9-1-2010

Generating Reliable Code from Hybrid-Systems Models

Madhukar Anand
Cisco Systems

Sebastian Fischmeister
University of Waterloo, sfischme@uwaterloo.ca

Yerang Hur
Posdata America R&D Center

Jesung Kim
The MathWorks

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Recommended Citation

Madhukar Anand, Sebastian Fischmeister, Yerang Hur, Jesung Kim, and Insup Lee, "Generating Reliable Code from Hybrid-Systems Models", *IEEE Transactions on Computers* 59(9), 1281-1294. September 2010. http://dx.doi.org/10.1109/TC.2010.84

Generating Reliable Code from Hybrid-Systems Models

Abstract
Hybrid systems have emerged as an appropriate formalism to model embedded systems as they capture the theme of continuous dynamics with discrete control. Under this paradigm, distributed embedded systems can be modeled as a network of communicating hybrid automata. Several techniques for code generation from these models have also been proposed and commercially implemented. Providing formal guarantees of the generated code with respect to the model, however, has turned out to be a hard problem. While the model is set in continuous time with concurrent execution and instantaneous switching, the code running on an inherently discrete platform, can be affected by the sampling interval, round-off errors, and communication delays between the sensor, controller, and actuators. Consequently, semantic differences between the model and its code can arise with potentially different system behavior. This paper proposes a criterion for faithful implementation of the hybrid-systems model with a focus on its switching semantics. We discuss different techniques to ensure a faithful implementation of the model, and test the feasibility of our concepts by implementing a model heater system. In this heater case study, we successfully eliminate all fault transitions and, thereby, generate code with correct behavior complying with the specification.

Keywords
Formal languages, software engineering

Generating Reliable Code from Hybrid-Systems Models

Madhukar Anand, Sebastian Fischmeister, *Member*, *IEEE*, Yerang Hur, *Member*, *IEEE*, Jesung Kim, *Member*, *IEEE*, and Insup Lee, *Fellow*, *IEEE*

Abstract—Hybrid systems have emerged as an appropriate formalism to model embedded systems as they capture the theme of continuous dynamics with discrete control. Under this paradigm, distributed embedded systems can be modeled as a network of communicating hybrid automata. Several techniques for code generation from these models have also been proposed and commercially implemented. Providing formal guarantees of the generated code with respect to the model, however, has turned out to be a hard problem. While the model is set in continuous time with concurrent execution and instantaneous switching, the code running on an inherently discrete platform, can be affected by the sampling interval, round-off errors, and communication delays between the sensor, controller, and actuators. Consequently, semantic differences between the model and its code can arise with potentially different system behavior. This paper proposes a criterion for faithful implementation of the hybrid-systems model with a focus on its switching semantics. We discuss different techniques to ensure a faithful implementation of the model, and test the feasibility of our concepts by implementing a model heater system. In this heater case study, we successfully eliminate all fault transitions and, thereby, generate code with correct behavior complying with the specification.

Index Terms—Formal languages, software engineering.

1 Introduction

Modern real-time embedded systems are complex, distributed, feature-rich applications. For example, a car incorporates 30 to 60 microcontroller units [1], [2] and desired functionality includes automatic parking, automatic car coordination, and automatic collision avoidance. The development of such functionality is time consuming and difficult, since faults in the temporal or value domain may lead to system failures, which in turn can lead to catastrophes with possibly human losses. Model-based development of real-time embedded systems promises to simplify and accelerate the implementation process. This is because of its promises such as formal guarantees and code generation. Several mathematical models such as Timed Automata [3], Hybrid Systems [4], and State charts [5] have been successfully applied to such systems.

- *M. Anand is with the Cisco Systems, 425 East Tasman Drive, San Jose, CA 95134. E-mail: anandmkr@cisco.com.*
- *S. Fischmeister is with the Department of Electrical and Computer Engineering, University of Waterloo, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada. E-mail: sfischme@uwaterloo.ca.*
- *Y. Hur is with the Posdata America R&D Center, 2350 Mission College Blvd, #703, Santa Clara, CA 95054. E-mail: yehur@posdata-usa.com.*
- *J. Kim is with The MathWorks, 3 Apple Hill Dr., Natick, MA 01760. E-mail: Jesung.Kim@mathworks.com.*
- *I. Lee is with the Department of Computer and Information Science, School of Engineering and Applied Science, University of Pennsylvania, Levine Hall, Room 602, 3330 Walnut Street, Philadelphia, PA 19104-6389. E-mail: lee@cis.upenn.edu.*

Manuscript received 27 May 2009; accepted 3 Nov. 2009; published online 8 Apr. 2010.
Recommended for acceptance by S. Shukla.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-05-0231.
Digital Object Identifier no. 10.1109/TC.2010.84.

1.1 Modeling with Hybrid Systems

Hybrid systems are an appropriate modeling paradigm for embedded control software, because it can be used to specify continuous change of the system state as well as discrete transition of states [6], [7].

Traditionally, control theory and related fields have addressed the problem of designing robust control laws to ensure optimal performance of processes with continuous dynamics. While this is important, control-theory-based approach does not consider the problem of implementing the control laws in software and associated problems involving concurrency and communication. The computer science perspective, on the other hand, is that of a discrete world, largely ignoring the physics of the environment (continuous variables) the embedded system is dealing with and consequently, unable to provide safety and performance guarantees of the system. Hybrid-systems approach combines aspects of these two approaches and is, therefore, better at modeling embedded systems.

Benefits of hybrid-systems modeling are significantly enhanced, if code is generated automatically from the model such that the correspondence between the model and the code is precisely understood. Code generation from hybrid-systems models eventually involves assigning a *rate* by which the continuous state evolves. In such a *discretized* hybrid-systems model, the state changes in a discrete manner according to the rate typically assigned by the model designer. Further, the concurrency of the model is broken in distributed implementations where delays in updates can result in semantic differences. Realizing a faithful implementation of the model, therefore, involves addressing all of these issues.

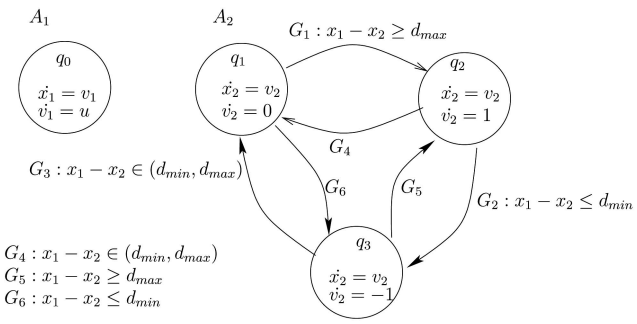


Fig. 1. A system with two agents.

Consider the following vehicle coordination problem (adapted from MoBIES Vehicle-Vehicle Automotive OEP Problem [8]) as an example of a hybrid-systems model:

Example 1 (Vehicle Coordination). Consider the example of vehicle coordination where there are two vehicles. The first vehicle is a leader. It follows the dynamics depicted as agent A_1 in Fig. 1. x_1 denotes the distance of leader from the baseline, v_1, its velocity. The dynamics of the leader are determined by the control function u. The second vehicle trails the leader and maintains a safe distance from it as described in the figure by agent A_2. Its distance from the baseline is given by x_2 and velocity by v_2. If it is closer than d_{min} from the leader, it will slow down with a rate $\dot{v_2} = -1$ and if it is farther than d_{max}, it will accelerate with a rate $\dot{v_2} = 1$. The invariant in state q_1 is $x_1 - x_2 \in [d_{min} - \eta, d_{max} + \eta]$, in q_2 is $x_1 - x_2 \geq d_{min} - \eta$, and in q_3 is $x_1 - x_2 \leq d_{max} + \eta$, where η is the tolerance parameter.

1.2 Problem Statement and Contributions

One problem of code generation for hybrid systems is that the generated code must faithfully implement the hybrid-systems model. This means that transitions in the software state machine should occur according to the hybrid-systems model. Thus, transitions that only occur in the software but not in the model (i.e., faulty transitions) or transitions that only occur in the model but not in the software (i.e., missed transitions) are undesired and should be prevented. This work integrates previous efforts in reliable and faithful code generation [4], [9], [10], [11] by providing a uniform representation with better elucidation of all the introduced concepts, including code snippets, and an example case study.

We introduce a technique called instrumentation of guards to prevent faulty mode transitions but at the same introduce no missed transitions. Instrumentation shrinks the guard so that the transition is made safe (not faulty but not missed). It is related to the hysteresis technique used in many control systems. Like hysteresis, it too uses a higher threshold when switching from below and a lower threshold when switching from above, so that transitions are not made erroneously. However, instrumentation does not prevent oscillations in the controller. It is definitely possible that an instrumented system exhibits oscillations. For such systems, it may become essential to use hysteresis in addition to instrumentation.

This work contributes through the following elements:

- We introduce the problem of reliable code generation with respect to switching discrepancies (Sections 2.1-2.3).
- We present techniques to eliminate the faulty transitions in code (Section 3.1).
- We develop a sufficient condition to check for missed transitions (Section 3.2).
- We illustrate the introduced techniques on an example heater system model (Section 4).

1.3 Introduction to CHARON

This section introduces CHARON [12], a tool for modular specification of interacting hybrid systems based on the notions of agent and mode. For hierarchical description of the system architecture, CHARON provides the operations of instantiation, hiding, and parallel composition on agents, which can be used to build a complex agent from other agents. The discrete and continuous behaviors of an agent are described using modes. For hierarchical description of the behavior of an agent, CHARON supports the operations of instantiation and nesting of modes. Furthermore, features such as weak preemption, history retention, and externally defined Java functions facilitate the description of complex discrete behavior. Continuous behavior can be specified using differential as well as algebraic equations and invariants restricting the flow spaces, all of which can be declared at various levels of the hierarchy. The modular structure of the language is not merely syntactic, but also reflected in the semantics so that it can be exploited during analysis. The key features of CHARON include:

Architectural hierarchy. The building block for describing the system architecture is an agent that communicates with its environment via shared variables and also communication channels. The language supports the operations of composition of agents for concurrency, hiding of variables for information encapsulation, and instantiation of agents for reuse.

