are, in contrast to block diagram languages, *acausal* (also called non-causal), meaning that the direction of information flow between model components is not specified a priori. The current state-of-the-art EOO language is Modelica[6], which can be used, in particular, for modeling and simulating the physical parts of a CPS. The continuous-time semantics of the models are specified using differential-algebraic equations (DAEs), which can be further composed and connected into hierarchical model structures. Modelica also supports hybrid models, models combining discrete and continuous-time semantics. Functional hybrid modeling (FHM) [29], a related language category, is also based on acausal modeling, but uses a functional approach to model composition. The main strength of EOO and related languages is—compared to block diagram languages—that the topology of the real physical system and the model coincide, which simplifies model reuse.

Reactive languages are designed for implementing reactive systems, the cyber part of a CPS that continuously reacts to the physical environment. Lustre [16] and its commercial implementation SCADE, are synchronous reactive languages. These languages are equipped with a tool chain for efficient program compilation as well as for verification of program properties. Giotto [19] is a time-triggered reactive language for implementing embedded systems with hard real-time constraints.

Hardware Description Languages' (HDLs) main purpose is to model and specify digital circuits. HDL specifications are executable, either by performing discrete-event simulation or—for a subset of the languages—by automatically synthesize hardware logic. Analog and mixed signal (AMS) extensions, such as VHDL-AMS [21] and Verilog-AMS [1], extend these HDLs with differential equations, giving modeling capabilities similar to acusal modeling in EOO languages. HDLs are primary used for implementing the cyber part of a CPS. They can, for example, by synthesizing to field-programmable gate arrays (FPGAs), improve throughput compared to a pure software implementation.

Multi-formalism languages and tools implement and combine different formalisms and models of computation. Ptolemy II [12] is an open-source and extensible modeling and simulation environment. It follows an actor-based approach and supports heterogeneity by defining an executable abstract semantics (concretely, a Java interface) which actors implement. AToM^3 [10] is a graphical tool that combines multi-formalism modeling and meta-modeling. By using a formalism transformation graph (FTG), models can be automatically converted between different formalisms. Another approach of supporting multi-formalisms is to develop and combine different *domain specific languages* (DSLs). For example, Modelyze [4] is a host language that is especially designed for *embedding* DSLs that support different equation-based formalisms. Commercial tools with extensible toolboxes and integration between products, can also be seen as having multi-formalism capabilities. For example Matlab/Simulink/Stateflow/SimEvents combine several formalisms. A standardized co-simulation interface, such as the functional mock-up interface (FMI)[7], is another way of combining tools based on different formalisms.

5. DISCUSSION

There are several challenges when instantiating and applying this framework. In this section, we discuss three of these challenges and how we partially address them.

The first challenge concerns how categories of viewpoints, formalisms, languages, and tools are related to each other. We consider it essential that an analysis of stakeholders, concerns, and parts is made in order to provide a rationale for selecting formalisms, languages and tools. Such an analysis is highly context-dependent and it is therefore difficult to be exhaustive regarding viewpoints, and consequently difficult to give a general mapping between viewpoints and formalisms. The mapping between formalisms and languages/tools is, on the other hand, somewhat clearer. This mapping is, however, between formalisms and categories of languages and tools. Because languages within the same category can relate to slightly different formalisms, we introduce the notion of weak and strong relations (see Figure 3). In this paper, we have intentionally not formalized the meaning of these relations.

The second challenge deals with formalisms and how they are combined and used together. Existing languages and tools typically combine several formalisms, enabling modeling of complex systems. Precisely formalizing the combination of formalisms is, however, very difficult. The challenges of combining, relating, and refining formalisms are core parts of CPS design and consequently for the proposed framework.

The third challenge is how the framework's categories and relations can be used as guidelines for different stakeholders when selecting languages and tools. In this regard, our current work should be seen as preliminary; we have not yet evaluated how it can be used in practice. We envision, however, that in an extended version of this framework, the

[4]http://www.mathworks.com/products/simulink/
[5]http://www.scicos.org
[6]http://www.modelica.org
[7]http://www.functional-mockup-interface.org

stakeholders start by defining a (context-dependent) viewpoints matrix and, by doing so, elicit a number of useful viewpoints. Then, by using the framework, the stakeholders get a better understanding of the most relevant formalisms, languages, and tools. We contend that the mapping is useful as a guideline, although it does not give an easy cheat sheet for selecting tools; the area of CPS modeling is far too complex for making such oversimplifications.

