On the Partitioning of Syntax and Semantics For Hybrid Systems Tools

Jonathan Sprinkle, Aaron D. Ames, Alessandro Pinto, Haiyang Zheng, and S. Shankar Sastry

Abstract— **Interchange formats are notoriously difficult to finish. That is, once one is developed, it is highly nontrivial to prove (or disprove) generality, and difficult at best to gain acceptance from all major players in the application domain. This paper addresses such a problem for hybrid systems, but not from the perspective of a *tool interchange* format, but rather that of *tool availability* in a toolbox. Through the paper we explain why we think this is a good approach for hybrid systems, and we also analyze the domain of hybrid systems to discern the semantic partitions that can be formed to yield a classification of tools based on their semantics. These discoveries give us the foundation upon which to build semantic capabilities, and to guarantee operational interaction between tools based on matched operational semantics.**

I. Introduction

The study of *hybrid systems* is an exciting and complex topic for research. Generally, hybrid systems are systems where the model of computation for the entire system is heterogeneous. When many researchers refer to a hybrid system, they commonly mean a system with switched discrete states, each of which has its own continuous dynamics. This is the kind of hybrid system to which we refer throughout this paper. The complexity of this specific kind of hybrid system results in all kinds of interesting research problems and modeling challenges. Several tools have been developed to address issues of the design of hybrid systems in simulation, controller synthesis, verification, and validation. However, the complexity of each of these aspects of hybrid system design and development is such that no one tool is currently able of providing all capabilities for all classes of hybrid systems.

Hybrid systems are more prevalent now than they were, say 20 years ago, due in large part to the increasing tendency to control continuous processes with discrete processors. Previously, control by continuous hardware devices such as resistors, capacitors and inductors, was common, but the advent of embedded processors which are innately discrete (both in terms of execution and analysis) required the domain of computer engineering—relevant to design and program these embedded processors—to acquaint itself with the domain of hybrid systems, and vice versa. Creating models of computation and tools in the computing world that sufficiently emulate the continuous world is necessary in order to accurately control or simulate these systems; the degree of freedom in computing to produce this result, however, means that the design flexibility for each tool can be such that it is nontrivial to "connect" the tools with one another, even when the purpose of those tools is identical.

This work is supported by the Large NSF ITR Project on "Foundations of Hybrid and Embedded Software Systems," award number: CCR-0225610.

J. Sprinkle, A.D. Ames, A. Pinto, H Zheng, and S.S. Sastry are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720. {sprinkle,adames,apinto,hyzheng,sastry}@EECS.Berkeley.Edu

Because hybrid systems are interesting on multiple levels of the design phase (e.g., simulation and code generation) a logical problem was one of using the same model of a system across multiple tools. Depending on the level of abstraction of the design tool, this task was either easier or more difficult. Generally, the more domain-specific a tool, the more likely that design decisions were made which conflict with another tool. On the other hand, the more domain-independent a tool, the more likely that constructs unavailable in other tools may have been used.

A. HSIF

A previous attempt to join tools together—the Hybrid Systems Interchange Format (HSIF) [1]—was born in the DARPA MoBIES (Model Based Integration of Embedded Systems) program. HSIF was designed by committee—resulting in a large syntax and some enhancements which were tool-specific. HSIF, in fact, was defined in terms of existing tools, regardless of their design quality [2]. HSIF was successful in interchanging some models between several tools (notably Charon [3], CheckMate [4], and HyVisual [5], [6]). Many of these transformations were unidirectional, and, although bidirectionality in a toolchain is not necessarily a requirement, it was highly nontrivial to introduce enhancements to these translators that would allow so-called "round-tripping" during the design phase. Overall, HSIF was not successful in convincing the entire hybrid systems community that HSIF should be a part of the design philosophy of each tool.

B. The Columbus Project

It would be somewhat overstating to claim that HSIF was a failure. However it is true that there were problems with the HSIF philosophy which would have prevented its growth into a fully fledged interchange format. These problems, and the subtlety with which such an interchange format should be designed, are succinctly discussed in [7], and extensively covered in a Columbus Project report that discusses hybrid systems modeling and interchange issues [8].