Behavioral hierarchy. The building block for describing a flow of control inside an atomic agent is a mode. A mode is basically a hierarchical state machine, that is, a mode can have submodes and transitions connecting them. Variables can be declared locally inside any mode with standard scoping rules for visibility. Modes can be connected to each other through well-defined entry and exit points. The instantiation of modes so that the same mode definition can be reused in multiple contexts. Finally, to support exceptions, the language allows group transitions from default exit points that are applicable to all enclosing modes and to support history retention, the language allows default entry transitions that restore the local state within a mode from the most recent exit.

Discrete and continuous variable updates. Discrete updates are specified by guarded actions labeling transitions connecting the modes. Such updates correspond to mode switching, and are allowed to modify variables through assignment statements. Variables in CHARON can be declared as type analog, and they flow continuously during the continuous updates that model passage of time. The evolution of analog variables can be constrained in

three ways: differential equations (e.g., by equations such as $\dot{x} = f(x, u)$), algebraic equations (e.g., by equations such as $y = g(x, u)$), and invariants (e.g., $x - y < c$) which limit the allowed durations of flows. Such constraints can be declared at different levels of the mode hierarchy.

Example 2. The following code snippet shows how CHARON model of the obstacle avoidance controller from Example 1. The controller has three locations labeled ConstantVel, Accel, and Decel. The two continuous variables are velocity (v) and position (x). The mode TopMode captures the entire model. This mode is composed of the three submodes. The code for one such submode, ConstantVel, is also given below. The rate of change of position and velocity are captured by specifying the differential equation associated with it ($\dot{x} = v$, and $\dot{v} = 0$). The guard G_1 of Fig. 1 is encoded as the condition $Su(x)$.

Listing 1. CHARON code snippet for the vehicle controller of Example 1.

```
 mode TopMode (real x1, real v1){
    write analog x,v;
    mode q1 = ConstantVel();
    mode q2 = Accel();
    mode q3 = Decel();
     trans from default to q1 when true
    do{x = x1; v = v1;}
    trans from q1 to q2 when (Su(x) = true) do {}
    trans from q2 to q1 when (Su(x) = false) do {}
    ...
}
mode ConstantVel()
{
    write analog real x, v;

    diff {d(x) == v; d(v) == 0}
    invSu(x) = true
}
```

1.4 Related Work

Model-based automatic code generation has been an extensive research initiative in recent years and already successfully applied in industry [13]. Commercial modeling tools such as RationalRose [14], TargetLink [15], and SIMULINK [16] also support code generation and address the effect of errors in the code. However, their concerns are largely limited to numerical errors occurring each step during simulation, and the effect of such errors on discrete behavior is not addressed rigorously. Synchronous languages for reactive systems, such as STATECHARTS [5], ESTEREL [17], and LUSTRE [18], [19] also support code generation. However, they do not explicitly support continuous time modeling. SHIFT [20] is a language for hybrid automata that also supports code generation, but it concentrates on dynamic networks.

Model-based development of embedded systems is also promoted by other projects with orthogonal concerns: Ptolemy supports integration of heterogeneous models of computation [21] and GME supports metamodeling for development of domain-specific modeling languages [22]. Girard et al. [23] also consider hybrid-systems modeling of embedded applications; however, their focus is on verification of safety properties and not code generation. There also exist other efforts toward model-driven development of embedded software from models other than hybrid systems [24]. In a closely related work, Stauner [25] discusses at length, the discrete refinement of hybrid automata, considering implementation effects such as sampling errors and its impact on verification.

2 Techniques for Generating Reliable Code

2.1 Model and Overview

Formally, a hybrid model consists of a real vector x denoting the continuous state, a finite set of discrete states P that associates x with a differential equation $\dot{x} = f_p(x)$, for each $p \in P$, and a set of transitions $E \subseteq P \times P$. The continuous state x evolves according to the differential equation $\dot{x} = f_p(x)$ when the current discrete state is p. When the current discrete state is changed from p to p', x is optionally reset to a new value $R(x, p, p')$ defined by a map $R : \mathbb{R}^n \times P \times P \rightarrow \mathbb{R}^n$, and continues evolution in accordance with a new differential equation $\dot{x} = f_{p'}(x)$ associated with p'. To control the discrete behavior, discrete transitions can be guarded by predicates over x. That is, a set $G((p, p')) \subseteq \mathbb{R}^n$ for each $(p, p') \in E$ specifies the necessary condition on the continuous state that the transition (p, p') can be taken. Note that a discrete transition is not necessarily taken immediately even if the guard is true. To enforce a transition, an invariant set $I(p) \subseteq \mathbb{R}^n$ is associated for each $p \in P$ to specify the condition that the discrete state can stay in p (that is, the condition that x will follow $\dot{x} = f_p(x)$). An outgoing transition should be taken before the continuous state goes out of the invariant set.

This framework assumes that there is a network of hybrid automata (called *agents*) communicating via a set of shared variables. A single agent is denoted by $\mathcal{A} = (A, SV)$ where A is the hybrid model of the agent, and SV is the set of shared variables. A system of communicating hybrid agents is represented by the tuple $\mathcal{C} = \langle (A, SV)_1, \ldots, (A, SV)_n \rangle$. Every $s \in SV$ is assumed to be updated by a unique agent, and that it follows rectangular dynamics, i.e., $\dot{s} \in [\mathcal{L}_1, \mathcal{L}_2]$, $\mathcal{L}_1$, $\mathcal{L}_2 \in \mathbb{Q} \setminus \{0\}$. Such rectangular automata are of practical significance, as hybrid systems with very general dynamics can be locally approximated arbitrarily closely using rectangular dynamics [26]. Further, the guard and invariants are assumed to be conjunction of rectangular sets on variables (e.g., $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$). The guards are also assumed to be such that at most one of them is enabled at a time. Implementation of the continuous model involves assigning a suitable sampling rate to every agent.

Definition 1 (Discrete Communicating Hybrid Automata (DCHA)). *Given a system of communicating hybrid agents $\mathcal{C}$, and a relative period of update of variables ρ, $\rho \in \mathbb{Z}^+$, the discretized system of communicating agents (DCHA) is given by $\mathcal{D} = \langle (A, SV, \rho)_1, \ldots, (A, SV, \rho)_n \rangle$, such that $gcd(\rho_1, \ldots, \rho_n) = 1$.*

DCHA is the model of code that is implemented on actual platforms. Therefore, the guarantees of execution are provided with reference to this model. In the definition of DCHA (Definition 1), we have restricted the period of updates of different agents to be relatively prime. This restriction is not necessary, but has been used to keep the model of automaton distinct from its implementation. In fact, we associate this period of update for an agent with an absolute time interval of update in the code that implements the model.

For a rigorous definition of system of communicating agents and their semantics, the reader is referred to [9]. When the discretized model is mapped to a real-time task in the code-generation environment, each agent is assigned a period of execution.

Definition 2 (Code). *The code implementing a DCHA $\mathcal{D}$ is given by the tuple $\mathcal{K} = \langle (A, SV, clk, h, \sigma)_0, \ldots, (A, SVclk, h, \sigma)_n \rangle$, where $clk \in \mathbb{R}^+$ represents the physical time, σ represents the local copy of the shared variables, $h_i (= k\rho_i)$ is the actual period of evaluation, which is a multiple of the relative period ρ_i.*

The definition of platform consists of a mapping between the model and the node that executes the code corresponding to that model, the communication delay involved, and a quantum of execution supported at each node. The quantum is defined by how often a computation can be performed on any node.

Definition 3 (Platform). *A platform $\mathcal{P}$ is defined as the tuple $\langle \mathcal{N}, \mathcal{M}, \phi, \nu \rangle$, where $\mathcal{N}$ is a system of nodes, $\mathcal{M} : \mathcal{A} \rightarrow \mathcal{N}$ is a function that maps an agent to a node on which it is to be executed, ϕ is a map that takes agents as input and returns the upper bound on communication delay between the two agents in $\mathcal{A}$,[1] and $\nu \in \mathbb{R}^+$ is the baseline period, i.e., the quanta of the period of execution of any agent.*

2.2 Code Generation Procedure from Hybrid-System Models

This section gives a brief overview of the procedure of code generation from hybrid models. The translation of continuous behavior specified by differential and algebraic equations is presented first, and then the translation of discrete actions specified by guarded transitions. Later in the section, the issue of discrepancy between the model and the generated code is discussed along with real-time resource concerns and choice of correctness criteria [4], [9].

A differential equation of the form of $\dot{x} = f(x)$ specifies continuous change of variable x at the rate specified as the first derivative $f(x)$ of x with respect to time (i.e., $dx/dt = f(x)$). Continuous change of a variable can be simulated by stepwise update of the variable based on a numerical method that computes an approximate value of the variable after a discrete time step (e.g., Runge-Kutta method [27]). The simplest numerical method is the one known as Euler's method, which projects the value of the variable at the next time step through linear extrapolation. For example, a differential equation $\dot{x} = 2$ is translated into an assignment statement $x := x + 2 \times h$, where h is the step size. In fact, no more sophisticated method is necessary if the right-hand side of the differential equation is a constant.

1. We assume that the communication delay is symmetric between two agents.

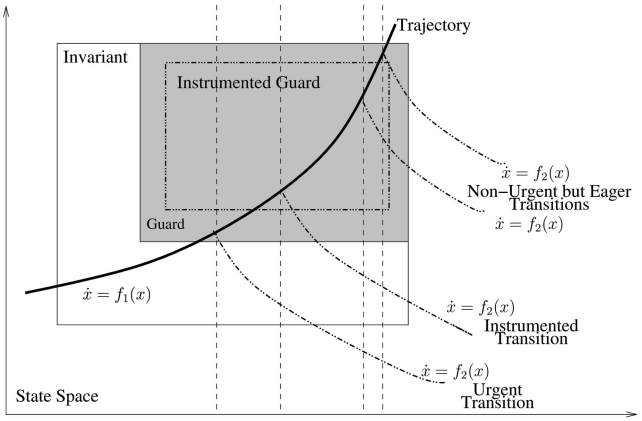


Fig. 2. Different types of transition policies [10].

