Code Generation from Hybrid Systems Models for Distributed Embedded Systems [*]

Madhukar Anand, Jesung Kim, and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{anandm,jesung,lee}@saul.cis.upenn.edu

Abstract

Code generation from hybrid system models is a promising approach to producing reliable embedded systems. This approach presents new challenges as the precise semantics of the model are hard to capture in the code. A framework for generating code was introduced for single threaded/processor environments . Here, we extend it by considering code generation for distributed environments. We also define criteria for faithful implementation of the model. To this end, we define faulty *and* missed *transitions. For preventing faulty transitions, we build on the idea of instrumentation we have developed for sound simulation of hybrid systems. Finally, we present sufficient conditions to avoid missed transitions and provide examples.*

1 Introduction

A model-based approach is an emerging paradigm for developing robust embedded software, and has been the focus of increasing research effort. In this approach, models are used during the design phase to ensure that systems under consideration have desired properties. A software implementation then can be produced automatically by compiling the model in many cases. This enhances the quality of the end product significantly since it can eliminate time-consuming and error-prone manual programming. Even though manual programming may give a better opportunity of fine tuning for performance optimization, the advantage is easily offset in embedded computer systems where proven safety is utmost important.

An embedded system typically consists of a collection of digital programs that interact with each other and with an analog environment. For such systems, *hybrid systems* [1, 15] is an appropriate modeling paradigm. In hybrid systems, the state can be changed continuously as well as discretely. In this model, continuous state evolution is specified by differential equations while discrete state changes are modeled by the legacy finite state machine. Typically, the code generator translates a hybrid systems model into a set of functions that can be invoked periodically by the underlying RTOS 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.

[*]This research was supported in part by NSF CCR-0086147, NSF CCR-0209024, and ARO DAAD19-01-1-0473.

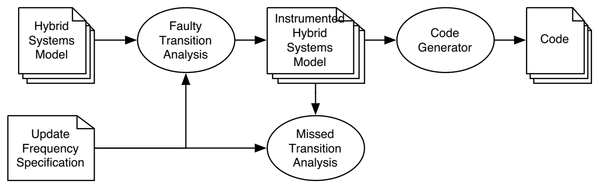
Hybrid Systems Model
Faulty Transition Analysis
Instrumented Hybrid Systems Model
Code Generator
Code
Update Frequency Specification
Missed Transition Analysis

Figure 1. Work flow of our framework.

This paper proposes a framework for automatic code generation and validation for hybrid systems models to distributed execution environment. Our framework combines and extends previously proposed techniques [5, 12].

In our framework, the code is generated and validated against hybrid systems models in three steps as illustrated in Figure 1. 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

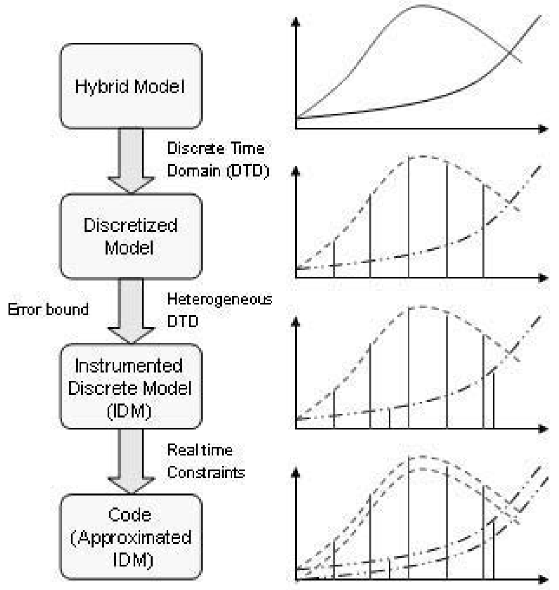

Figure 2. Design Flow

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 in to the code generator to produce the code.

The work flow described in the Figure 1 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. The semantic relationship is depicted in Figure 2. At each stage, 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.

