Generating Sound and Resource-Aware Code from Hybrid Systems Models*

Madhukar Anand, Sebastian Fischmeister, Jesung Kim, and Insup Lee

Department of Computer and Information Science
School of Engineering and Applied Sciences
University of Pennsylvania
{anandm,jesung,lee}@cis.upenn.edu, sfischme@seas.upenn.edu

Abstract. Modern real-time embedded systems are complex, distributed, feature-rich applications. Model-based development of real-time embedded systems promises to simplify and accelerate the implementation process. Although there are appropriate models to design such systems and some tools that support automatic code generation from such models, several issues related to ensuring correctness of the implementation with respect to the model remain to be addressed.

In this work, we investigate how to derive sampling rates for distributed real-time systems generated from a hybrid systems model such that there are no switching discrepancies and the resources spent in achieving this are a minimum. Of particular interest are the resulting mode switching semantics and we propose an approach to handle faulty transitions and compute execution rates for minimizing missed transitions.

1 Introduction

Modern real-time embedded systems are complex, distributed, feature-rich applications. For example a car incorporates thirty to sixty micro-controller units [1] 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 [2], Hybrid Systems [3], State-charts [4] have been successfully applied to real-time embedded systems. For embedded control software, hybrid systems are an appropriate modeling paradigm because it

* This research was supported in part by NSF CNS-0509143, NSF CNS-0720703, NSF CNS-0720518, FA9550-07-1-0216, OEAW APART-11059 and ARO W911NF-05-1-0182.

M. Broy, I.H. Krüger, and M. Meisinger (Eds.): ASWSD 2006, LNCS 4922, pp. 48–66, 2008.

can be used to specify continuous change of the system state as well as discrete transition of states [5, 6].

Although modeling and analysis play an important part in development of applications, it is also essential to establish the same guarantees in the implementation of the model. In particular, it is imperative that the correspondence between the model and the code is precisely understood. In keeping with this objective, our efforts are directed towards automatic and faithful code generation from hybrid systems models.

Introduction to CHARON. CHARON [7], is 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 constraints, 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.

Code generation from hybrid system models. A problem for code generation from verified models is to understand the relationship between the model and the code. The model's verification and analysis are only useful, if the generated code has the same properties as the model. Several code generators can derive code from a model, however, the relationship between model and the code using continuous time is not their primary concern (c.f., [4, 8] or commercial tools like Real-Time Workshop or TargetLink). On the other hand, some academic code generators ensure that the model and the code have the same properties (c.f., [9]), but the issue remains challenging.

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.

Dynamic elements in the model aggravate the problem of faithful implementation. Such dynamic elements are, for instance, battery power output, sensor quality, actuator precision, which change over time and with respect to the changing environment. Current models rely on a steady environment and resource set. Our research is motivated by the need to provide formal semantics and guarantees in dynamic environments.

1.1 Related Work and Problem Statement

Model-based automatic code generation has been an extensive research initiative in recent years, in the industry as well as academia. Commercial modeling tools such as RationalRose [10], TargetLink [11], and SIMULINK [12] 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 to discrete behavior is not addressed rigorously. Synchronous languages for reactive systems, such as STATECHARTS [4], ESTEREL [13], and LUSTRE [14], also support code generation. However, they do not support hybrid systems modeling. SHIFT [15] is a language for hybrid automata that also also supports code generation, but the focus is on dynamic networks. A complementary project is the time-triggered language Giotto that allows describing switching among task sets so that timing deadlines can be specified in a platform independent manner separately from the control code [16]. This concern is orthogonal, and in fact, CHARON can be compiled into Giotto.

Model-based development of embedded systems is also promoted by other projects with orthogonal concerns: Ptolemy supports integration of heterogeneous models of computation [8] and GME supports meta-modeling for development of domain-specific modeling languages [17]. Girard et al [18] 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 towards model-driven development of embedded software from models other than hybrid systems(c.f., [19]). In a closely related work, Stauner [20] discusses at length, the discrete refinement of hybrid automata, considering implementation effects such as sampling errors and its impact on verification.

There are several modeling tools for hybrid systems such as CHARON [7], PTOLEMY [8], SHIFT [15], the Matlab/Simulink Hybrid Toolbox [21], HYTECH [22], and d/dt [23]. However, most of them are only modeling tools and do not support automatic-code generation. A listing of many tools and their description is available at [24].

Code generation from hybrid models was introduced with focus on single-thread execution in [3]. This was extended to multi-threaded models accounting for faulty transitions in [25]. and distributed systems in [26,27]. All these previous works however, are not aware of the resources available on the implementation platform and therefore need manual parameter assignment in the code.