Once the differential equations are solved, algebraic equations are evaluated to reflect the change due to differential equations. The general form of algebraic equations is $y = g(x)$. An algebraic equation can be implemented by an assignment statement of the same form. That is, an algebraic equation $y = g(x)$ is simply translated into an assignment of the form $y := g(x)$.

Discrete actions of hybrid automata specify instantaneous switching of system dynamics and optional reset of variables. Discrete actions are specified by transitions between positions, where each position defines different dynamics. The transition has a guard that specifies the necessary condition for the transition to be taken, and may have optional assignments to variables that are performed at the moment when the transition is taken. When a transition is taken, differential and algebraic equations defined in the source position become no longer active, and those defined in the destination position take effect immediately.

The guard in the hybrid-system model enables or disables a transition, rather than immediately triggering a transition in hybrid-systems models. This means that enabled transitions may be taken delayed as long as the invariant is satisfied. Conceptually, transitions are nondeterministic in the model, and the implementation determines exactly when a transition is taken. An obvious policy is an *urgent* transition policy where a transition is taken as soon as the guard evaluates true. An *instrumentation* [9] transition policy is one that enforces transitions to be taken some time Δ after the transition is enabled, but no later than Δ before the transition is disabled. Yet another possibility is to enforce a transition once it is enabled. Such a policy is called an *eager* transition policy. In other words, an eager transition policy implies that an enabled transition is always taken, whereas the original hybrid-system model does not mandate this. Note that the urgent transition policy is an eager transition policy. The instrumented transition policy is an eager transition policy if the instrumented guard set is a nonempty set. The different types of transition policies are illustrated in Fig. 2 for a two-dimensional state space. In the example, invariant and guard sets are rectangular sets. The invariant is the outer rectangle, followed by the shaded region which is the guard set. The instrumented guard is the region inside the dashed rectangle (innermost rectangle). The trajectory of the hybrid system is shown in bold ($\dot{x} = f_1(x)$). The various dotted lines from this trajectory represent the different times at which transition to

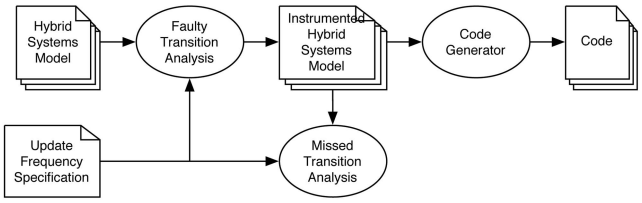


Fig. 3. Workflow of our framework [10].

a different state can be taken. In the next state, the system follows the dotted trajectory representing equation $\dot{x} = f_2(x)$.

This work only considers an eager transition policy.

2.3 Issues in Reliable Code Generation

Typically, the code generator translates a hybrid-systems model into a set of functions that can be invoked periodically by the underlying runtime system to simulate the original model. Intuitively, as the period gets close to zero, the behavior of the generated code will get close to the model. However, it is generally not guaranteed that the discrepancy will be bounded by using a smaller period due to the discrete nature of hybrid systems. For example, small errors in solving the differential equations numerically may lead to a discrete state change that should otherwise not occur, resulting in an entirely different trace thereafter. Thus, validation of the generated code against the originating model is essential for model-based code generation paradigm.

In this work, we propose a framework for automatic code generation and validation for hybrid-systems models to distributed execution environment. Our framework combines and extends previously proposed techniques [9], [28]. In our framework, the code is generated and validated against hybrid-systems models in three steps as illustrated in Fig. 3. First, the model is analyzed whether a transition that is not possible in the model may occur when it is translated into code according to the user assigned update frequency. To prevent such faulty transitions, the model is instrumented such that transitions are taken conservatively considering errors due to discreteness of the code. Second, the instrumented model is analyzed to check whether a transition may be missed. In this stage, each transition is analyzed whether it is enabled long enough compared to the user assigned update frequency. Finally, the instrumented model is fed into the code generator to produce the code. The workflow described in Fig. 3 reflects a translation of the model to the code. Conceptually, this translation progresses from the hybrid-systems model defined in continuous time to the code that runs in a distributed discrete environment. At each stage of the code generation process shown in Fig. 3, there is a successive relaxation of behavioral semantics. Hence, it is essential to carefully analyze and identify criteria for a faithful implementation of the model. This is the principal focus of this paper.

There are a number of issues that need to be addressed to provide guarantees in the generated code. The focus here is on preventing switching discrepancies. The continuous semantics of the model are implemented in the code with the help of numerical methods which introduce an error due to discretization in addition to the round-off and truncation errors on target platforms. These errors along with the order of scheduling of the reads may cause a transition to be falsely enabled. If such a faulty transition is taken, the dynamics of the system may be completely different from the intended model. The example below highlights such a possibility.

TABLE 1
A Run with Faulty Transition

t	$x_1(A_1)$	$x_1(A_2)$	$x_2(A_2)$	State of A_1	State of A_2
0.06	0.3072	0.3072	0.2018	q_0	q_2
0.10	0.3172	0.3072	0.2018	q_0	q_2
0.12	0.3172	0.3072	0.2072	q_0	q_3

Example 3 (Faulty Transition). Consider the vehicle coordination system in Example 1. We will describe a working scenario where the delay in communication of variables causes the system to make a faulty transition.[2] Let us say that the relative period of update for agents A_1 and A_2 be $(5, 3)$ and the actual periods of updates be 0.1 s and 0.06 s, respectively. Also, let $u = 2$, $d_{min} = 0.1$, and $d_{max} = 0.5$, and initial positions of vehicles be $x_1^0 = 0.3072$ and $x_2^0 = 0.2$, from the baseline, initial velocities $v_1^0 = 0$, $v_2^0 = 0$, $\phi(A_1, A_2) = \phi(A_2, A_1) = 0.03$ s, and the current states of agents be q_0 and q_2.

More formally, let the system of agents be executed on two nodes N_1 and N_2. Then, the agents are given by $(A_1, \{x_1, x_2\}, clk, 5, \{x_1(A_1), x_2(A_1)\})$ and $(A_2, \{x_1, x_2\}, clk, 3, \{x_1(A_2), x_2(A_2)\})$ while the implementation is given by $\langle\{N_1, N_2\}, \{(A_1, N_1), (A_2, N_2)\}(((A_1, A_2), 0.03), ((A_2, A_1), 0.03)), 0.02\rangle$, respectively. In the definition, clk is a clock representing the physical time of the system. Table 1 shows a potential run of the system. The table shows the value of variable x_i on agent A_j at times 0.06 s, 0.1 s, and 0.12 s. As the value of the variable could be different on different agents (due to communication delays), we describe the local values by the notation $x_i(A_j)$. In this run, at time 0.12, the difference between vehicles is $0.3172 - 0.2072 = 0.11 (> 0.1)$, but the estimated distance at A_2 is $0.3072 - 0.2072 = 0.0956 < 0.1$. Due to this discrepancy, the agent A_2 makes a faulty transition to q_3.

Aside from faulty transitions, implementations of hybrid-system models are also vulnerable to missed transitions. Insufficient sampling rates, choice of scheduling of reads, etc., may cause a transition to be missed. Missing some transitions may cause the system to end up in a erroneous state. This is illustrated with an example below.

Example 4. Consider the system in Example 1. Consider that the system model is as specified in Example 3, but with actual periods of update to be $(0.25\text{ s}, 0.15\text{ s})$. Also, let $d_{min} = 0.25$, $d_{max} = 0.5$, and the control parameter $u = 0$. $x_1 = 0.48$, $v_1 = 5$, and $v_2 = 4.5$ at $t = 0.15$, and the current state of A_2 be q_2. Further, let $d = x_1 - x_2$ and $\dot{d} = \dot{x_1} - \dot{x_2} \in [0.45, 0.5]$. The guard G_4 is then the condition $d \in (0.25, 0.5)$. On instrumentation, the guard becomes $d \in (0.25 + 0.1 \times 0.5, 0.5 - 0.1 \times 0.5) = (0.3, 0.45)$ as the maximum skew is 0.1, and $\mathcal{L}_2 = 0.5$. A partial run of the system is shown in Table 2. Although instrumenting the guards ensures that there are no faulty transitions now, the transition from q_2 to q_1 is missed in the run. At time $t = 0.3$ s, the agent A_2 instead transits to q_3.

2. It must be mentioned that this is merely an illustrative example, and the values here are for the sake of the example.

TABLE 2
A Run with Missed Transition

t	$x_1(A_1)$	$x_2(A_2)$	State of A_1	State of A_2
0.15	0.48	0.0	q_0	q_2
0.25	0.98	0.0	q_0	q_2
0.30	0.98	0.6862	q_0	q_3

In providing guarantees of faithful implementation, it is desired to have no faulty or missed transitions [10] from the point of view of switching, and errors in variables are bounded from the perspective of dynamics. A scheduling-independent guarantee that the global state of the code be consistent with that of the model is yet another objective. The following definition formally captures this idea.

Definition 4 (Faithful Implementation). *Let VC be the set of all variables and α_x be the maximum bound on the error of a variable x. Given a trace of states of the code $\mathcal{K}$ of model $\mathcal{D}\langle q_0, q_1, \ldots\rangle$, at physical time stamps $\langle clk_0, clk_1, \ldots\rangle$, where $q_i = (q_{A_1}, \ldots, q_{A_n})_i$, if, $\forall i$,*

1. *$\forall x \in VC, |x_{\mathcal{D}} - x_{\mathcal{K}}| < \alpha_x$, where $x_{\mathcal{K}}$ and $x_{\mathcal{D}}$ represent the value of variable in the code and the model, respectively.*
2. *$\forall j : q_{\mathcal{K}} = q_{\mathcal{D}}$ or $q_{\mathcal{K}} = q'_{\mathcal{D}}$ where $q_{\mathcal{K}}$ is the state of the code, $q_{\mathcal{D}}$ and $q'_{\mathcal{D}}$ are the projection of the state of the model onto the code for agent A_i at logical times lt_i (corresponding to clk_i) and $lt_i + h_j$, respectively.*

Then, $\mathcal{K}$ is said to be a faithful implementation of $\mathcal{D}$. If the code satisfies condition 1, then it is said to be bounded numerical error implementation.