C. Integration strategies

The previous interest of *tool integration*, as addressed by the HSIF effort, was not without its problems. For instance the design of the solution was immediately constrained by the maturity and design decisions of each tool. One danger of this is that the influence of certain design decisions could make the entire process untenable for all tools. The best example of this is that not all tools allowed hierarchy in the discrete state specification (i.e., state machines to describe the discrete states), and furthermore that the semantics of hierarchical discrete states were not globally accepted. This led to the decision that all HSIF models in the initial version would be restricted to so-called flattened models (without hierarchy). A further danger is that modifications to a particular tool will then influence the tool integration format (after all, a tool integration format is coupled to the format and capabilities of the tools), which is more useful for tools which are in their “steady state” rather than those whose formats, execution policies, and subtleties are emerging.

Examples such as this, as well as a tight coupling with the tool, make the justification of a tool integration strategy quite difficult. We suggest that a better goal is to utilize the *capabilities* provided by numerical analysis tools, as well as hybrid systems tools, to provide simulation, verification, etc. We draw the analogy to that of a toolbox, where different tools are chosen for different kinds of tasks (e.g., a hammer for a nail, a screwdriver for a screw). Further extending this analogy, not all kinds of screws and nails are the same. Some nails require different weight hammers, or rubber mallets; some screws are flathead while others require phillips.

From the hybrid systems perspective, the design flow for each kind of system is itself a hybrid system, where discrete switches between simulation, verification, and controller synthesis phases require a different kind of analysis tool. We are certain that the hammer/nail analogy could be extended throughout this paper, but from here onward we choose to switch from the metaphorical to the material aspects of the problem, and address the philosophy of using tools for their capabilities, and not simply because they exist as a tool.

D. Paper goal, and overview

The goal of this paper is to acquaint those who are familiar with hybrid systems with some of the computer science reasons that a unified model is difficult to achieve across tools, and also to acquaint those with computer sciences experience why the hybrid systems domain is so difficult to model, due to its complex semantics. Our secondary goal is to encourage readers to investigate the tools with which they are familiar—or which they have designed—in the categories that we describe in this paper, in order to determine whether or not they are suitable for integration in our framework, and to encourage them to collaborate to achieve that end.

The organization of the remainder of this paper is as follows. In Section II we give literature and personal perspective on how to partition the syntax and semantics of hybrid systems based on the theory in the domain. In Section III we discuss our proposal for a toolbox-style framework, which takes the advice of the literature in its layout and philosophy, and provides a well-defined interface in which to lay out the integration of various tools to take advantage of their capabilities. In Section IV we look to the future, and present an estimate for the availability of the toolbox for general enhancement and use. Finally, we present our conclusions.

II. Addressing semantics

The semantics of a particular hybrid systems model (where here *model* means an instance in abstraction or reality, such as the canonical water tank problem, or the bouncing ball) varies between experts. Such variances are not always differences in interpretations, but frequently are made to simplify pieces of the domain for various reasons. Simplifying assumptions such as these may actually have cascading effect on the ability of a certain notation or tool to adequately model other aspects of a hybrid system, or to allow certain kinds of assertions to be guaranteed on further reflection. This problem and reasons for limiting such assumptions in particular is addressed in [5], to which we direct the reader for an in-depth discussion with numerous examples and justifications for certain design decisions.

We stipulate that not all design decisions must be the same in order to have broad tool inter-operability. The Metropolis [9] group has spent significant effort to develop a methodology and framework for the specification of formal semantics. Through the use of formal semantics, as well as the judicious use of abstraction, the Metropolis project aims to reduce the complexity of a complete design flow by accommodating multiple models of computation and design constraints. While much of the preliminary work in Metropolis has been applied to electronic-system design (e.g., processors and embedded hardware), many of the solutions and design abstractions in Metropolis are directly applicable to the world of hybrid systems tools because of the commonality of mismatched semantics, multiple models of computation, and different application domains of a tool (e.g., simulation versus verification).

As previously mentioned, some exploration of the computing world is necessary to understand why these issues emerge, and how they may be addressed. In order to give more detail on why some semantics should be logically partitioned between tools, and some semantics should be common across all tools, some brief background is required on different kinds of syntax and semantics, as discussed in the following sections. For more insight into this subtle topic, we refer the reader to Harel and Rumpe [10].