Contributions. In this paper, we extend previous, related work to consider the problem of calculating sampling rates based on the platform resource model. Specifically, we investigate how to derive criteria to preserve the model's switching semantics based on the platform resource model to ensure that there are no faulty or missed transitions. Our ideas are demonstrated in the context of the modeling language CHARON, where we propose an additional step in the code generation module. This additional step involves (1) specifying a platform resource model for the available hardware and its properties and (2) using

this model to compute the optimal sampling rates so that switching semantics are preserved while expending the least amount of resources in the process.

2 Basic Model and Assumptions

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 $R : \mathbb{R}^n \times P \times P \to \mathbb{R}^n$, 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 and externally updated variables. 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.

In this paper, we assume that there is a network of hybrid automata (called *agents*) communicating via a set of shared variables. We will denote a single agent 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 $\mathcal{C} = \langle (A, SV)_1, \ldots, (A, SV)_n \rangle$. We assume that every $s \in SV$ is updated by a unique agent, and it follows dynamics such that $\dot{s} \in [\mathcal{L}_1, \mathcal{L}_2], \mathcal{L}_1, \mathcal{L}_2 \in \mathbb{Q} \setminus \{0\}$. Such linear automata are of practical significance, as hybrid systems with very general dynamics can be locally approximated arbitrarily closely using rectangular dynamics [28]. The transition guards at a location are assumed to be such that at most one of them is enabled at a time.

Example 1. Consider the example of vehicle coordination where we assume that there are two vehicles. The first vehicle is the leader and follows the dynamics depicted as agent A_1 in Figure 1. x_1 denotes the distance of the leader from the baseline, v_1, its velocity. The leader's dynamics are determined by the control function u. The second vehicle trails the leader and maintains a safe distance from it. The dynamics of this vehicle is described as the 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 slows with a rate $\dot{v_2} = -1$ and if it is farther than d_{max}, it accelerates with a rate $\dot{v_2} = 1$. The invariant in the 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. It is assumed that, there is an infrastructure for communicating variables between the vehicles and that, the transmission delay is bounded and known. □

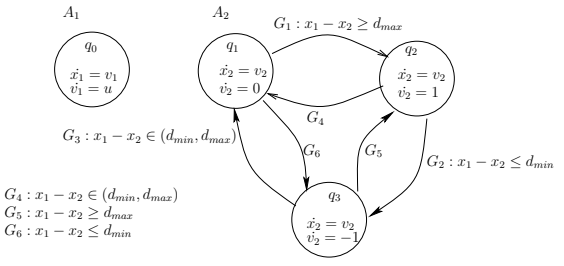

Fig. 1. A system with two agents

We now formally define the resource model and the platform on which the code will be implemented. Implementation of the continuous model involves assigning a suitable sampling rate to every agent. Such a discretization of the continuous model can be defined as,

Definition 1. *(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 $gcd(\rho_1, \ldots, \rho_n) = 1$*[1,2]*.* □

In our notation, we denote the maximum difference between the sampling rates of agents as *skew*[3].

Note that the DCHA is the model implemented on actual platforms. We will therefore, give guarantees of execution with reference to this model. For a rigorous definition of system of communicating agents and their semantics, we refer the reader to [25].When the discretized model is mapped to a real time task in the code-generation environment, each agent is assigned a period of execution. These periods of execution are assigned taking into consideration correctness guarantees and the resources available at each node. The exact procedure for assigning periods is elaborated in Sections 4.

Our objective in incorporating the resource model (i.e., model of memory, energy, CPU,etc) in addition to the hybrid model is that, we can generate a minimal sampling frequency that can be supported on the platform. This optimum is calculated by ensuring the model semantics and conserving the resources available. Therefore, we define a resource as consisting of a utilization function and a specification of energy utilization for every operation.

Definition 2. *(Resource) A resource R is defined by the tuple $\langle id, U, \mathcal{E} \rangle$, where, $id \in \mathbb{Z}^+$ is a unique identifier of the resource, U is the maximum amount of utilization, and an optional field $\mathcal{E}$ which indicates the amount of energy consumed per unit of utilization. A node N is defined as a set of interacting resources.* □

[1] Greatest common divisor.

[2] This assumption is not necessary but introduced to keep the discussion simple.

[3] We acknowledge that our definition here differs from the standard definition of skew. However, our use of the term is motivated by similar considerations.

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 finally 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 as input the agent ids and returns the bound on communication delay between two agents in $\mathcal{A}$, and ν is the baseline period, i.e., the quanta of the period of execution of any agent.*

Note that we assume a underlying reliable communication mechanism The abstractions for platform and code consider only the basic of all the actual implementation effects and make several simplifying assumptions. For instance, jitter, clock drift, message loss, and other errors in the system, which are often observed in real systems, have not been considered here. These effects can potentially weaken some of the results presented in this work. The aim of this work however, is to establish a sound theory as a first attempt at categorizing some of the implementation effects. More artifacts of the implementation can be incorporated into the model at the cost of more involved analysis.