Condition 1 states that the error in continuous variables is bounded. Condition 2 states that the state of the code of agent A_j at any given time, corresponds to either the corresponding logical state in the model or corresponds to the next logical state. This tolerance is given because the exact time of scheduling is not specified and if the agent has finished execution for that period, its logical state will reflect the next logical state. However, this is a rather strong criteria to enforce in time-delayed systems where the delays could be different for every agent and even within an agent, the delay in taking a transition can vary based on the communication delay with the agent updating the variables in the guard. For example, in one mode, an agent could depend on x that arrives 0.0001 s late and in another mode, it could depend on y that arrives 0.0002 s later. Therefore, the criteria is relaxed by requiring that code enters the state of the model no later than the maximum possible delay. Formally, we can state the following:

Definition 5 (Relative Faithful Implementation). *Let VC be the set of all variables and α_x be the maximum bound on the error of a variable x. Given a trace of states of the code $\mathcal{K}$ for an agent A_j, $\langle q_0, q_1, \ldots\rangle$, at physical time stamps $\langle clk_0, clk_1, \ldots\rangle$, if, $\forall clk$,*

1. *$\forall x \in VC, |x_{\mathcal{D}}(lt) - x_{\mathcal{K}}(lt)| < \alpha_x$, where $x_{\mathcal{K}}$ and $x_{\mathcal{D}}$ represent the value of variable in the code and the model, respectively, and lt, the logical time in the code.*

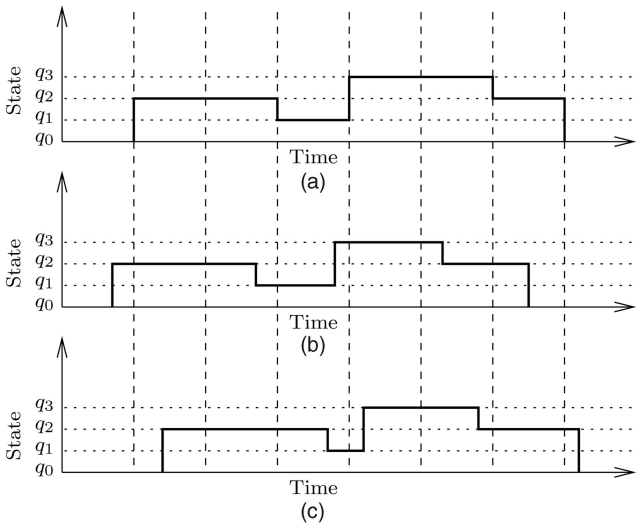


Fig. 4. An example hybrid-system model and its implementations.

2. *$\forall j : q_{\mathcal{K}} = q_{\mathcal{D}}, q_{\mathcal{K}} = q'_{\mathcal{D}}$, or $\exists q'_{\mathcal{K}} : q'_{\mathcal{K}} = q_{\mathcal{D}}$, where $q_{\mathcal{K}}$ is the state of the code, $q_{\mathcal{D}}$, $q'_{\mathcal{D}}$, are the projection of the state of the model onto the code for agent A_i at logical times lt_i (corresponding to clk_i) and $lt_i + h_j$, respectively. $q'_{\mathcal{K}}$ is the state of the code at a time $t < lt_i + \phi_j + \varphi + h_j$, where $\phi_l = \max_l \phi(i, l)$ and φ is the maximum skew due to different rates of updates.*

Then, code for A_j is said to be a relative faithful implementation. If $\forall j$, A_j is a relative faithful implementation, then $\mathcal{K}$ is a relative faithful implementation of $\mathcal{D}$.

As in Definition 4, Condition 1 states that the error in continuous variables is bounded. Condition 2 states that the state of the code of agent A_j at any given time, corresponds to either the corresponding logical state in the model or corresponds to the next logical state (similar to Condition 2 of Definition 4). Further, if the state of the code is not one of these, then this must be due to the communication delay and skew, so the state of the model is entered within time $t < lt_i + \phi_j + \varphi + h_j$. This is because, all the updates to variables would reach an agent in within time $lt_i + \phi_j + \varphi$. At that point, if the agent has not yet been scheduled for the cycle, it will take the transition during that cycle. If it has already been scheduled, then the transition will be taken in the next cycle.

Example 5. Consider the transitions of a hybrid-system model and two implementations as described in Fig. 4. The run records transitions between four states labeled q_0, q_1, q_2, and q_3. The run of the model is shown in Fig. 4a. The dashed vertical line represents the sampling periods, i.e., times at which the discrete implementation updates the variables and states. Implementation in Fig. 4b is a faithful implementation, as the state of the implementation match those of the model at the sampling periods. Assuming the maximum skew and communication delay is half the sampling period, Fig. 4c is a relative faithful implementation of the model, as the transitions in the implementation are taken at the next sampling period where the updated values of variables are available.

3 Addressing the Issues in Reliable Code Generation

Note that even if we use the relaxed notion of correctness, validation of correct implementation is still nontrivial. The reasons include the following:

- Only a small class of differential equations can be solved exactly. For most cases, numerical methods yield an approximate solution. Hence, obtaining numerical bounds on error is often not possible. Errors due to numerical integration of differential equations are, thus, generally analyzed and represented by the O notation, and a constant error bound can be rarely analyzed, if not impossible. The problem is even more complex when we consider switching of differential equations and the precision of the floating-point unit.
- A transition that must be taken to satisfy the invariant may be missed because the transition condition is not evaluated frequently enough.

We believe that a general solution that addresses all the problems in the general hybrid systems is unlikely to exist, because a general solution for a constant error bound of numerical integration is not known. However, for some limited class of hybrid systems (e.g., linear hybrid systems), a constant error bound can be easily obtained. We have addressed some issues related to sampling in previous work (see [29]), but have not integrated all of them into this framework.

In the remainder of this section, we focus on techniques to identify and address switching related issues in reliable code generation of hybrid-system models.

3.1 Preventing Faulty Transitions

Faulty transition is a transition that is taken in the code but is not possible in the model. It occurs either because the transition was taken on the basis of an older value of the variable or because of numerical errors in the variables.

Definition 6 (Faulty Transition). *Let $p_0, \ldots p_n$ be a sequence of positions in the trace of the code. (p_{n-1}, p_n) is a faulty transition if $p_0, p_1, \ldots, p_{n-1}$ is a valid sequence of positions in the model, but $p_0, p_1, \ldots, p_n$ is not.*

Static instrumentation was introduced by Hur et al. [9] as an approach to prevent faulty transitions. The idea there was to instrument the guards and the invariants with maximum possible error in variables and switch conservatively. Once the guards and the invariants have been instrumented, the code generated from $\mathcal{D}$ can be assured of no faulty transitions. Though this approach has the advantage that guarantees can be given statically, there is a trade-off associated with this conservative switching. Since the guard and invariant sets are shrunk, the probability of not taking a transition increases. Yet another disadvantage is that it is not always possible to determine this error bound beforehand as with most differential equations, it is only possible to get a local estimate of error which is only available at runtime. Therefore, in this work, we introduce a technique to dynamically instrument the guards and invariants based on the runtime estimates of errors. The main advantage of this approach, as opposed to static instrumentation, is that it shrinks the guard and invariant sets by a smaller amount, and thereby reducing the risk of missed transitions.

Errors in variables could be due to numerical errors in solving the differential equations, or are timing-induced due to the different rates of execution of the agents. The numerical errors introduced during the solution of differential equations are of the two types: the truncation error due to truncations in Taylor series expansions and the round-off error due to a finite precision of real numbers in the computer. Both the truncation error and the round-off errors are accumulated from during the integration process and can be quite dominant sources of numerical error as the number of integration steps increases. Determining the error bounds for these errors accurately is hard. While computing the exact or close enough bounds is difficult and tricky, computing approximate error estimates, at least within an order of magnitude is possible. For ordinary differential equations, classical methods like Runge-Kutta method [27] can be used to compute the order of error bound given the step size of simulation. This is a harder proposition for a system of differential and algebraic equations (see [30] for some details). For the purposes of instrumenting the guards, a technique for bounding the numerical errors has been presented by Hur et al. [31]. In this work, we assume that reliable bounds for numerical errors have been computed a priori.

As mentioned above, the other source of error in variables is timing-induced, i.e., due to the different rates of execution of the agents. This can be estimated from the maximum delay in communication between the agents. The communication delay itself is obtained by monitoring and the maximum *skew*, φ (from Definition 5).

Once we have computed the bounds on numerical errors, we can define guard and invariant set instrumentation as follows:

Definition 7 (Instrumentation). *Let p be a state of agent A_j with $E_{A_j}(p)$ being the set of discrete transitions, and the interval under consideration be $[lt, lt + \Delta]$. If the guard set $g \in G_{A_j}(e)$, $e \in E_{A_j}$ is of the form, $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$, the invariant $I_{A_j}(p) = \bigwedge_i x_i \in [l'_{x_i}, u'_{x_i}]$, φ and ϕ compute the skew and delay between the agents, then, the instrumented guards and invariant are given by,*

$$g^{inst} = \bigwedge_i x_i \in \left[l_{x_i} + \gamma_{p,x_i} + \mathcal{L}_{2_{x_i}}\delta_{x_i},\, u_{x_i} - \gamma_{p,x_i} - \mathcal{L}_{2_{x_i}}\delta_{x_i}\right] \quad (1)$$

$$I^{inst} = \bigwedge_i x_i \in \left[l'_{x_i} + \gamma_{p,x_i} + \mathcal{L}_{2_{x_i}}\delta_{x_i},\, u'_{x_i} - \gamma_{p,x_i} - \mathcal{L}_{2_{x_i}}\delta_{x_i}\right] \quad (2)$$

where $\delta_{x_i} = \varphi(A_i, A_j) + \phi(A_i, A_j)$, x_i is updated by agent A_i, with $\dot{x}_i \in [\mathcal{L}_{1_{x_i}}, \mathcal{L}_{2_{x_i}}]$, and γ_{p,x_i} is the round-off and truncation error in x_i in the state p.