Related works. Commercial modeling tools such as SIMULINK 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. That is, an estimation of local numerical errors are checked against the user-given tolerance represented in the form of relative errors and absolute errors, but the effect of such errors on discrete behavior is not addressed rigorously. Therefore, even if the simulation and/or automatically generated code satisfies local error tolerance, it is generally not guaranteed that the behavior is consistent to the mathematical model.

Synchronous languages for reactive systems, such as STATECHARTS [11], ESTEREL [7], and LUSTRE [10], also support code generation. However, these languages do not support specification of continuous activities, and thus the semantic difference between the model and the code is not a main issue. SHIFT is a language for dynamic networks of hybrid automata [8], and it also supports code generation, but the focus is not on correctness issues. Model-based development of embedded systems is also promoted by other projects with orthogonal concerns: Ptolemy supports integration of heterogeneous models of computation [9] and GME supports meta-modeling for development of domain-specific modeling languages [13].

In [5], code generation from hybrid models was introduced with focus on single-thread execution. This was extended to multithreaded models accounting for faulty transitions in [12]. Here we extend the previous work by first identifying criteria for faithful implementation of the model. Then, we analyze both faulty and missed transitions for the case of distributed execution and propose necessary extensions: For the case of faulty transitions, we eliminate the necessity of EDF scheduling when the guard sets are disjoint. For missed transitions, we generalize the theorem introduced in [5] for the case of different step sizes/multithreaded execution.

Organization of the paper. The remainder of this paper is organized as follows. The next section gives an overview of hybrid systems models along with an introduction of our hybrid systems modeling language CHARON. We then describe the code generation procedure and related correctness issues. In the following, we describe our approach to validating the generated code against the model. The final section gives a concluding remark and suggests future works.

2 Hybrid Systems Model

Hybrid systems is a formal model that combines continuous dynamics specified in differential equations and finite state machine based discrete control. Formally, a hybrid model consists of a real vector $x = (x_1, x_2, \ldots, x_n)$ 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 the 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

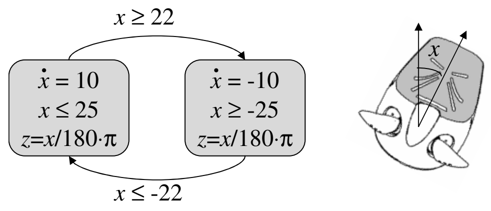

Figure 3. Hybrid model for a robot dog.

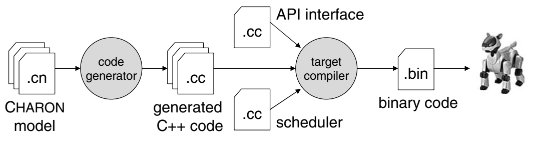

Figure 4. Code generation process.

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 $\dot{x} = f_p(x)$).A transition out of the current discrete state should be taken before the continuous state goes out of the invariant set.

For example, Figure 3 shows a simple hybrid model for a robot dog panning its head. It consists of two positions or discrete states, each of which specifies constant increase/decrease ($\pm 10°/s$) of variable x, which represents the angular position of the head. Transitions cause the direction of the movement of the head to be reversed by switching the position (and hence dynamics) when the head is moved beyond a certain degree ($\pm 22°$). The transitions can be taken any time while the guard is true (i.e., the time when the transition is taken can be non-deterministic). The invariant of each location specifies that the switch should occur before the head moves beyond its allowed range ($x \leq 25$ and $x \geq -25$).