Example 2. Consider the vehicle coordination system with two agents as shown in Figure 1. For the trailing vehicle (V_2), the resources could be a battery, the CPU and the sensor for tracking the leader(V_1). The resource model for the vehicle V_2 can thus be represented as,

V_2	id	U_{max}	$\mathcal{E}$
	$Batt_1$	1000mAh	-
	CPU	5Mhz	0.001J/op
	$Sensor_1$	1kHz	0.2J/sample

The target platform here consists of two nodes, the leader and the trailing vehicle, and if we assume no communication delays, it is described as $\langle \{V_1, V_2\}, \mathcal{M}, \phi(V_1, V_2) \rangle$, where $\mathcal{M} = \{A_1 \rightarrow V_1, A_2 \rightarrow V_2\}$, and $\phi(A_i, A_j) = 0$. The baseline period (quantum of execution) is the smallest sampling period that could be supported. For example, we could have ν to be 0.01. □

3 Code Generation from Hybrid System Models

This section gives a brief overview of the procedure of code generation from hybrid models. We first present translation of continuous behavior specified by differential equations and algebraic equations, and then explain translation of discrete actions specified by guarded transitions. Later in this section, we discuss the issue of discrepancy between the model and the generated code, real-time resource concerns and choice of correctness criteria. For more details on code generation, we refer the reader to [3, 25].

3.1 Code Generation Procedure

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 [29]). 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.

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 equations 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 triggers 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 non-deterministic 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. We have proposed a transition policy what we call *instrumentation* [25] that enforces transitions to be taken some time Δ after the transition is enabled but no later than Δ before the transition is disabled. The value of Δ is chosen such that all faulty transition possibilities are eliminated (Section 4.1). Yet another possibility is to enforce a transition once it *evaluated* to be enabled. We call such a policy an *eager* transition policy. Surely, 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 non-empty set. We only consider an eager transition policy in this paper.

3.2 Switching Discrepancies in the Code

There are a number of issues, such as ensuring the switching semantics and faithful translation of continuous dynamics, that need to be addressed to provide

guarantees in the generated code. Here, we focus 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 roundoff 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.

Example 3. (Faulty Transition) Consider the vehicle coordination system in Example 1. 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 $0.1s$ and $0.06s$, respectively. Also, let $u = 2$, $d_{min} = 0.1$, $d_{max} = 0.5$, and initial positions of vehicles be $x_1^0 = 0.3072$ and $x_2^0 = 0.2$, from the baseline, initial velocities $v_1^0 = 0, v_2^0 = 0$, the communication delay $\phi(A_1, A_2) = 0.03$, and the current states of agents be q_0 and q_2. Then, a possible run of the system is,

t	$x_1(A_1)$	$x_1(A_2)$	$x_2(A_2)$
0.06	0.3072	0.3072	0.2018
0.10	0.3172	0.3072	0.2018
0.12	0.3172	0.3072	0.2072
...			

where $x_i(A_j)$ denotes the value of variable x_i on agent A_j. Notice that 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 $0.3072 - 0.2072 = 0.0956 < 0.1$ and the system makes a faulty transition to q_3. □

Although the above example indicates a faulty transition, since the transition is made to q_3 in which the trailing vehicle decelerates, it is not critical to ensuring safety. However, in some cases, if the system makes a faulty transition to an accelerating state q_2, then, the trailing vehicles accelerates. This is critical to safety as the gap between the vehicles decreases in this case. The example below illustrates this.

Example 4. (Faulty Transition) Now consider that the relative period of update for agents A_1 and A_2 be $(2, 1)$ and the actual periods of updates be $0.2s$ and $0.1s$, respectively. Also, let $d_{min} = 0.1$, $d_{max} = 0.2$, and initial positions of vehicles be $x_1^0 = 0.19$ and $x_2^0 = 0.1$ from the baseline, initial velocities $v_1^0 = 0.1, v_2^0 = 0.2$, the communication delay $\phi(A_1, A_2) = 0.01$, and the current states of agents be q_0 and q_3. The first vehicle reverses its direction at $\dot{v_2} = -1$ at time $0.1s$. Then, a possible run of the system is,

t	$x_1(A_1)$	$x_1(A_2)$	$x_2(A_2)$
0	0.19	0.19	0.1
0.1	0.21	0.19	0.1
0.2	0.20	0.21	0.11
...			