A. Syntax

The syntax of a language describes the allowable expressions made up of syntactic elements that can be made in that language. Expressions are made up of syntactic elements (e.g., in C++, `void` is a syntactic element), and the syntax of the language describes what ordering and nesting of language elements and expressions are valid (e.g., in C++, `void main( void );` is an expression).

By producing a syntax, a language’s notation is able to be formalized, similar to how mathematical notation is formalized. For instance, the statement $f(x) = 7 + x$ is an expression in mathematics, and is consistently recognized as a valid syntax for the definition of a function. Even slight deviations in this syntax, though, make the statement nonsensical, e.g., $f((x = 7x+$, resulting in a statement that is unable to be interpreted in generally accepted mathematics.

B. Semantics

The interpretation of some syntax is what gives it meaning, or *semantics*. Again, [10] does an excellent job in defining some of the subtleties of semantics as they pertain to language issues. For the purposes of this paper, we can identify two similar, yet significantly distinguishable, kinds of semantics that provide some insight into the modeling and execution of hybrid systems.

1) Denotational semantics: The *denotational semantics* define the semantics of the model, independent of the execution platform. Simply put, the denotational semantics tell the model builder what behavior the model *should* have. In general, the denotational semantics are in line with or defined by some model of computation, such as dataflow, control flow, continuous, discrete, etc. For this paper, the denotational semantics are the intuitive interpretation of the hybrid systems domain.

2) Operational semantics: The *operational semantics* define the semantics of how a model is executed, according to some denotational semantics. Simply put, the operational semantics tell the computer how a model *does* execute. As such, the operational semantics is *technically* decoupled from the denotational semantics, to the degree in which the execution platform differs from the model of computation, but *practically* should exactly reflect the denotational semantics. For example, a model of computation may allow scheduling of an event at a certain time in the future, but no mechanism exists for this in the operating system; the job of the operational semantics is to produce this scheduling effect with the given interfaces of the operating system.

C. Partitioning the $\mathscr{H}$-tuple

The mathematical notation of hybrid systems is varied, to say the least. In the common hybrid systems literature, a hybrid system is defined as a tuple (frequently referred to as the $\mathscr{H}$-tuple), and the length of this tuple varies according to the specifics of the definition. For example, a hybrid system has been defined[1] to be the tuple

$$\mathscr{H} = (Q, X, V, Y, Init, f, h, I, E, G, R). \tag{1}$$

where Q is set of discrete variables, X is a set of state variables, V is a set of input variables, Y is a set of output variables, $Init$ is a set of initial conditions, f is a set of ordinary differential equations, h is a set of output functions, I is a set of domains, E is a set of edges, G is a set of guards and finally R is a set of reset maps. We will avoid a deep explanation of each of the components of $\mathscr{H}$ discussed here, since the purpose is to demonstrate how the tuples defining hybrid systems change in composition more than substance. For example, another tuple defining a hybrid system is given by

$$\mathscr{H} = (\mathsf{H}, \mathbf{S}) \tag{2}$$

where H is a small category and $\mathbf{S}$ is a functor from H to the category of dynamical systems [11]. This definition exhibits a significant difference, upon a cursory examination, because much of the definition is hidden using abstraction techniques, i.e., the definition of small categories and functors.

Presented with these two definitions of a hybrid system—which represent opposite ends of the notational spectrum—the reader may suspect that there is no commonality between the different definitions of hybrid systems utilized in the literature. In fact, this is not the case, though there is some truth to the assertion. All definitions of hybrid systems share the same underlying structure; it is largely the formalization of this structure that changes.

The commonality between most definitions of hybrid systems is that they all have a discrete component: usually in the form of a graph; and a continuous component: a collection of dynamical systems indexed by the vertices of the graph, together with a collection of maps between these dynamical systems, indexed by the edges of the graph. Few researchers dispute the abstraction of a hybrid system into these well known mathematical constructs. However, there is no uniform way in which the tuple is modeled. We will focus on the different ways in which the continuous component of a hybrid system is specified, interpreted, and executed, since this is where many of the differences arise.