Example 6. Consider the system in Example 3 in the time interval [0, 1]. If $d = x_1 - x_2$, then, $\dot{d} = \dot{x_1} - \dot{x_2} = t$. Since $t \in [0.05, 1]$, $\dot{d} \in [0.05, 1]$. Now, given $\phi(A_1, A_2) = 0.03$, skew at $t = 0.12$ as 0.02, and assuming the bound on round-off and truncation errors is 0.001, the transition guard, $x_1 - x_2 \leq 0.1$ upon instrumentation becomes $x_1 - x_2 \leq (0.1 - 0.001 - 1 \cdot (0.02 + 0.03)) = x_1 - x_2 \leq 0.049$ which prevents the faulty transition at $t = 0.12$.

The theorem below formally states that instrumentation prevents faulty transitions.

Theorem 1. *Let the code $\mathcal{K}$ of the model $\mathcal{D}$ be implemented on a distributed platform. Let for every agent A_j, p be the current state with $I_{A_j}(p)$ the set of invariants in that state, and $G_{A_j}(e)$ the set of guards. If $\forall G \in G_{A_j}(e)$ that evaluate to true, and $\forall x$, G is instrumented as given in Definition 2 then there will be no faulty transitions.*

Proof (Sketch). The essential idea behind *instrumentation* is to reflect the effect of numerical errors and synchronization errors in the generated code to the invariants and guards of each position. The theorem is proved by showing that the resulting hybrid automata produce a sound trace on discrete transition steps. This proof is accomplished, in turn, by proving each of the following statements:

1. [9, Lemma 1] Every run of a system of DCHA has an equivalent run in the originating system of hybrid-system automata if in the originating system the dynamics do not change (insensitive) in between the sampling interval. The following definition captures the measure of insensitivity more formally.

Definition 8 (h-Insensitivity [9, Definition 13]). *Given a communicating hybrid automata $\mathcal{A}$, the invariant $I_x(p)$ corresponding to a position p and a continuous variable x is said to be insensitive if every x is such that $x(t) \in I_x(p)$ (i.e., x satisfies its invariant), $x(t+h) = x(t) + \int_t^{t+h} F_p(x)dt \in I_x(p)$ implies $x(t+\delta) = x(t) + \int_t^{t+\delta} F_p(x)dt \in I_x(p)$ for all $\delta \in [0, h]$, where F is the derivative of x, and $x(t)$ denotes the valuation of x at time t. When all invariants are h-insensitive, then the hybrid automata $\mathcal{A}$ is said to be h-insensitive. If every model $\mathcal{A}$ is h-insensitive, then we say that the system of communicating hybrid automata $\mathcal{C}$ is h-insensitive.*

1. [9, Theorem 1] Given a communicating hybrid automata $\mathcal{A}$, and its corresponding instrumented automata B, assuming that A is $h_{\mathcal{B}}$-insensitive, then $\mathcal{B}$ always produces a safe run, i.e., a run that is always included in that of A.
2. [9, Theorem 2] Given a system of communicating hybrid automata $\mathcal{C} = \langle(\mathcal{A}, SV)_1, \ldots, (\mathcal{A}, SV)_n\rangle$ and their corresponding instrumented versions $\mathcal{C} = \langle(\mathcal{B}, SV)_1, \ldots, (\mathcal{B}, SV)_n\rangle$, such that the automata $\mathcal{B}_j$ is $h_{\mathcal{B}_j}$ insensitive, then, every run of the system of communicating hybrid automata is included in that of the originating system.

For a formal treatment of the model, the statements, and the proofs, we refer the reader to the work by Choi, Hur, and others [9], [32], [33]. □

Notice that in Example 6, the instrumentation reduces the guard interval substantially. It is possible that with the shrinking of the guard set, the transition is missed completely. The next section will analyze and derive a condition to check for missed transitions and possibly avoid them by higher sampling.

3.2 Preventing Missed Transitions

Missed transitions are transitions that are enabled in the model but not taken in the code. They occur either because the guard is not evaluated sufficiently or scheduling affected the order of evaluation.

Definition 9 (Missed Transition). *Let $p_0, \ldots p_n$ be a sequence of positions in a terminated trace of the code, i.e., $p_n = \perp$, where $\perp$ denotes a state that violates the invariant. There is a* missed transition *at p_{n-1}, if $p_0, p_1, \ldots, p_{n-1}$ is a valid sequence of positions in the model, but $p_0, p_1, \ldots, p_n$ is not.*

In general, a transition will not be missed, if it stays enabled long enough to be detected. The theorem below gives a sufficient condition to prevent missed transitions.

Theorem 2 [10]. *Let the code $\mathcal{K}$ of the model $\mathcal{D}$ be implemented on a distributed platform, h_j be the period of sampling in agent $\mathcal{A}_j$. Let I be an instrumented invariant in a state and $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}], g \subseteq I$ represent the instrumented guard of a transition in that state. If lt represents the current logical time at A_j, $x_i(lt)$ the current estimate of x_i at A_j, and if T_{x_i} are defined as,*

$$T_{x_i}(k) = \begin{cases} \left[lt + \frac{l_{x_i} - x_i(lt)}{\mathcal{L}_{k_{x_i}}} + \delta_{max}, lt + \frac{u_{x_i} - x_i(lt)}{\mathcal{L}_{k_{x_i}}} + \delta_{min}\right], \\ \qquad \text{if } (x_i(lt) < l_{x_i}), \dot{x}_i > 0, \\ \left[lt + \frac{u_{x_i} - x_i(lt)}{\mathcal{L}_{k_{x_i}}} + \delta_{max}, lt + \frac{l_{x_i} - x_i(lt)}{\mathcal{L}_{k_{x_i}}} + \delta_{min}\right], \\ \qquad \text{if } (x_i(lt) > u_{x_i}), \dot{x}_i < 0, \end{cases}$$

then, the transition will not be missed if,

$$\left\| \bigcap_i \left(\bigcap_{k=1,2} T_{x_i}(k) \right) \right\| \geq 2h_j$$

where $\delta_{min} = \varphi_{min} + \phi$, $\delta_{max} = \varphi_{max} + \phi$ between agents A_i and A_j, then, the transition will be detected and will not be missed if they are taken as soon as enabled.

Proof (Sketch). The proof of the theorem is sketched in two parts. First, a condition on the overlap of guard and invariant that will allow detection of the enabling of the transition is derived. Then, given that the guard is of the form $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$, a sufficient condition to meet this overlap, based on the periods of execution of agents is presented.

To prove the first statement, consider a task-period set $\Omega = \{(\tau_i, h_i)\} 1 \leq i \leq n$. Each task τ_i is treated as a periodic task with period h_i executing in a distributed environment. Let the execution time of τ_i be η_i and this is scheduled to run every h_i time units. Note that η_i here includes both execution time and also perhaps communication delay associated. Also, the time used here is in the reference frame at the processor executing task τ_i. Therefore, in the worst case, τ_i might be scheduled at time jh_i and a guard might be enabled (in the code, perhaps on a different processor) immediately after that, i.e., at time $jh_i + \epsilon$, $\epsilon > 0$ and be detected only when τ_i is next scheduled to run which may be as late as $(j+2)h_i - \eta_i$. Since eager switching is assumed, this transition will be taken at $(j+2)h_i - \eta_i$. Thus, if a guard is not enabled at $(j+2)h_i - \eta_i$, it will go undetected and this

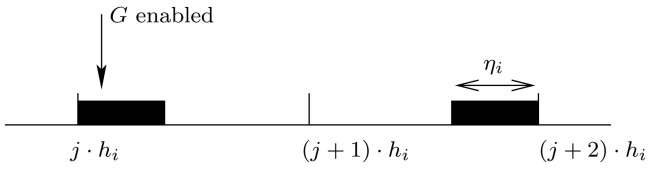


Fig. 5. Worst case scenario.

will result in a missed transition. Hence, the guard should stay enabled for at least $((j+2)h_i - \eta_i) - (kh_i + \epsilon) = 2h_i - \eta_i - \epsilon$ time units. Since ϵ is arbitrary, to be safe, it should stay enabled in the code for $2h_i$ time units so that the transition is not missed. This is illustrated in Fig. 5. Now, consider the guard set $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$. Let the current logical time be lt and current values of variables at agent A_j given by $x_i(lt)$. Consider the case where $x_i(lt) < l_{x_i}$ and $x_i(lt) > 0$, the argument for the case where $x_i(lt) > u_{x_i}$ and $x_i(lt) < 0$ is similar. Since $\dot{x}_i \in [\mathcal{L}_{1_{x_i}}, \mathcal{L}_{2_{x_i}}]$, $\dot{x}_i$ can utmost grow as $\mathcal{L}_{2_{x_i}}$. The guard on x_i, ($[l_{x_i}, u_{x_i}]$) will then be enabled for the time interval

$$T_2 = \left[lt + \frac{l_{x_i} - x_i(lt)}{\mathcal{L}_{2_{x_i}}} + \delta_{max}, lt + \frac{u_{x_i} - x_i(lt)}{\mathcal{L}_{2_{x_i}}} + \delta_{min} \right],$$

assuming that in the worst case, the notification for enabling of the guard gets to A_j in time δ_{max} and the notification for exiting comes at δ_{min}. This is true because x_i is continuous and the guards are assumed to be disjoint in time; otherwise, there could be resets and the dynamics of x_i would be different. Similarly, if $\dot{x}_i$ grows as slow as $\mathcal{L}_{1_{x_i}}]$, then, it will be enabled for the time interval of

$$T_1 = [lt + \frac{l_{x_i} - x_i(lt)}{\mathcal{L}_{1_{x_i}}} + \delta_{max}, lt + \frac{u_{x_i} - x_i(lt)}{\mathcal{L}_{1_{x_i}}} + \delta_{min}].$$

Therefore, if $T_1 \cap T_2 \neq \emptyset$, then it represents the time interval for which guard on x_i will be enabled. Hence, considering the time interval for each of the x_is, the time interval when the guard will definitely be true. From the above arguments, one can conclude that a condition in Theorem 2 gives a *sufficient condition* for preventing missed transitions, if the transitions are taken as soon as they are detected. □

The example below illustrates a case where a transition is missed when the sufficient condition is not met.