We have been developing the modeling language CHARON, a design environment for specification and analysis of hybrid systems [2]. As a language, CHARON has many object-oriented features to aid design of complex hybrid systems. In CHARON, the building block for describing the system architecture is an *agent* that communicates with its environment via shared variables. The language supports the operations of *composition* of agents to model concurrency, *hiding* of variables to restrict sharing of information, and *instantiation* of agents to support reuse. The building block for describing flow of control inside an atomic agent is a *mode*. A mode is basically a hierarchical state machine, that is, a mode can have sub-modes and transitions connecting them. Variables can be declared locally inside any mode with the standard scoping rules for visibility. Modes can be connected to each other only via well-defined entry and exit points. We allow *sharing* of modes so that the same mode definition can be instantiated in multiple contexts. Discrete updates in CHARON are specified by *guarded actions* labeling transitions connecting the modes. Some of the variables in CHARON can be declared *analog*, and they flow continuously during continuous updates that model passage of time. The evolution of analog variables can be constrained in three ways: differential constraints, algebraic constraints, and invariants which limit the allowed durations of flows. CHARON supports compositional trace semantics for both modes and agents [4]. For analysis it supports simulation, and formal verification of safety properties for a restricted subset, namely, models with finite discrete state and linear continuous dynamics in every mode [2, 3].

Figure 4 illustrates the code generation process in CHARON. The code generator translates high-level models written in CHARON into C++ programs as we will review in the next section. The generated programs are compiled by the target compiler along with support programs that handle the platform dependency such as device access and task scheduling. See [5, 14] for details.

3 Code Generation

This section gives a brief overview of the procedure of code generation from hybrid models. See [5, 14] for details. 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.

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 [16]). 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.

As explained in the previous section, the guard 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. This introduces various transition policies, as illustrated in Figure 5. 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* [12] 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 not to take a transition. Indeed, this policy makes sense when there will be a chance of taking another transition before the invariant is violated. Conversely, a policy that enforces a transition once it is 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.

We translate a transition into an if-then statement where the guard becomes the if-condition and the statement block contains the assignments, essentially implementing the urgent transition policy. The then-block also contains an additional statement that updates an internal variable storing the current position. If the user specifies a constant Δ for instrumentation, the if-block is modified such that the condition becomes true at time Δ after the original condition becomes true. Assuming that the guard sets are rectangular as in Figure 5, that is, the predicates are of the form $x \sim c$ where c is a constant and $\sim$ is one of $\leq$, $<$, $>$, and $\geq$, the modification is simple. For example, if the guard $10 \leq x \leq 20$ with the dynamics $\dot{x} = 2$ is instrumented by Δ, it becomes $10 + 2\Delta \leq x \leq 20 - 2\Delta$. Note that the instrumented guard is always a subset of the original guard. Note also that the guard set may be so small that it becomes an empty set after instrumentation. In this case, the transition is permanently disabled, and the eager transition policy is violated. We will address this issue in Section 4.2.

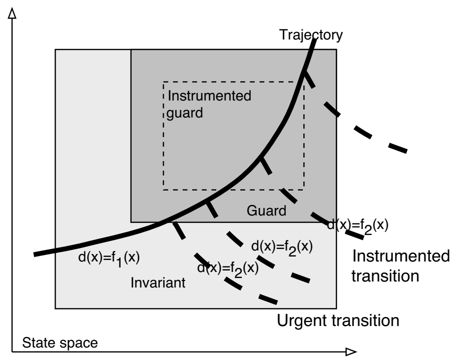

Figure 5. Transition policies.

Conceptually, the if-block should be executed continuously (i.e., infinitely frequently), since continuous variables can be updated any time. In the generated code, however, variables are updated synchronous to execution of differential equations and algebraic equations. Therefore, the code for transitions is executed only once after continuous actions are performed at every step. However, this strategy is not safe in a distributed / multithreaded execution environment where shared variables can be updated asynchronous to the testing of the guards.

3.2 Correctness of the Generated Code

The behavioral semantics of hybrid systems is defined in the continuous time domain as explained in the previous section, and thus it is generally not possible for the generated code to capture the semantics precisely. That is, hybrid systems models specify continuous change of variables, whereas the code can update variables only discretely. Moreover, even if we take discrete update for granted, computation of the values of variables are subject to errors because (1) the code normally relies on numerical algorithms and floating-point operations to solve differential equations, and (2) variables may be updated at different times due to scheduling decision and different update frequency. Thus, it is possible that the code, even if it is generated automatically from the model, exhibits a behavior that is not expected from the model.