D. Example: Transition Semantics

Although a graph-like representation of the discrete portion of a hybrid system is common across most (if not all) hybrid systems abstractions, there is a subtle interpretation among simulation and verification tools which is important to distinguish. Namely, the conditions under which the edge $e = (q, q') \in E$ can—or must—be taken are not universal.

Using the notation in (1), $Guard_{(q,q')} = G_e = \{x \in I_q \mid g_e(x) \leq 0\}$, where I_q is the domain of the state q. Abstractly, hybrid systems experts agree that the intersection of the flow of the state of the system with some guard, G_e, prescribes a discrete change in the behavior of the system.

Transition semantics prescribe that some time, t, is the exact time at which a transition takes place. Triggered/as-is semantics enforce $t_{\mathrm{as-is}} = \min\{t \in \mathbb{R} \mid g(x(t)) = 0\}$. Enabled semantics enforce $t_{\mathrm{enable}} \in \{t \in \mathbb{R} \mid g(x(t)) \leq 0 \wedge x(t) \in I_q\}$. Thus, $t_{\mathrm{as-is}}$ is not necessarily equal to t_{enable} (see Fig. 1).

Because triggered/as-is semantics may require a different transition time than enabled semantics, there is the possibility for multiple simultaneously-enabled guards as well as an inherent nondeterminicity as to the time at which the transition occurs. In verification, the use of enabling semantics can have an advantage because verification results are broader

[1]This definition is based on that of a previous version of the yet unpublished work *The Art of Hybrid Systems*, by Lygeros, Tomlin, and Sastry. Since the use of this definition it has been altered to include fewer elements, and we include it here as evidence of the emerging nature of the definition of hybrid systems, as well as to show an extreme example of the often unwieldy notations required to manage the complexity of the domain.

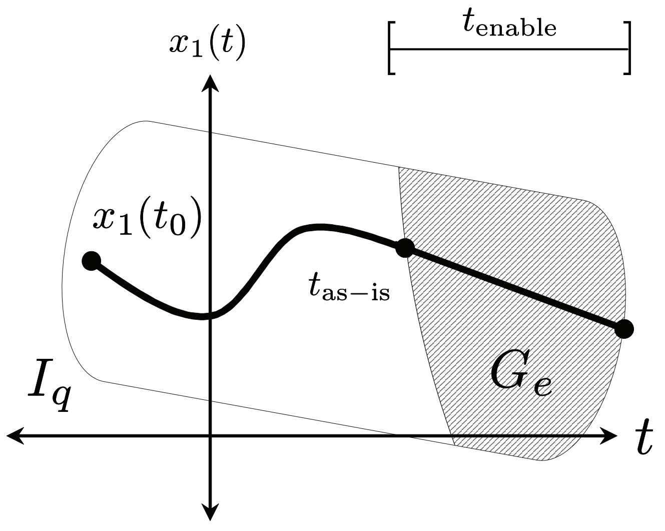

Fig. 1. Flow of the state, $x_1(t)$, and guard, G_e Note that the $t_{\mathrm{as-is}}$ is a single point, and t_{enable} is a range of possible values.

than a particular trace. For simulation, the triggered/as-is semantics are convenient because they provide a way to enforce deterministic execution, i.e., provide a single trajectory of the system.

While the details of these semantics are implemented in the tools themselves, it is possible to distinguish between the tools based on their adherence or indifference to triggered/as-is semantics. This differentiation between tools allows for a partitioning of the *allowed semantics* into different syntax elements, and can give confidence or cast suspicion on the ability of two tools to produce acceptably similar traces.

In the rest of this paper, we show how we have partitioned the $\mathscr{H}$-tuple used to define hybrid systems into logical components that reflect the mathematical concepts (rather than modeling concepts) with which an abstract hybrid system is specified. Our logical partitioning uses hierarchical components to hide information (similar to the definition in (2)), and thus uses structure and containment as an aid to ensure well-formedness of a definition (e.g., it is possible to check that each discrete state has a flow, by looking *in* the discrete state). The choices we made are described next.