Example 7. Consider the case of Example 4. As a quick check, if the system evolves as fast as 0.5, then

$$T_2 = (\frac{0.48 - 0.45}{0.5} + 0.1, \frac{0.48 - 0.3}{0.5} + 0.05) = (0.16, 0.41).$$

Similarly,

$$T_1 = (\frac{0.48 - 0.45}{0.45} + 0.1, \frac{0.48 - 0.3}{0.45} + 0.05) = (0.167, 0.45).$$

$\|T_1 \cap T_2\| = 0.243 \not\geq 2(0.15)$ does not satisfy the sufficient condition for preventing missed transitions. However, if the period of execution is chosen to be 0.12, it can be seen that the transition will not be missed.

Theorems 1 and 2 give us a sufficient condition to ensure a relative faithful implementation that is recorded in the following result:

Theorem 3. *Let the code $\mathcal{K}$ of the model $\mathcal{D}$ be implemented on a distributed platform. If in the code for every agent A_i, every $G \in G_{A_j}$ is dynamically instrumented as per Definition 7, every guard and the corresponding invariant satisfy the condition of overlap in Theorem 2, and all variables in $\mathcal{K}$ have bounded error, then, $\mathcal{K}$ is a relative faithful implementation of $\mathcal{D}$.*

Proof. For code $\mathcal{K}$ to have a relative faithful implementation, it has to meet the two criteria outlined in Definition 5. If $\mathcal{K}$ has bounded error for each variable, then it meets the first condition in the definition.

Further, if $\mathcal{K}$ is implementing a dynamically instrumented model with the bounds as per Definition 7, there will be no faulty transitions in the model. If the instrumented model satisfies the conditions of overlap between the guard and the invariant in Theorem 2, there will be no missed transitions because of shrunk instrumented guards. This fact, along with the instrumented transition policy being an urgent transition policy will ensure that all enabled transitions will be taken.

With no faulty or missed transitions, the code $\mathcal{K}$ will maintain the state of the model $\mathcal{D}$ once each transition is taken. Consider the second condition of Definition 5. If all updates have been received by the time the guard and invariants are evaluated, then the implementation will progress into the next state before the model (i.e., $q_{\mathcal{K}} = q'_{\mathcal{D}}$ of Definition 5). Otherwise, all the updates will arrive within $\phi_j = \max_i \phi(i,j) + \varphi$ after they have been updated. If the agent has not been scheduled for that cycle, the transition will be taken in that cycle itself, and it has already been scheduled, then the transition will be taken next time the agent is scheduled to execute.[3] In either case, we have, the time of transition in the code t is such that $t < lt_i + \phi_j + \varphi + h_j$. Therefore, $\exists q'_{\mathcal{K}} : q'_{\mathcal{K}} = q_{\mathcal{D}}$ of Definition 5.

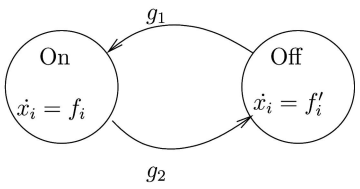
From these observations, it follows that $\mathcal{K}$ is a relative faithful implementation of the model $\mathcal{D}$. □

To conclude this section, we would like to add that, the result on missed transitions is only useful to detect whether an instrumented guard can still allow for transitions out of a state. This by itself does not ensure any liveness property of the system. It is the eager transition policy, i.e., forcing the code to take the transition as soon as a guard is enabled, that enforces liveness of the system.

4 Case Study: Heater Model in CHARON

As a case study to illustrate the framework introduced, consider the heater benchmark controller as described in [34] with three rooms and one heater where the three rooms communicate their temperature to the heater. The temperature of a room depends on other rooms, the outside

3. Note that, the above result assumes that the assumption that at most one transition is enabled still holds after the guards are instrumented.



$$f_i := c_i + b_i(u - x_i) + \sum_{i \neq j} a_{i,j}(x_j - x_i)$$
$$f'_i := b_i(u - x_i) + \sum_{i \neq j} a_{i,j}(x_j - x_i)$$
$$g_1 := (h = i) \wedge (x_i \leq on_i) \quad g_2 := (h \neq i) \vee (x_i \geq off_i)$$

Fig. 6. Hybrid-Systems model for the Room Heater Thermostat.

temperature, and on whether the heater is present in the room. The heater is controlled by a typical thermostat, i.e., it is switched on if the temperature is below a certain threshold, and off if it is beyond a higher threshold. When the temperature in any room falls below a certain desired level, it may get a heater, provided the temperature in that room is significantly higher. The desired objective is to maintain all the three rooms within the comfortable temperature range. A heater is moved from room j to room i if 1) room i has no heater, 2) $x_i \leq get$, and 3) $x_j - x_i \geq dif$ where get and dif are constants and can differ for each room.

The hybrid-systems model for heater is described in Fig. 6. In the model, x_i is the temperature in each room and $a_{i,j}$, b_i, c_i are control parameters. The heat exchange between rooms i and j is assumed to be symmetric, i.e., $a_{i,j} = a_{j,i}$. Each room has upper and lower thresholds on_i and off_i, respectively.

The heater controller is simulated as a distributed system. Each room has a temperature sensor that broadcasts its temperature according to the schedule. Since the network communication introduces delays, the current temperature in Room i is denoted as *real temperature* x_i and the last temperature which has been reported to the controller as the *used temperature* x'_i. The error bound is the absolute difference between x_i and x'_i.

4.1 Implementation in CHARON

The implementation in CHARON includes the following parts: the environment with the three rooms, the heater controller, and the tree schedule. The environment with the three rooms is modeled as specified in Fig. 6.

Listing 2. Part of the CHARON code for Room 1.

```
1 mode HeatedMode1(real u, real heatVal) {
    write analog real x1;
    read analog real x2, x3, h;
    diff{d(x1) == −0.9 ∗ x1 + 0.5 ∗ x2 + 0.4 ∗ u + heatVal}
  }
6
  mode UnheatedMode1(real u) {
    write analog real x1;
    read analog real x2, h;
    diff{d(x1) == −0.9 ∗ x1 + 0.5 ∗ x2 + 0.4 ∗ u}
11 }
```

It is a straightforward implementation, and Listing 2 shows the two modes, heated and unheated, of the CHARON code for Room 1. The variable u is initialized with 4, and the variable $heatVal$ is initialized with 6. Rooms turn the heater off at a temperature of 25.

The heater controller also directly follows from the specification given in Fig. 6. However, it is important that the controller does not operate on the real temperature x_i but instead operates on the used temperature x'_i. The transceiver updates the used temperature as it receives new data sent by the sensors. Listing 3 shows part of the CHARON code for the heater control. It reads the values x'_i represented as xi_used. The values for the variables $thld$ and dif are initialized with 10 and 1, respectively.

Listing 3. Part of the CHARON code for the heater control.

```
mode TopModeHeater (real thld, real dif){
  read analog real x1_used, x2_used,x3_used;
  write analog real h;
4 ...
    trans from default to q1 when true do{h = 1;}
    trans from q1 to q2 when ((x2_used < = thld) &&
            (x1_used-x2_used) >= dif) do {h = 2; ...}
    trans from q1 to q3 when ( (x3_used <= thld) &&
9           (x1_used-x3_used) >= dif) do {h = 3; ...}
    trans from q2 to q3 when ( (x3_used <= thld) &&
            (x2_used-x3_used) >= dif) do {h=3; ...}
  ...}
```

Finally, the third part is the transceiver. The schedule is implemented in a round robin fashion, and it assigns $x'_i = x_i$ if its x_i's slot.