For example, consider a hybrid model shown in Figure 6. The model specifies a simple behavior of a robot dog turning its head towards the left and upwards, and conditionally turning on the LED on the face of the robot. The model consists of three positions and three variables. Variables θ

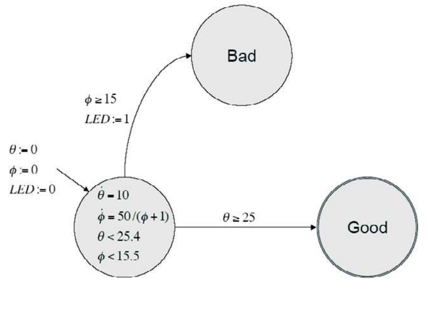

Figure 6. Faulty transitions due to numerical errors

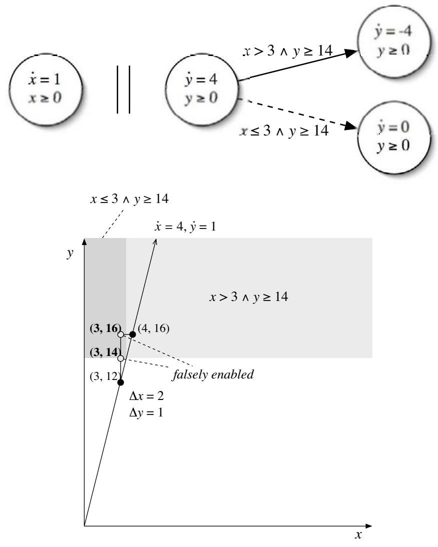

Figure 7. Faulty transition due to timing errors.

and ϕ represent the tilt of the head of a robot towards the left and upwards, respectively, and the variable *LED* represents the status of the LED. At the initial position, variables ϕ and θ evolve continuously according to the differential equations $\dot{\theta} = 10$ and $\dot{\phi} = 50/(\phi + 1)$, making the head moves towards the left and upwards with the rate given by the expression at the right-hand side of the equations (i.e., $d\theta/dt = 10$ and $d\phi/dt = 50/(\phi(t) + 1)$). As time elapses, the condition associated with each transition becomes true and a transition will be taken either to "good" or "bad", and stays there forever. Specifically, transition to "good" is enabled at time $t = 2.5$, while transition to "bad" is enabled later at time $t = 2.55$.[1] Since the invariant condition $\theta < 25.4$ specifies that control cannot stay at the initial position beyond $t \geq 2.54$, transition to "bad" should never occur. Unfortunately, if we translate the model to code using a numerical method that approximates a differential equation in a discrete fashion, it is possible that the transition to "bad" is detected enabled before the time when the transition to "good" is enabled due to numerical errors. For example, if the code employs Euler's method with the step size of 0.032, transition to "bad" is detected enabled at time $t = 2.496$, when the transition to "good" is not enabled yet. Hence, it is possible that automatically generated code takes the wrong transition and turns on the LED.

Figure 7 shows another example of wrong transitions. In this example, the model consists of two parallel components. Unlike the previous example, the dynamics is constant, and hence numerical errors are no longer a concern. However, a faulty transition may occur when each component is translated into code separately and executed concurrently, possibly with different step sizes. For example, if we assign step sizes $h_x = 1$ and $h_y = 2$, the transition is enabled in the code when $(x, y) = (3, 14)$, while the transition is never enabled in the model. Even if we assign the same step size $h_x = 2$ and $h_y = 2$, a faulty transition may occur when $(x, y) = (3, 16)$.

[1]($\dot{\phi} = 50/(\phi + 1)$, $\phi_0 = 0$ is equivalent to $\phi = \sqrt{100t + 1} - 1$, and $\sqrt{100t + 1} - 1 = 15$ gives $t = 2.55$).

Thus, we need a new criterion to validate the code against the model from which the code originates. In particular, we need to allow a certain degree of errors in variables, yet we may want to enforce discrete states to be correct.