III. The *hyper* toolbox framework

The *hyper* toolbox philosophy is to address issues of operational semantics through configuration of syntax elements. This allows a model builder to ensure that the operational semantics of the constructed model will be compatible with the modeler's intuitive denotational semantics. We suggest such functionality through strong typing in the syntax tree.

A. Syntax

The syntax of *hyper* is an evolution of that of HSIF, to the degree that many of the concepts emerge from the mathematical definition of a hybrid system, and its domain concepts. As discussed in Section II-C, we chose to partition the definition of the hybrid system into logical components. For brevity we will show only the vector field's partition.

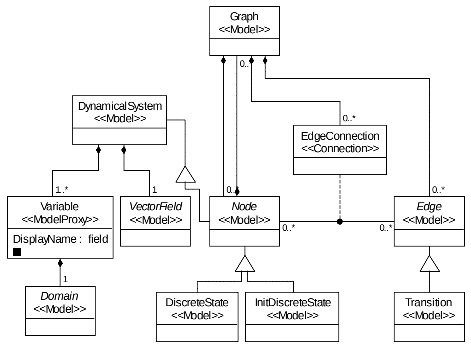

Fig. 2. This model shows a graph which is made up of nodes and edges. Those nodes may be used to model discrete states in the hybrid system, which are indicated using the inheritance triangle, which has an exact semantics (e.g., 'a `DiscreteState` "Is A" `Node`').

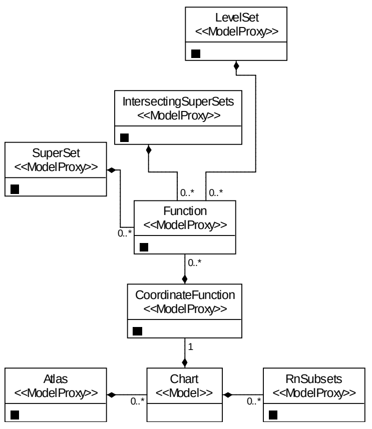

Fig. 3. Complex domains require a functional expression to define their bounds. In these cases, the reader may understand that the *type* of the object (e.g., `LevelSet`, or `SuperSet`) determines how the function is used to form the domain. For example, the same function ${x_1}^2+{x_2}^2-1$ is interpreted as ${x_1}^2+{x_2}^2-1=0$ for a `LevelSet`, and ${x_1}^2+{x_2}^2-1 \geq 0$ for a `SuperSet`.

The syntax of the hyper modeling framework is defined using the MetaGME modeling language [12], [13], which is a metamodeling language used to define domain-specific modeling languages (in our case, the domain of hybrid systems). By using this approach, we can generate the modeling language that *hyper* will use from the definition of what *hyper* considers to be its *own* domain. Fig. 2 shows[2] a simplified version of the graph portion of a hybrid system (discussed in Section II-C); `Nodes` are `DynamicalSystems`, and contain one or more `Variables` (upon which are defined `Domains`), and `VectorFields`.

In the specification of a hybrid system, the vector field defines the flow of the state over time, and some kinds of vector fields are shown in Fig. 4. In the figure, a `VectorField` contains a reference to a defined `StateVariable` (denoting the state variable upon which this `VectorField` defines a flow), and the kind of `VectorField` may be specialized as a `CoordinateFunction` (used to define

[2]A name in *italics* is a UML notation to denote abstract objects, i.e., objects that cannot be instantiated themselves, but whose subtypes can be instantiated.

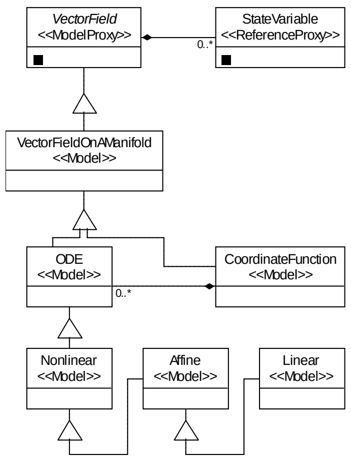

Fig. 4. Kinds of vector fields which may be used to specify a flow. Use of these types allow a correlation between domains and vector fields, to determine whether a mismatch has occurred during specification, and to provide analysis through external tools.

flows on a manifold defined by an atlas), or as an `ODE`, which is subtyped in turn by `NonlinearODE`, in turn `AffineODE`, and in turn `LinearODE`.