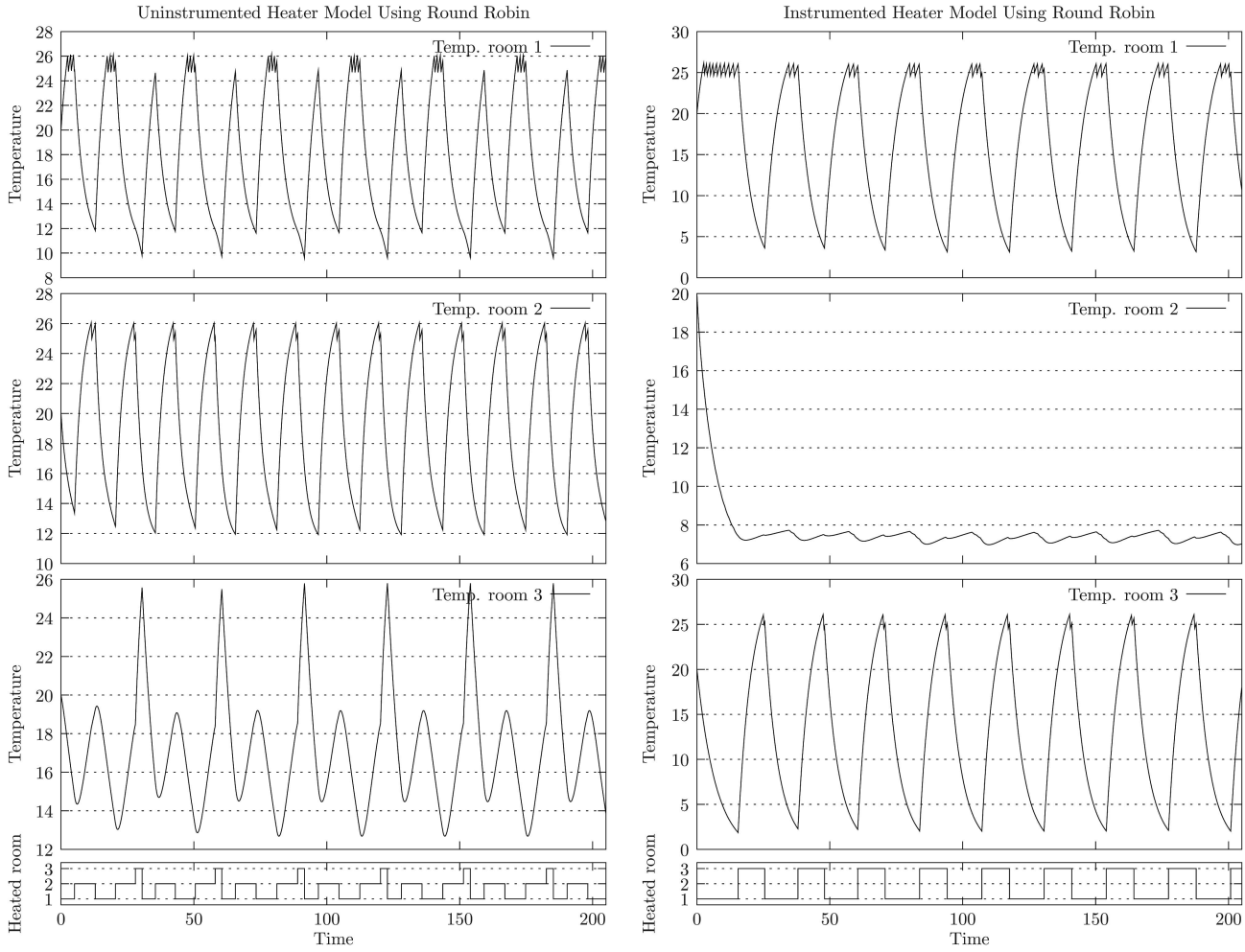
Listing 4. Part of the CHARON code for the tree schedule.

```
    mode CommChannel(real sl_len, real thld, real
    dif) {
    write analog real x1_used, x2_used, x3_used,
    dec;
3  read analog real x1, x2, x3, h;
    private analog real tclk, clk;
   ...
    trans from default to reset when true
8       do {tclk = clk;dec = 0;}
    trans from reset to r1 when
          ( (tclk < (clk-slot_lenght)) )
          do {tclk = clk; x1_used = x1; dec = 0;}
    trans from r1 to r2 when
13        ((tclk < (clk-slot_length)))
          do {tclk = clk;x2_used = x2;}
    trans from r2 to reset when
          ( (tclk < (clk − slot_length)) )
          do {tclk = clk; x3_used = x3; dec = 1;}
   diff {d(clock) == 1.0}
...}
```

4.2 Faulty Transitions and Instrumentation

Listing 4 shows part of the CHARON code for our implementation of the schedule. The agent reads the real temperatures x_i and writes the used temperature values x'_i. Depending on the switching logic, it updates different values.

Listing 5 describes the detection of faulty transitions in the system. The listing shows one particular transition



(a) Uninstrumented version.

(b) Instrumented version

Fig. 7. Heater simulation.

where the heater moves from Room 1 to Room 2. The transitions are named *q1q2ok* and *q1q2faulty*. The transition *q1q2ok* is a valid transition, because the guard condition holds on both, the used values x_i' and the real values x_i. The transition *q1q2faulty* is a faulty transition, because the guard condition holds only for the values x_i' but not for the values x_i. The evaluation simply counts the number of times the system takes a transition with the suffix *"faulty."*

Listing 5. Detecting faulty transitions in the heater.

```
...
    trans q1q2ok from q1 to q2 when (
      ((x2_used <= thld) &&
       (x1_used-x2_used) >= dif) &&
      ((x2 <= thld) && (x1 − x2) >= dif))
)do {h = 2; ...}

    trans q1q2faulty from q1 to q2 when (
      ((x2_used <= thld) && (x1_used-x2_used)
      >= dif) &&
      !((x2 <= thld) && (x1-x2) >= dif))
) do {h = 2;...}
...
```

Listing 6 describes the procedure for instrumenting guards. For both transitions, the safety margin is added by shrinking the guard condition. For the threshold value, instr_t is subtracted, and for the difference, instr_d added. The simulations are then rerun and the number of times which the system takes a transition with the suffix *"faulty"* is counted.

Listing 6. Guard instrumentation of the heater.

```
...
   trans q1q2ok from q1 to q2 when (
     ((x2_used <= (thld-instr_t)) &&
   (x1_used-x2_used) >= (dif+instr_d)) &&
     ((x2 <= thld) && (x1 − x2) >= dif))
     ) do{h = 2; ...}

   trans q1q2faulty from q1 to q2 when (
((x2_used <= (thld-instr_t)) &&
   (x1_used-x2_used)>= (dif+instr_d)) &&
   !((x2 <= thld) && (x1 − x2) >= dif))
```

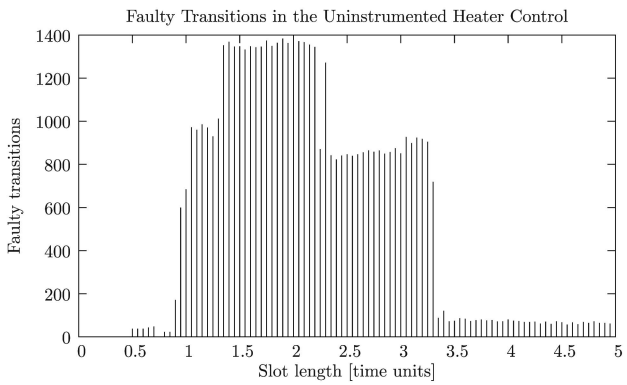


Fig. 8. Faulty transitions in the heater simulation with uninstrumented guards.

```
    ) do {h=2; ...}
...
```

4.3 Evaluation

Fig. 7a shows the results of simulating the hybrid-systems model with uninstrumented control. The figure consists of two general parts: the temperature display of the rooms and the control status. The three top parts show the real temperature of each room over the time of the simulation. Each room was initialized with a temperature of 26. With reference to the model in Fig. 6, the values of get, dif, on, and off are 15, 2, 25, and 26. The bottom part shows the status of the heater over the time of the simulation. Only one heater was used with the condition that it can only be in one room at any given time. The heater starts off in Room 1 and it initially stays there, because all rooms are above the threshold value. Then, it starts switching between different rooms to heat them as necessary.

However, the uninstrumented control can result in faulty transitions. The faulty transition typically occurs when the heater starts heating up one room that is close, but below the threshold. In the switching condition, the control might still use a value where the heater is still below the threshold, while in reality, it is already above the threshold. Fig. 8 shows the number of observed faulty transitions on 10,000 simulation steps.

Once, the guard conditions are instrumented with the value of 10 for the threshold and the difference, there are no more faulty transitions (hence, no equivalent to Fig. 8 for this case). Fig. 7b shows the instrumented version of the heater control. It clearly shows that in this case the price of a correct implementation is a tardier control. To our surprise, the controller never directly heats Room 2. This room, sitting in between rooms 1 and 3, gets heated indirectly through the heat transfer (loss) of Room 1 and Room 3. Choosing different starting conditions cause the heater to move to all rooms, however, we found this setting to be a more interesting one, because it demonstrates that faulty transitions can lead to a very different behavior.

5 Conclusions

Hybrid-systems-based design is a promising yet challenging approach for producing reliable embedded software. Providing formal guarantees is difficult due to the semantic differences between the model and the code arising as a result of discretization and communication delays.

This paper has presented an approach to guarantee faithful switching semantics that involves preventing missed and faulty transitions. In contrast to related and prior work, this work has defined a notion of relative faithful implementation for systems. A runtime instrumentation technique to prevent faulty transitions has been proposed and a sufficient condition to prevent missed transitions in the generated code has been identified. These concepts have been illustrated via an implementation of a model heater system. In the future, the focus will be to fully integrating the infrastructure into the CHARON development environment and provide comprehensive techniques to detect missed transitions in the code at runtime.

In conclusion, we discuss some aspects of the techniques introduced here, and their applicability to large-scale real-world systems. First off, is the issue of scalability. The work here mainly focuses on preventing faulty transitions and detecting missed transitions. The main requirement for instrumentation is the bounds on the errors, which depends on the system at hand and might be tricky. However, Instrumentation itself is a fast operation. Checking missed transitions depends on computing the intersection of sets, which may be hard for complex sets. This could potentially be the only performance bottleneck of this framework. However, it must be noted that there exist techniques for efficient computation of the intersection (see [35]) and the procedure is performed offline, and therefore, it is only a one-time cost.

The abstractions for platform and code consider only the basic of all the actual implementation effects and make several assumptions. These effects can potentially weaken some of the results presented in this work (e.g., instrumentation bounds). However, we do expect that similar techniques would be sufficient to handle these aspects of real-world implementations. The aim of this work is to establish a sound theory at categorizing some of the implementation effects. More artifacts of the implementation can be incorporated into the model as deemed necessary.

Instrumentation can cause tardier control as shown in the case study. We consider this a positive result as it provides motivation to study optimization and trade-offs between incorrect implementation (missed or faulty transitions) and faster controller response. It is not part of this framework to characterize this trade-off, since our motivation was to produce a sound implementation of a given model.

Finally, the faithful implementation criteria introduced in this work may not be enough to enforce a strict correlation between the model and its implementation, especially when one considers the introduced tardiness. There have been a number of metrics introduced in literature (e.g., [36], [38]) that relate to approximating continuous systems. Currently, many of these metrics have yet to be fully extended to hybrid systems. We expect that metrics developed for hybrid systems could be incorporated into our framework.