Definition 1. *(Faithful Implementation) We define that the code is a* faithful implementation *of the model if there exists a constant error bound on the variables in the code and the discrete states are equivalent modulo the timing of execution in the code.* □

We have given a informal definition here. For a formal treatment of the system including definitions and proofs of theorems, we refer the reader to [6].

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.

- Errors in variables may cause a transition that should otherwise not be taken.

- Variables may be updated at different times due to different update frequency and scheduling decision, and may cause a transition that should otherwise not be taken.
- 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 the other issues in the previous works, but not in the same framework.

4 Validation of the Generated Code

4.1 Model assumptions

In this paper, we limit our attention to the case of the model with constant dynamics i.e., hybrid systems where the right-hand side of every differential equation is a constant. This is needed to avoid errors in numerical integration and it also makes instrumentation analysis easier. We consider a distributed system of agents having different sampling frequencies. We also assume that the system free from clock drift. Further, the times for sensing and communication have been assumed to be small enough to be included in the execution time. Each of the guard sets is assumed to be a rectangular set. This makes the instrumentation easy.We also assume that the guard sets are disjoint with each other. The reason is twofold: First, we can relax the need for an EDF like execution with this assumption [12] since at most one agent has an enabled transition at any moment. Secondly, within the same agent, the disjoint condition prevents non-deterministic choice, since at most one transition is enabled at any position.

4.2 Preventing faulty transition

A faulty transition is a transition that has occurred in the code, but there is no corresponding transition in the model.

Definition 2. *(Faulty Transition) Let $p_0, \dots p_n$ be a sequence of positions in the trace of the code. We say that (p_{n-1}, p_n) is a faulty transition if $p_0, p_1, ..., p_{n-1}$ is a valid sequence of positions in the model, but $p_0, p_1, ..., p_n$ is not.* □

A faulty transition is a violation of equivalency of discrete states in a faithful implementation. It may occur due to the following reasons.

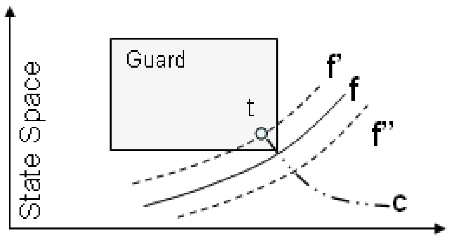

Figure 8. Faulty transitions due to error in variables.

1. Errors in the variables cause the guard to be evaluated true that should otherwise be false.
2. Variables are updated at different times due to scheduling and/or different update frequencies, causing the guard to be evaluated to be true.

Figure 8 illustrates faulty transitions due to error in variables. The dashed curve above ($f^{'}$) and below ($f^{''}$) represent the upper and lower bounds on the error. At time t, the guard is enabled as per the value specified by $f^{'}$ and not enabled as per f and a transition c results. Clearly, this transition is a faulty transition, since such a transition is not possible in the model.. These kinds of faulty transitions can be avoided if we limit the class of hybrid models to the one explained in Section 4.1.

Figure 7 in Section 3 presented a case of faulty transitions due to timing. To prevent the this from occurring, we have proposed a technique what we call *instrumentation* [12]. 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 a strict execution order to ensure that transitions occur in an order consistent to the model. While the execution order constraint can be relatively easily satisfied by using the EDF scheduler in a centralized execution environment, in a distributed execution environment, it incurs severe overheads because it requires synchronization operation to enforce execution order between tasks running on different machines.