At time 0.2, the difference between vehicles is $0.20 - 0.11 = 0.09 (< d_{max})$, but the estimated distance at A_2 is $0.21 - 0.11 = 0.1 (= d_{max})$ and the system could a faulty transition to q_2. Since q_2 is an accelerating state, this transition reduces the distance between vehicles, potentially causing a collision. □

Yet another possibility for switching errors is that of 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. We illustrate this with an example below.

Example 5. (Missed Transition) Consider the system in Example 1. Let the relative periods of execution be $(5, 3)$, the actual periods of update $(0.25s, 0.15s)$, $d_{min} = 0.25$, $d_{max} = 0.5$, the control parameter $u = 0$. $x_1 = 0.48$, $v_1 = 5$, $v_2 = 4.5$ at $t = 0.15$, and the current state of A_2 be q_2. Further, let $d = x_1 - x_2$, $\dot{d} = \dot{x_1} - \dot{x_2} \in [0.45, 0.5]$. The guard G_4 is then the condition $d \in (0.25, 0.5)$ which on instrumentation will become $d \in (0.25 + 0.1 \times 0.5, 0.5 - 0.1 \times 0.5) = (0.3, 0.45)$ as the maximum skew is 0.1, and $\mathcal{L}_2 = 0.5$. We would then have a run of the system as,

t	$x_1(A_1)$	$x_2(A_2)$
0.15	0.48	0.0
0.25	0.98	0.0
0.30	0.98	0.6862
...		

We see that the transition from q_2 to q_1 is missed here, and at time $t = 0.3s$, the system transits to q_3. □

Switching can also be affected by resource constraints and its dynamic nature. For example, as the battery wears off, it may not yield the same output causing a deadline miss of some task. If the tasks scheduled to run do not meet the deadlines, it may affect the dynamics which in turn could induce faulty transitions. To counter this, in our proposed approach, we start with an assignment of relative periods to different agents. From these relative periods, and the current estimates of resources, the actual periods of execution are synthesized. We choose these actual periods so that these correspond to the least amount of energy used while retaining the guarantees of switching behavior.

3.3 Correctness Criteria

The generated code and the model can be termed equivalent if the code exhibits a trace that is also a trace in the model. However, to account for delays in communication and skew due to different rates of execution in the code, we relax this requirement and define a relative faithful implementation. Under this relaxed form of correctness, the code exhibits a trace of the model, but the state of the model is entered at a later time. This can be captured formally as,

Definition 4. *(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 timestamps $\langle clk_0, clk_1, \ldots \rangle$, 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.*
2. *$\forall j, \exists q_{\mathcal{D}}, q_{\mathcal{K}} = q_{\mathcal{D}}, (lt_{\mathcal{D}} - lt_{\mathcal{K}}) < \phi_j(lt_{\mathcal{K}}) + \varphi(lt_{\mathcal{K}})$ where $q_{\mathcal{K}}$ is the state of the code of logical time $lt_{\mathcal{K}}$, at physical time clk, $q_{\mathcal{D}}$ is the projection of the state of the model onto the code for A_j at logical time $lt_{\mathcal{D}}$, $\phi_j = \max_i \phi(i, j)$ and φ is the maximum skew due to different rates of updates at logical time $lt_{\mathcal{K}}$.*

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

Informally, a relative faithful implementation says that (1) the error in continuous variables is bounded , and (2) the difference between the state of the model and the implementation can be off by at most the sum total of worst case communication delay and skew in update of variables. Under this relaxed scheme of things, the implementation can enter the state of the model after updates are received (which can arrive at worst $\phi_j(lt_{\mathcal{K}}) + \varphi(lt_{\mathcal{K}})$ late). We now present the framework that would help ensure that the implementation is relative faithful to the model.

4 Proposed Implementation Framework

In this section, we propose a framework for code generation with an emphasis to avoid switching discrepancies (faulty and missed transitions) and conserve resources while giving these guarantees. The Figure 2 provides an overview of how the model-driven development process in our framework with CHARON: first, the developer creates the application-specific hybrid systems model by programming agents, modes, and mode changes and by defining relative update periods. Then he specifies the platform resource model, which includes, for instance, an agent-to-node assignment, each node's hardware properties, power levels, communication delays, and agent's worst-case execution times. This resource model is then fed into a constraint solver, which computes the optimal agent's sampling rates to prevent faulty and missed transitions as described in the following sections. Note that we do assume that the base period of updates of variables in an agent (ρ) are provided beforehand, and we compute the actual period of updates (which is a multiple of base periods) depending on available resources.

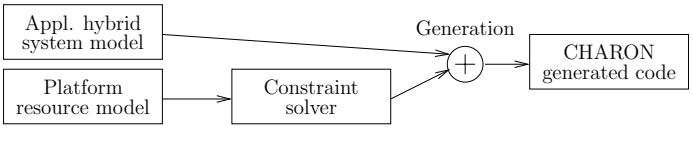

Fig. 2. Resource-aware Code-generation Framework

Before we elaborate on computing the optimal rates of sampling of agents, we highlight the solution to avoiding faulty and missed transitions.