B. Impact of syntax on semantics

In the previous subsection we alluded that certain vector fields were useful when defining flows on certain domains. The use of these strong types allows another feature for which the *hyper* framework is intended: the integration of tools with certain *capabilities* to perform analysis—both of the system or controller, and also to confirm that the model of the system or controller conforms to necessary constraints (e.g., that a `VectorField` is a section of the tangent bundle of the manifold defined by an `Atlas`). Mathematical and computational tools, such as Mathematica and Matlab, will prove useful in this area, but the *hyper* framework is designed to tie directly into such tools to utilize their capabilities, without requiring the system modeler to be an expert in the field of computational mathematics. The stronger the types (i.e., the less inference required), the easier it is to export the model to computational tools that include those types as primitives in their language.

1) Choosing tools based on equation types: If the `VectorFields` used to specify a hybrid system are strongly typed as shown in Fig. 4, it is possible to determine the set of tools which can perform simulation, verification, etc., on models which use certain specific subsets of those types. For example, for `Nonlinear`, almost all simulation tools are included in the set of usable tools. However, tools such as d/dt [14] and CheckMate [4] are unable to use pure nonlinear equations when running verification algorithms. When considering the linearity of guard conditions, the set of simulation tools is divided into subsets as well, resulting in those which can use nonlinear guards (among them, Simulink/Stateflow, Modelica, HyVisual, Scicos), and those which can use only linear guards (including CheckMate and HyTech).

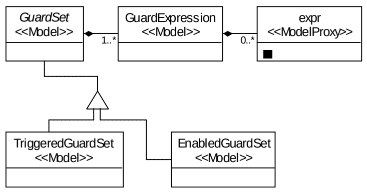

Fig. 5. The types `TriggeredGuardSet` and `EnabledGuardSet` describe whether the contained `GuardExpression` uses triggered/“as-is” semantics, or enabled semantics (see Section II-D).

2) Choosing tools based on transition semantics: In Section II-D we discussed triggered and enabled semantics. If these two types of transitions semantics are strongly typed, as shown in Fig. 5, it is possible to discriminate tools which are compatible with the semantics of these transitions based simply on the type of the objects. While this requires more input time from the perspective of the modeler in an open system, it is more simple to request that the modeler choose the tool first, and let the transition semantics follow from there, as discussed next.

3) Choosing model components based on tools: In addition to determining the tools which can be used based on existing models, it is also possible to determine the model structure based on tool constraints. That is, given that a user wants to use HyVisual for simulation, and CheckMate for verification, it is possible to restrict the set of models which can be created to those which are linear ODEs and linear conjunction of guard expressions. Then, the user is then prevented from creating models that will be unusable in those tools based on constraints generated from tool configuration definitions.

IV. Future work: Adding to *hyper*

The *hyper* framework is designed to extend with the introduction of new elements in the $\mathscr{H}$-tuple. As new elements for defining domains, vector fields, reset maps, and guards are introduced (such as stochastic behaviors or disturbances, invariant sets, hyperplanes), there exists a framework in which to introduce them as types, and to partition them in terms of their operational semantics according to their well-understood mathematical denotational semantics.

Finally, we hope to integrate ongoing work in the definition of a Metropolis metamodel for the interchange of hybrid systems models. This promises to provide an intuitive semantics for how models should be executed, and could produce abstract behaviors of models based on execution semantics chosen in the *hyper* model.

V. Conclusions

This paper details how the $\mathscr{H}$-tuple commonly used as a notation for the definition of a hybrid system may be partitioned in both syntax and semantics to abstract the components of the definition of a hybrid system into a more intuitive model. By strongly typing the elements of the syntax, and introducing concepts which are more commonly used to define modeling languages and definitions of semantics (e.g., operational and denotational semantics), we suggest that the modeling formalism lends itself to an extensible framework that will grow with the continued introduction of new concepts in the hybrid system domain, and as tools emerge which are designed to analyze, verify, or simulate hybrid systems with specific (or more interestingly, without specific) restrictions on their behavior.