To use the technique in a distributed system, we propose to analyze the model before code generation if execution order needs to be strictly enforced. The key idea is that execution order need not be enforced to when there is no transition between the time when a variable is written and the time when the variable is read. This condition is always true when the guard sets are disjoint with each other. This ensures that, while a guard is being true, transitions that would reset the variables involved in the guard will not be taken. Thus, in this case, tasks can be executed asynchro-

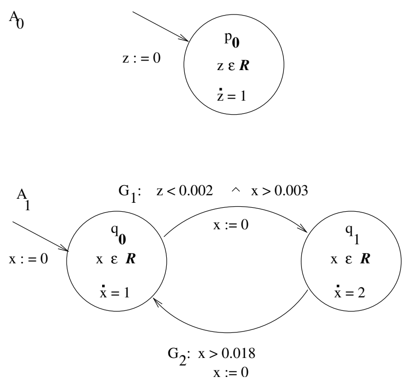

Figure 9. Missed transition Example.

nously yet still free from faulty transition if the guards are instrumented, because it is guaranteed that there will be no transition between the time that a variable is written and the time that a variable is read.

Theorem 1. *Consider a distributed system of agents. Let the guard set of every agent be disjoint. Further, let the guard sets be instrumented with δ where δ is the upper bound on numerical and synchronization errors. Then, the instrumented guard sets will ensure that there will be no faulty transitions.* □

4.3 Preventing Missed Transition

A *faithful* implementation of the code must not only consider faulty transitions but also the possibility of *missed* transitions. A missed transition is a transition that must be taken to satisfy the invariant, but is not taken by the code.

Definition 3. *(Missed Transition) Let $p_0, \dots p_n$ 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. We say that there is a* missed transition *at p_{n-1}, if $p_0, p_1, ..., p_{n-1}$ is a valid sequence of positions in the model, but $p_0, p_1, ..., p_n$ is not.* □

A missed transition occurs due to the following reasons:

- The guard is not evaluated sufficiently frequently.
- Evaluation of the guard may be scheduled before an update of a variable that enables the transition and after an update of a variable that disables the transition, and thus misses the chance to detect that the transition has been enabled.

Now, let us consider a numerical example presenting each of the above possibilities.

Example 1: Consider a system consisting of two agents A_0 and A_2 as shown in Figure 9. Let the sampling frequency A_0 be 0.001 and that of A_1 be 0.003. Let us denote the current position of agent A as P_A.Then a possible run is:

t	z	P_{A_0}	x	P_{A_1}
0.001	0.001	p_0	0.000	q_0
0.002	**0.002**	p_0	**0.000**	q_0
0.003	0.003	p_0	0.009	q_0
0.004	0.004	p_0	0.009	q_0

Clearly, if the guard in A_1 was sampled more often, then at t = 0.002, it would have made a transition to state q_1. This transition is missed because of insufficient sampling.

The following example shows that missed transitions can also occur due to scheduling decisions.

Example 2: Now consider the same system as in Figure 9 but also consider the effect of scheduling. Let the sampling frequency of A_0 be 0.001 and that of A_1 be 0.002. Now, with scheduling considerations, let us assume that A_1 is always scheduled ahead of A_0. We would hava run as:

t	z	P_{A_0}	x	P_{A_1}
0.001	0.001	p_0	0.000	q_0
0.002	**0.002**	p_0	**0.006**	q_1
0.003	0.003	p_0	0.006	q_0
0.004	0.004	p_0	0.012	q_0

However, if we reverse the order, i.e., assume that A_0 is always scheduled ahead of A_1 then, it can be easily seen that the transition will be missed. Therefore,even though the guard is tested sufficiently frequently, the choice of scheduling could result in missed transitions.

We have identified a sufficient condition to prevent the former case [5] when the code is executed with a single frequency and with a single thread. In this paper, we generalize the idea to the case with different frequencies and with multiple threads possibly on distributed systems.. The previous result indicates that the absence of missed transition is assured if the guard and the invariant overlaps at least time h, where h is the step size (i.e., inverse of the frequency). In the case where variables have different sampling frequencies, the following theorem gives us a sufficient condition for no missed transitions.