4.1 Preventing Faulty Transitions

A faulty transition is a violation of equivalence of discrete states in a faithful implementation. It may occur due to the following reasons : 1) errors in the variables cause the guard to be evaluated true that should otherwise be false, or 2) variables are updated at different times due to scheduling and/or different update frequencies, causing the guard to be evaluated to be true. To prevent this from occurring, we have proposed a technique what we call instrumentation. The essence of that technique is to refine the model by tightening transition conditions according to the maximum errors due to numerical and different sampling rates. The approach enforces that the transitions in the code are consistent with the model.

Errors in variables could be due to roundoff, truncation or be timing-induced due to the different rates of execution of the agents. Roundoff and truncation errors are assumed to be given, the communication delay is obtained by monitoring and the maximum *skew*, denoted by φ due to dissimilar periods can be computed by, $\varphi(A_i, A_j)_{max} = \max_{n\in[1..N]} \left(nh_j - \left\lfloor \frac{nh_j - \phi(A_i,A_j)}{h_i} \right\rfloor h_i\right)$ where $N = \frac{LCM(h_i,h_j)}{h_j}$[4], h_i and h_j are the step sizes in the sampling of Agent A_i and A_j, respectively.

Definition 5. *(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 $\phi(A_i, A_j)$ compute the skew and delay between the agents, then, the instrumented guards and invariants are given by,*

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

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

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 roundoff and truncation error in x_i in the state p. □

We now illustrate how instrumentation prevents faulty transitions with the following example.

Example 6. Consider the system in Example 3 and the time interval under consideration be [0,1]. If we denote $d = x_1 - x_2$, then, $\dot{d} = \dot{x_1} - \dot{x_2} = 2t - t = t$. Since $t \in [0.05, 1]$, we can say that $\dot{d} \in [0.05, 1]$. Now, given that $\phi(A_1, A_2) = 0.03$, and

[4] Least common multiple.

the skew at $t = 0.12$ is 0.02, and assuming the bound on roundoff and truncation errors is 0.001, the transition guard, $x_1 - x_2 \leq 0.1$ upon instrumentation becomes $x_1 - x_2 \leq (0.1 - 0.001 - 1 \cdot (0.02 + 0.03)) = x_1 - x_2 \leq 0.049$. Therefore, the faulty transition at $t = 0.12$ can be prevented. □

The theorem below formally states that instrumentation prevents faulty transitions. For a sketch of the proof, we refer the reader to [25].

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 every guard (in $G_{A_j}(e)$) that evaluates to true is instrumented as given in Definition (5) then there will be no faulty transitions.* □

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

As a final note in this section, we add that while instrumentation reduces the guard set, it does not affect switching in any other way. In particular, if the original interval was such that only one transition was enabled from the location at any time, the property remains valid with instrumented guards as well as it is a subset of the original interval.

4.2 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. 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. *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 T_{x_i} are defined as,*

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

$k = 1, 2$, then, the transition will not be missed if,

$$\left\| \bigcap_i \left(\bigcap_{k=1,2} T_{x_i}(k) \right) \right\| \geq 2h_j \tag{3}$$

where $\delta_{min} = \varphi_{min} + \phi(A_i, A_j)$, $\delta_{max} = \varphi_{max} + \phi(A_i, A_j)$ *between agents* A_i *and* A_j, *and* $\dot{x_i} \in [\mathcal{L}_{1_{x_i}}, \mathcal{L}_{2_{x_i}}]$, *then, the transition will be detected and will not be missed if they are taken as soon as enabled.*

Proof. (sketch) We proceed to sketch the proof of the theorem in two parts. First, we will derive a condition on the overlap of guard and invariant that will allow us to detect the enabling of the transition. Then, given that the guard is of the form $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$, we will derive a sufficient condition to meet this overlap, based on the periods of execution of agents.

To prove the first statement, assume that we are given a task-period set $\Omega = \{(\tau_i, h_i)\} 1 \leq i \leq n$. Each task τ_i will be 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, we speak of time 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 we assume eager switching, 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 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, we can claim that it should stay enabled in the code for $2h_i$ time units so that the transition is not missed. This is illustrated in Figure 3. Now, consider the guard set $g = \bigwedge_i x_i \in [l_{x_i}, u_{x_i}]$. Let the current

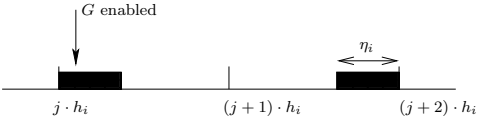

Fig. 3. Worst case scenario

logical time be lt and current values of variables at agent A_j given by $x_i(lt)$. We will 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 $T_2 = [lt + \frac{l_{x_i} - x_i(lt)}{\mathcal{L}_{2_{x_i}}} + \delta_{max}, lt + \frac{u_{x_i} - x_i(lt)}{\mathcal{L}_{2_{x_i}}} + \delta_{min}]$, 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_i's, we can find the time interval when the guard will definitely be true.