We have also presented a metamodel for our language using the GME modeling framework, which is used to generate languages from the formal specification of a domain. This provides a rapid way in which to produce prototypes for the modeling language, as well as interfaces for creating models using the *hyper* modeling language.

At the reading of this paper, we hope the reader will understand better some of the complexities which arise from small differences in the intuitive definitions of a hybrid system, and why these differences can stifle the interaction of two modeling tools that were built upon different assumptions in the domain. Further, we hope that the *hyper* framework will inspire toolmakers to examine their tools and produce a formal operational semantics (coupled with the denotational semantics) which will enable them to integrate their tool into the *hyper* framework, and thus provide modelers familiar with other tools the ability to utilize the *hyper* framework as a toolchain or as an alternative simulator.

VI. Acknowledgments

The authors for this particular paper are a short list of involved researchers who are members of a special interest group that discusses the *hyper* framework weekly at University of California, Berkeley. Active local members of this group, excluding the authors, include (alphabetically by surname) Alessandro Abate, J. Mikael Eklund, Alexander Kurzhanskiy, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. These members contributed greatly to the sentiment of this paper, if not the contents

Additional thanks are due to Gábor Karsai of Vanderbilt University, and Oleg Sokolsky of the University of Pennsylvania, who were the original designers of HSIF, and provided many insightful comments and suggestions over time regarding the *hyper* effort.

We also thank the National Science Foundation and Office of Naval Research for their support and direction of this hybrid systems research.

References

[1] J. Sprinkle, G. Karsai, and A. Lang, *Hybrid Systems Interchange Format (v.4.1.8)*, Vanderbilt University, http://www.isis.vanderbilt.edu/projects/mobies/downloads.asp, Jan. 2004.

[2] J. Sprinkle, "Generative components for hybrid systems tools," *J. of Obj. Tech.*, vol. 4, no. 3, pp. 35–40, Apr. 2005, Special Issue from GPCE Young Researchers Workshop.

[3] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical hybrid modeling of embedded systems," in *EMSOFT'01, First Workshop on Embedded Software*, Oct. 2001.

[4] A. Chutinan and B. H. Krogh, "Computational techniques for hybrid system verification," *IEEE Trans. on Automatic Control*, vol. 48, no. 1, pp. 64–75, 2003.

[5] E. A. Lee and H. Zheng, "Operational semantics of hybrid systems," in *Proceedings of Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds., vol. 3414. Springer-Verlag, Mar. 2005.

[6] A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng, "Hyvisual: A hybrid system visual modeler," University of California, Berkeley, Berkeley, CA 94720, Technical Memorandum UCB/ERL M03/30, July 2003.

[7] A. Pinto, A. Sangiovanni-Vincentelli, L. Carloni, and R. Passerone, "Interchange formats for hybrid systems: Review and proposal," in *Proceedings of Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds., vol. 3414. Springer-Verlag, Mar. 2005.

[8] L. Carloni, M. D. Di Bebedetto, C. Pinello, A. Pinto, and A. Sangiovanni-Vincentelli, "Modeling techniques, programming languages design toolsets for hybrid systems," The Columbus Project, Tech. Rep. DHS4-6, July 2004.

[9] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[10] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, Oct. 2004.

[11] A. D. Ames and S. S. Sastry, "A homology theory for hybrid systems: Hybrid homology," in *Proceedings of Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds., no. 3414. Springer-Verlag, Mar. 2005, pp. 86–102.

[12] G. Karsai, M. Maroti, A. Lédeczi, J. Gray, and J. Sztipanovits, "Composition and cloning in modeling and meta-modeling," *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 263–278, Mar. 2004.

[13] A. Lédeczi, A. Bakay, M. Maroti, P. Vőlgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *IEEE Computer*, pp. 44–51, Nov. 2001.

[14] E. Asarin, T. Dang, O. Maler, and O. Bournez, "Approximate reachability analysis of piecewise linear dynamical systems," in *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, vol. 1790. London: Springer-Verlag, Mar. 2000, pp. 20–31.