6. CONCLUSIONS

We propose a framework for assisting CPS designers in the modeling process by relating viewpoints, formalisms, languages, and tools. As part of future work, we intend to create a comprehensive extension of our framework where viewpoints, frameworks, languages and tools, and their relations are covered considerably more, both in terms of depth and width.

7. ACKNOWLEDGMENTS

This work was supported in part by NSF awards #0720882 (CSR-EHS: PRET), #0931843 (CPS: Large: ActionWebs), and #1035672 (CPS: Medium: Ptides), ARL award #W911NF-11-2-0038, AFRL, SRC FCRP MuSyC, and CHESS which receives support from Bosch, National Instruments, Thales, and Toyota. The first author was funded by the Swedish Research Council #623-2011-955 and the last author by the ArtistDesign Network of Excellence.

8. REFERENCES

[1] Accellera Organization. Verilog-AMS Language Reference Manual - Analog & Mixed-Signal Extensions to Verilog HDL Version 2.3.1, 2009.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] D. Broman and J. G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Dept., University of California, Berkeley, June 2012.

[5] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proc. of the IEEE*, 98(4):526 –545, 2010.

[6] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli. Languages and tools for hybrid systems design. *Found. Trends Electron. Des. Autom.*, 1:1–193, January 2006.

[7] F. E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, USA, 1991.

[8] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV '02*, 2002.

[9] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer, 1996.

[10] J. de Lara and H. Vangheluwe. AToM 3: A Tool for Multi-formalism and Meta-modelling. In *Fundamental approaches to software engineering*, volume 2306 of *LNCS*, pages 174–188. Springer-Verlag, 2002.

[11] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proc. of the IEEE*, 100(1):13 – 28, January 2012.

[12] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, Jan. 2003.

[13] J. El-khoury, D. Chen, and M. Törngren. A survey of modeling approaches for embedded computer control systems. Technical Report 2003:36, KTH, 2003.

[14] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *Hybrid Systems: Computation and Control*. ACM, 2011.

[15] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations: Nonstiff problems*. Springer, 1993.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, 1987.

[18] D. Henriksson, O. Redell, J. El-Khoury, M. Törngren, and K. Arzén. Tools for real-time control systems co-design – a survey. ISRN LUTFD2/TFRT-7612-SE, Dept. of Automatic Control, Lund Institute of Technology, 2005.

[19] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[20] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[21] IEEE 1706.1 Working Group. *IEEE Std 1076.1-1999, IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Press, New York, USA, 1999.

[22] ISO/IEC/IEEE 42010:2011. *Systems and software engineering - Architecture description, the latest edition of the original IEEE Std 1471:2000, Recommended Practice for Architectural Description of Software-intensive Systems*. IEEE and ISO, 2011.

[23] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[24] Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.

[25] P. Kunkel and V. Mehrmann. *Differential-Algebraic Equations Analysis and Numerical Solution*. European Mathematical Society, 2006.

[26] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[27] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[28] J. Lygeros. Lecture notes on hybrid systems, 2004.

[29] H. Nilsson, J. Peterson, and P. Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, Jan. 2003.

[30] C.-J. Sjöstedt. *Modeling and Simulation of Physical Systems in a Mechatronic Context*. PhD thesis, KTH School of Industrial Engineering and Management, 2009.

[31] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *Computers, IEEE Transactions on*, 57(10):1331 –1345, Oct. 2008.

[32] N. Trcka, M. Hendriks, T. Basten, M. Geilen, and L. J. Somers. Integrated model-driven design-space exploration for embedded systems. In *ICSAMOS*, pages 339–346, 2011.

[33] R. K. Yates. Networks of real-time processes. In E. Best, editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag, 1993.

[34] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2 edition, 2000.