From the above arguments, we can conclude that a Condition (3) 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 and the sufficient condition is not met.

Example 7. Consider the case of Example 5. As a quick check, we find that 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, $T_1 = (\frac{0.48-0.45}{0.45} + 0.1, \frac{0.48-0.3}{0.45} + 0.05) = (0.167, 0.45)$. We find that $\|T_1 \cap T_2\| = 0.243 \not\geq 2(0.15)$ does not satisfy the sufficient condition for preventing missed transitions. However, if we choose the period of execution to be 0.12, we can see that the transition will not be missed. □

With the Theorems 1 and 2, we have a sufficient condition to ensure a relative faithful implementation that we record in the following corollary.

Corollary 1. *Let the code $\mathcal{K}$ of the model $\mathcal{D}$ be implemented on a distributed platform. If the code for every agent A_i every $G \in G_{A_j}$ is dynamically instrumented so that G and corresponding invariant I 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 $\mathcal{D}$.* □

4.3 Minimal Periods of Execution

In this section, we describe an algorithm to choose minimal periods of execution to avoid missing a transition and meeting the resource constraints. The main idea of the approach described as Algorithm 1 is to scale the relative periods of execution so that they meet the supported level of utilization. Specifically, as we are interested in finding the minimal periods of execution, we start with the smallest possible assignment and keep incrementing the periods till the supported level of utilization is met. Note that our algorithm needs to be run every time the available resource changes. We assume that the run time is instrumented to take this into consideration and call the procedure appropriately.

This is implemented in the function *SMALLEST-K*. Here we consider schedulability under EDF and Rate Monotonic (RM) algorithms. The function takes as input α, that is the level of utilization permissible with the supported levels of energy, and returns the smallest multiple of the base period of update k for which all the agents mapped onto a particular node (N) can be scheduled. We assume that voltage scaling techniques (c.f., [30]) can be used to fix a level of utilization of the CPU. In addition to checking schedulability, we assume that a function *RESOURCE-CHECK* is implemented that checks to see if agents are scheduled with a particular period and other resource constraints. For example, the function could check to see, if the frequency of reading of sensor data

is less than the maximum permissible sampling frequency of the sensor. If an energy budget is associated, then it can be used to check whether the budget is met. If a particular k does not satisfy schedulability or resource constraints, it is incremented and then tested again. Note that increased k results in longer periods of execution. In the algorithm, W_j and ρ_j denote the maximum execution requirement and the sampling rate of Agent A_j which we assume are fixed.

Algorithm 1. Algorithm to find periods of execution of agents.

SMALLEST-K (α,N):
1: $k \leftarrow 1$
2: CASE-EDF:
3: **while** $\left(\left(\sum_{\mathcal{M}(j)=N} \frac{W_j}{k \cdot \rho_j} \not\leq \alpha\right) \vee (\text{RESOURCE-CHECK(N)} \neq \text{true})\right)$ **do**
4: $k \leftarrow k + 1$
5: **end while**
6: CASE-RM:
7: $J = \{j_1, \ldots j_n | \mathcal{M}(j_i) = N\}$
8: **while** $\left(\left(\forall j \in J, \sum_j \lceil \frac{\rho_j}{\rho_1} \rceil \cdot W_j \not\leq \alpha \cdot k \cdot \rho_j\right) \vee (\text{RESOURCE-CHK(N)} \neq \text{true})\right)$ **do**
9: $k \leftarrow k + 1$
10: **end while**
11: return k

SELECT-PERIODS-NODE (N):
1: $k_{N_{max}} \leftarrow$ SMALLEST-K(α_{min})
2: $k_{N_{min}} \leftarrow$ SMALLEST-K(α_{max})
3: return $(k_{N_{min}}, k_{N_{max}})$

SELECT-PERIODS ($\langle p_1, \ldots, p_n \rangle$):
1: $(k_{min}, k_{max}) \leftarrow (0, 0)$
2: **for** $N \in \mathcal{N}$ **do**
3: $(k_1, k_2) \leftarrow$ SELECT-PERIODS-NODE(N)
4: $(k_{min}, k_{max}) \leftarrow (\max(k_{min}, k_1), \max(k_{max}, k_2))$
5: **end for**
6: $k \leftarrow k_{max}$
7: **while** $k \geq k_{min}$ **do**
8: **if** (CHECK-MISSED($\langle p_1, \ldots, p_n \rangle, k$)) **then**
9: return k
10: **end if**
11: $k \leftarrow k - 1$
12: **end while**