Theorem 2. *Consider a distributed system of agents with an eager policy on transitions. If the guards are disjoint, and the guard and the invariant overlap by at least $2(h + h')$, where h is the frequency of evaluation of the guard and h' is the frequency of updates of the variables in the guard, then, there will be no missed transitions.*

Proof. (*sketch*) The evaluation of the guard might be scheduled at some time $jh, j \in \mathbb{Z}^+$ but a guard may be enabled immediately after that i.e., at time $jh+\epsilon, \epsilon > 0$. This will be detected in the code during the next evaluation which may be scheduled as late as $(j+2)h-\eta$. Since we assume eager transition policy, this transition will be taken at $(j+2)h-\eta$. Letting $\eta, \epsilon \rightarrow 0$, and observing that in the model the guard may be enabled as early as updates to the variables which in the worst case takes another $2h^{'}$ to reflect in the code, we get the desired result. □

Example 3: Consider the system in Example 1. Here, we have that the guard G_1, depends on variables x and z. The guard holds for time (0.003 - 0.002) = 0.001. Now, Applying Theorem 2 here we find, 2(0.002) = 0.004 ($>$ 0.001) and hence we cannot guarantee no missed transitions.

Example 4: Now consider yet another example. Consider the same system as in the previous example but different guard set. $G_1 : z < 0.01 \ \wedge \ x > 0.003$. In this case, the guard is enabled for (0.01-0.003) = 0.007 (> 0.004). Hence, by Theorem 2, we can say that there will be no missed transitions. Indeed, it is easy to verify for any scheduling order, that the transition will not be missed.

Note that for the implementation to be faithful, we need to guarantee both no missed as well as no faulty transitions and Theorems 1 and 2 give us a sufficient condition.

Theorem 3. *Consider a distributed system of agents. Let the guard set of every agent be disjoint and every invariant be instrumented with some constant. Let the instrumented guard and the invariant overlap at least $2(h + h^{'})$. And if all the variables in the code have a constant error bounded by δ, then the implementation is a faithful implementation with no missed transitions.* □

5 Conclusion

We have been developing a code generation and validation framework for hybrid systems models. In this paper, we have presented an extension of this framework to distributed embedded systems. We have defined criteria for faithful implementation of the model and identified sufficient conditions to guarantee this.

There are several possible directions for future research. In a distributed implementation, communication delay is a concern and has to be considered, perhaps at the model-level. Here we have considered systems with constant dynamics, we are currently trying to extend this to include larger class of systems. We are also working on a dynamic instrumentation technique that will extend code generation to systems for which local error can be estimated. Since embedded systems are resource constrained, it is natural to also consider code optimization as a logical next step.

With the framework itself, we are planning to implement it in a more integrated form by using a general IDE (e.g., Eclipse). We are also planning to extend the architectural description facility of our framework by incorporating modeling languages specialized for architecture description (e.g., AADL).

References

[1] 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 Comp. Science*, 138:3–34, 1995.

[2] R. Alur, T. Dang, J. M. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, Jan. 2003.

[3] R. Alur, T. Dang, and F. Ivančić. Counter-example guided predicate abstraction of hybrid systems. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pages 208–223. Springer-Verlag, 2003.

[4] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositioal refinement of hierarchical hybrid systems. In *Hybrid Systems: Computation and Control, Fourth International Workshop*, LNCS 2034, pages 33–48, 2001.

[5] R. Alur, F. Ivančić, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchial hybrid models. In *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

[6] M. Anand, J.Kim, and I.Lee. Sound code generation from hybrid system models: Some theoretical results. In *Technical report, MS-CIS-05-03, University of Pennsylvania*, 2005.

[7] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[8] A. Deshpande, A. Göllu, and P. Varaiya. SHIFT: a formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems V*, LNCS 1567. Springer, 1996.

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

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79:1305–1320, 1991.

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

[12] Y. Hur, J. Kim, I. Lee, and J.-Y. Choi. Sound code generation from communicating hybrid models. In *Proceedings of HSCC*, LNCS 2993, pages 432–447, 2004.

[13] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[14] J. Kim and I. Lee. Modular code generation from hybrid automata based on data dependency. In *Proceedings of RTAS*, 2003.

[15] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600. Springer-Verlag, 1991.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: the Art of Scientific Computing, 2nd Ed.* Cambridge University Press, Cambridge, UK, 1999.