ACKNOWLEDGMENTS

This research was supported in part by NSF CNS-083452, NSF CNS-0720703, NSF CNS-0720518, NSF CNS-0931239, NSERC DG 357121-2008, and ORF RE03-045.

REFERENCES

[1] N. Martin, "Lock Who's Talking: Motorola's c.d. Team," LockSmart Online Article, Nov. 1998.
[2] R.N. Charette, "This Car Runs on Code," *IEEE Spectrum,* Feb. 2009.
[3] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, no. 2, pp. 183-235, Apr. 1994.
[4] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky, "Generating Embedded Software from Hierarchical Hybrid Models," *SIGPLAN Notices,* vol. 38, no. 7, pp. 171-182, 2003.
[5] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231-274, 1987.
[6] 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,* vol. 138, pp. 3-34, 1995.
[7] O. Maler, Z. Manna, and A. Pnueli, "From Timed to Hybrid Systems," *Proc. Research and Education in Concurrent Systems (REX) Workshop, Real-Time: Theory in Practice,* 1991.
[8] F. Ivancic, "Report on Verification of the Mobies Vehicle-Vehicle Automotive Oep Problem," http://citeseer.ist.psu.edu/ivancic02report.html, 2002.
[9] Y. Hur, J. Kim, I. Lee, and J.-Y. Choi, "Sound Code Generation from Communicating Hybrid Models," *Proc. Int'l Workshop Hybrid Systems: Computation and Control (HSCC),* pp. 432-447, 2004.
[10] M. Anand, J. Kim, and I. Lee, "Code Generation from Hybrid Systems Models for Distributed Embedded Systems," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC),* pp. 166-173, 2005.
[11] M. Anand, S. Fischmeister, J. Kim, and I. Lee, "Distributed-Code Generation from Hybrid Systems Models for Time-Delayed Multirate Systems," *Proc. Fifth ACM Int'l Conf. Embedded Software (EMSOFT '05),* pp. 210-213, 2005.
[12] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular Specification of Hybrid Systems in CHARON," *Proc. Int'l Workshop Hybrid Systems: Computation and Control (HSCC),* http://citeseer.ist.psu.edu/article/alur00modular.html, pp. 6-19, 2000.
[13] P. Traverse, I. Lacaze, and J. Souyris, "Airbus Fly-By-Wire: A Total Approach to Dependability," *Proc. IFIP World Congress,* pp. 191-212, 2004.
[14] "Rational Rose," *IBM,* http://www-306.ibm.com/software/awdtools/developer/rose/, 2010.
[15] TargetLink, http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm, 2010.
[16] "Simulink," *The MathWorks,* http://www.mathworks.com/products/simulink/, 2010.
[17] G. Berry, *The Foundations of Esterel.* MIT Press, http://citeseer.ist.psu.edu/62412.html, 2000.
[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proc. IEEE,* vol. 79, no. 9, pp. 1305-1320, Sept. 1991.
[19] N. Halbwachs and P. Raymond, "A Tutorial of Lustre," http://citeseer.ist.psu.edu/halbwachs01tutorial.html, 2009.
[20] A. Deshpande, A. Göllu, and P. Varaiya, "SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata," *Hybrid Systems IV,* Springer, 1996.
[21] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Luvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity—The Ptolemy Approach," *Proc. IEEE,* vol. 91, no. 1, pp. 127-144, Jan. 2003.
[22] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proc. IEEE,* vol. 91, no. 1, pp. 145-164, Jan. 2003.
[23] A.R. Girard, A.S. Howell, and J.K. Hedrick, "Model-Driven Hybrid and Embedded Software for Automotive Applications," *Proc. Second RTAS Workshop Model-Driven Embedded Systems (MoDES '04),* 2004.
[24] B. Shah, R. Dennison, and J. Gray, "A Model-Driven Approach for Generating Embedded Robot Navigation Control Software," *Proc. 42nd Ann. Southeast Regional Conf. (ACM-SE 42),* pp. 332-335, 2004.
[25] T. Stauner, "Discrete-Time Refinement of Hybrid Automata," *Proc. Fifth Int'l Workshop Hybrid Systems: Computation and Control (HSCC '02),* pp. 407-420, 2002.
[26] T.A. Henzinger and P.H. Ho, "Algorithmic Analysis of Nonlinear Hybrid Systems," *Proc. Seventh Int'l Conf. Computer Aided Verification,* P. Wolper, ed., vol. 939, pp. 225-238, http://citeseer.ist.psu.edu/henzinger96algorithmic.html, 1995.
[27] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing,* second ed. Cambridge University Press, 1999.
[28] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky, "Generating Embedded Software from Hierarchical Hybrid Models," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '03),* 2003.
[29] M. Anand, J. Kim, S. Fischmeister, and I. Lee, "Generating Sound and Resource-Aware Code from Hybrid System Models," *Model-Driven Development of Reliable Automotive Services,* pp. 48-66, 2008.
[30] K.E. Brenan, S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations,* second ed. SIAM, 1996.
[31] Y. Hur, J.-H. Sim, J. Kim, and J.-Y. Choi, "Ensuring Sound Numerical Simulation of Hybrid Automata," *J. Computing Science and Eng.,* vol. 3, no. 2, pp. 73-87, June 2009.
[32] J.-Y. Choi, Y. Hur, and I. Lee, "IHA: Ensuring Sound Numerical Simulation of Hybrid Automata," Technical Report MS-CIS-03-06, Univ. of Pennsylvania, 2003.
[33] J.-Y. Choi, Y. Hur, J. Kim, and I. Lee, "Sound Synchronization of Communicating Hybrid Automata," Technical Report MS-CIS-03-30, Univ. of Pennsylvania, 2003.
[34] A. Fehnker and F. Ivancic, "Benchmarks for Hybrid Systems Verification," *Proc. Int'l Workshop Hybrid Systems: Computation and Control (HSCC),* pp. 326-341, http://citeseer.ist.psu.edu/fehnker04benchmarks.html, 2004.
[35] G.E. Blelloch and M. Reid-Miller, "Fast Set Operations Using Treaps," *Proc. 10th Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA '98),* pp. 16-26, 1998.

[36] A.A. Julius and G.J. Pappas, "Approximate Equivalence and Approximate Synchronization of Metric Transition Systems," *Proc. 44th IEEE Conf. Decision and Control,* Dec. 2006.
[37] C. Kossentini and P. Caspi, "Approximation, Sampling and Voting in Hybrid Computing Systems," *Proc. Int'l Workshop Hybrid Systems: Computation and Control (HSCC '06),* 2006.

Madhukar Anand received the BS and MS degrees in mathematics and computing from the Indian Institute of Technology (IIT), Kharagpur, and the PhD degree in computer and information science from the University of Pennsylvania in 2008. From Fall of 2008, he is working with the Data Center Routing Team at Cisco Systems. His research interests include real-time and embedded systems, networked embedded systems, data center networking, formal methods, hybrid systems, and wireless sensor networks. Among other awards, he has won the Institute Silver Medal for academic excellence from IIT Kharagpur.

Sebastian Fischmeister received the Dipl-Ing degree in computer science from Vienna University of Technology, Austria, in 2000, and the PhD degree in computer science from the University of Salzburg, Austria, in December 2002. He is an assistant professor with the Department of Electrical and Computer Engineering at the University of Waterloo in Canada. His primary research interests include software technology and distributed systems for real-time embedded systems. He is a member of the IEEE.

Yerang Hur received the BS, MS and PhD degrees at Seoul National University, Korea, and the University of Pennsylvania, respectively. His specialties include design of embedded software, radio resource management and power saving method for wireless broadband communication systems, QoS architecture for real-time systems, parallel and distributed simulation, and design of high-assurance computer systems. At the University of Pennsylvania, he was an architect of CHARON toolset developed for embedded system design. In 2004, he joined Posdata America R&D Center, where he has designed the MAC layer and developed algorithms for one of the first certified mobile WiMAX base stations in the world. Also, he has served as the leader of the various subteams in the Technical Working Group of WiMAX Forum since 2006 and was awarded the WiMAX Forum president individual contribution award in 2007. He has contributed to IEEE 802.16e and IEEE 802.16m standards as well with more than 100 official submissions from 2005 to 2009, and currently working on design of 4G broadband communication systems including LTE systems and mobile WiMAX Release 2.0 systems. He is a member of the IEEE.

Jesung Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1998, respectively. He is a senior team leader at The MathWorks. Before joining The MathWorks, he was a postdoctoral researcher at the University of Pennsylvania from 2002 to 2005, and at Seoul National University from 2000 to 2002. From 1998 to 2000, he worked for Hyundai Electronics as a research engineer. His research interests include model-based design, embedded systems, and computer system architecture. He is a member of the IEEE.

Insup Lee received the BS degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the PhD degree in computer science from the University of Wisconsin, Madison, in 1983. He is the Cecilia Fitler Moore Professor of computer and information science and the director of PRECISE Center at the University of Pennsylvania. His research interests include real-time systems, embedded systems, formal methods and tools, medical device systems, cyber-physical systems, and software engineering. The theme of his research activities has been to assure and improve the correctness, safety, and timeliness of real-time embedded systems. He has published widely and received the best paper award in RTSS 2003 with Insik Shin on compositional schedulability analysis. He was a chair of the IEEE Computer Society Technical Committee on Real-Time Systems (2003-2004) and an IEEE CS Distinguished Visitor Speaker (2004-2006). He has served on many program committees and chaired several international conferences and workshops, and also on various steering committees, including the Steering Committee on CPS Week, Embedded Systems Week, and Runtime Verification. He has served on the editorial boards of several scientific journals, including the *IEEE Transactions on Computers*, the *Formal Methods in System Design*, and the *Real-Time Systems Journal*. He is a founding co-editor-in-chief of the *KIISE Journal of Computing Science and Engineering* since Sept 2007. He was a member of Technical Advisory Group (TAG) of President's Council of Advisors on Science and Technology (PCAST) Networking and Information Technology (NIT). He received the IEEE TC-RTS Technical Achievement Award in 2008. He is a fellow of the IEEE.