The function *SELECT-PERIODS-NODE* returns the maximum and minimum possible utilization and returns the range of scaling factor k possible on that node. The $K_{N_{min}}$ corresponds to the smallest periods possible on the node N with the supported amount of resources on node N. The *SELECT-PERIODS* function takes as input the present set of states $\langle p_1, \ldots, p_n \rangle$ and computes the possible values of k for every node and computes the range of k's possible for all

the nodes. This range is represented by (k_{min}, k_{max}). To find the minimal value of k, we start iterating from k_{max} since it represents the least utilization level. At each iteration, we check to see whether choosing that value of k would result in a missed transition. The function *CHECK-MISSED* implements this check. Thus, at the end of the while loop (Steps 7-12), we would have found a k which can be supported on all nodes while being guaranteed for no missed transitions. Once we have found the value of k, we can supply the parameters to the code.

Example 8. (Room Heater) Our example for illustrating the algorithm is adapted from the heater benchmark for hybrid systems verification [31]. The benchmarks considers the case of a set of rooms being heated by limited number of heaters that are shared by the rooms. The number of heaters is strictly less than the number of rooms. In our example, we consider two rooms and one heater. The model of this system, described in Figure 4 consists of two thermostats and a heater. The temperature in a room is assumed to vary as, $\dot{x_i} = c_i h_i + b_i(u - x_i), i = 1, 2$ where h_i is 1 if the heater is in the room, otherwise 0, u is the outside temperature, and c_i, and b_i are constants. The heater model is a pure switched system. If $(x_i \leq get_i) \wedge (x_j - x_i \geq dif_i)$, then the heater is moved from room j to room i, where $i = 1, 2; j = 2/i$.

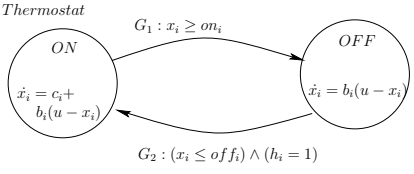

Fig. 4. The hybrid system model of the thermostat

The system is implemented on two nodes. There are two agents, one to check whether the heater has to be moved (A_1), and the other for switching on or switching off the heaters(A_2). The controller in the room with the heater runs both of them, and the controller in the other room runs only the second agent.

Let us assume that the relative periods of the two agents are $(3, 1)$ and the relative worst case execution times be $(2, 1)$. Let us also assume that the levels of utilization are 0.25 and 0.5. In the room with the heater (say room 1), the controller has to schedule both the agents so, we have $k_{1_{min}}$ is such that $\frac{1}{k_{1_{max}}}(\frac{2}{3} + \frac{1}{1}) \leq 0.5$ which yields $k_{1_{min}} = 4$. Similarly with utilization 0.25, we can get $k_{1_{max}} = 7$. In room 2, since there is only one agent to be scheduled, we have, $k_{2_{min}}$ is such that $\frac{1}{k_{1_{max}}}(\frac{1}{1}) \leq 0.5$ which yields $k_{2_{min}} = 2$. Similarly with utilization 0.25, we can get $k_{2_{max}} = 4$. Therefore, after taking the maximum over both nodes, we get $(k_{min}, k_{max}) = (4, 7)$. The agent A_2 in room 1 is waiting on transition G_1 and in room 2 is waiting on transition G_2. It can be seen that $k = 7$ that corresponds to utilization 0.25, indeed satisfies the sufficient condition for no missed transitions. □

5 Conclusions and Future Work

We have proposed a framework for generating resource-aware code from hybrid systems models with guarantees of no switching discrepancies. Our approach is an effort to bridge the semantic gap between the model and the code due to discretization and resource constraints. We accomplish this by incorporating a resource model of the target platform in addition to the application model and generating parameterized code from this model. The parameters are supplied at runtime by monitoring the state of the resources and checking for missed transitions.

There are potentially many directions of future work. We hope to complete the implementation of the framework. In the paper, we have largely focused on power and CPU as the main resources. We would like to extend it to more comprehensive set of resources. Also, in the present scheme of things, a change in resource levels or transition on any agent can trigger a recalculation of the periods of all the agents. This is so because of the assumption that all the agents have relative periods of execution. An alternative, would be to start with constraints on periods, such as $\rho_1 \leq 2\rho_2$. This way, we would only need to recompute the periods whenever the constraints are about to be violated. Another possible extension to the framework, would be to mask faults and failures or consider graceful degradation by viewing it as an extreme case of resource dynamism. Finally, we hope to use ideas from runtime monitoring [32] to monitor and steer the system towards desirable behavior.

Acknowledgments. We would like to thank anonymous referees for their suggestions in improving this paper.

References

1. Martin, N.: Lock who's talking: Motorola's c.d. team. LockSmart Online Article (1998)
2. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Scienc 126, 183–235 (1994)
3. Alur, R., Ivančić, F., Kim, J., Lee, I., Sokolsky, O.: Generating embedded softwarefrom hierarchial hybrid models. In: Proceedings of LCTES (2003)
4. Harel, D.: Statecharts: A visual formalism for complex systems. Science of ComputerProgramming 8, 231–274 (1987)
5. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Comp. Science 138, 3–34 (1995)
6. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Real-Time:Theory in Practice, REX Workshop. LNCS, vol. 600, Springer-Verlag, Heidelberg (1991)
7. Alur, R., Grosu, R., Hur, Y., Kumar, V., Lee, I.: Modular specification of hybridsystems in CHARON. In: HSCC, pp. 6–19 (2000)
8. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Luvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity–the Ptolemy approach. Proceedings of the IEEE 91, 127–144 (2003)

9. Alur, R., Grosu, R., Hur, Y., Kumar, V., Lee, I.: Charon: a language for modular specification of multi-agent hybrid systems. Technical Report MS-CIS-00-01, Dept. of Computer and Information Science, University of Pennsylvania (2000)
10. RationalRose, http://www-306.ibm.com/software/awdtools/developer/rose/
11. TargetLink, http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm
12. Simulink, http://www.mathworks.com/products/simulink/
13. Berry, G., Gonthier, G.: The synchronous programming language esterel: design,semantics, implementation. Technical Report 842, INRIA (1988)
14. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language Lustre. Proceedings of the IEEE 79, 1305–1320 (1991)
15. Deshpande, A., Göllu, A., Varaiya, P.: SHIFT: a formalism and a programming language for dynamic networks of hybrid automata. In: HS 1997. LNCS, vol. 1567, Springer, Heidelberg (1996)
16. Henzinger, T., Kirsch, C., Sanvido, M., Pree, W.: From control models to real-time code using Giotto. IEEE Control Systems Magazine (2003)
17. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. In: Proceedings of the IEEE, vol. 91, pp. 145–164 (2003)
18. Model-Driven Hybrid and Embedded Software for Automotive Applications. In: 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES 2004) (2004)
19. Shah, B., Dennison, R., Gray, J.: A model-driven approach for generating embeddedrobot navigation control software. In: ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference, pp. 332–335. ACM Press, New York (2004)
20. Stauner, T.: Discrete-Time Refinement of Hybrid Automata. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 407–420. Springer, Heidelberg (2002)
21. Hybrid Toolbox - Hybrid Systems, Control, Optimization, http://www.dii.unisi.it/hybrid/toolbox
22. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HYTECH: A model checker for hybridsystems. International Journal on Software Tools for Technology Transfer 1, 110–122 (1997)
23. Asarin, E., Dang, T., Maler, O.: The d/dt Tool for Verification of Hybrid Systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 365–370. Springer, Heidelberg (2002)
24. Tools, H.S.: http://wiki.grasp.upenn.edu/graspdoc/hst/
25. Hur, Y., Kim, J., Lee, I., Choi, J.Y.: Sound Code Generation from Communicating Hybrid Models. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 432–447. Springer, Heidelberg (2004)
26. Anand, M., Kim, J., Lee, I.: Code generation from hybrid systems models for distributed embedded systems. In: Proceedings of the IEEE ISORC, pp. 166–173 (2005)
27. Anand, M., Fischmeister, S., Kim, J., Lee, I.: Distributed-code generation from hybrid systems models for time-delayed multirate systems. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 210–213. ACM Press, New York (2005)
28. Henzinger, T.A., Ho, P.H.: Algorithmic analysis of nonlinear hybrid systems. In: Wolper, P. (ed.) Proceedings of the 7th International Conference On Computer Aided Verification, Liege, Belgium, vol. 939, pp. 225–238. Springer, Heidelberg (1995)

29. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: the Art of Scientific Computing, 2nd edn. Cambridge University Press, Cambridge (1999)
30. Pillai, P., Shin, K.: Real-time dynamic voltage scaling for low-power embedded operating systems. In: Proceedings of the 18th Symposium on Operating Systems Principles SOSP 2001 (2001)
31. Fehnker, A., Ivancic, F.: Benchmarks for Hybrid Systems Verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
32. Tan, L., Kim, J., Lee, I.: Testing and Monitoring Model-based Generated Program. In: Proceeding of Runtime Verification Workshop (RV 2003), Boulder, Colorado (2003)