Generating Embedded Software from Hierarchical Hybrid Models

Rajeev Alur, Franjo Ivančić, Jesung Kim, Insup Lee and Oleg Sokolsky*
Department of Computer and Information Science
University of Pennsylvania

Abstract

Benefits of high-level modeling and analysis are significantly enhanced if code can be generated automatically from a model such that correspondence between the model and the code is precisely understood. For embedded control software, it is believed that *hybrid system* is an appropriate modeling paradigm because of it can be used to specify continuous dynamics as well as discrete switching between modes. Establishing a formal relationship between hybrid model and code, however, is particularly challenging since an execution behavior can be different from behaviors allowed by the model due to sampling and switching errors. In this paper, we describe a formal approach to compile the modeling language CHARON that allows hierarchical specifications of interacting hybrid systems. We show how to exploit the semantic foundations of CHARON to generate code from a model in a modular fashion, and identify sufficient conditions on the model that guarantee the absence of switching errors in the compiled code. The approach is demonstrated by compiling a model for coordinated motion of legs for walking onto Sony's AIBO robot.

1 Introduction

An embedded system typically consists of a collection of digital programs that interact with each other and with an analog environment. As computing tasks performed by embedded devices become more sophisticated, the need for a sound discipline for writing embedded software becomes more apparent (c.f. [18, 22]). Model-based design paradigm, with its promise for greater design automation and formal guarantees of reliability, is particularly attractive in this domain. Contemporary industrial control design already relies heavily on tools for mathematical modeling and simulation. Even though many of such tools support automatic code generation from the model (for example, Simulink, see `www.mathworks.com`), the emphasis has been performance-related optimizations, and many issues relevant to correctness are not satisfactorily addressed. First, the precise relationship between the model and the

*This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, and DARPA ITO MOBIES F33615-00-C-1707.

generated code is rarely specified or formalized. Second, the continuous blocks are either ignored, or discretized before code generation. Finally, code generation typically means generation of tasks, and does not incorporate scheduling. Consequently, the correspondence between the model and the code is lost, and analysis results established for the model are not meaningful for the code. The desire to bridge this gap motivates our research.

Traditionally, control theory and related engineering disciplines, have addressed the problem of designing robust control laws to ensure optimal performance of systems with continuous dynamics. For example, given system dynamics $\dot{x} = f(x, u)$, where x represents the system state and u represents the control input, one can design a control law $u = g(x)$ with respect to the given specification (c.f. [5, 10]). To implement this control law, one must first determine the sampling period Δ. The software, then, is a task that consists of "sense x; compute $u = g(x)$; send u to actuators," which must be scheduled every Δ time-units. Compared to the mathematical model $\dot{x} = f(x, u); u = g(x)$, the behavior of the generated code may be different for many reasons: the system state x may not follow the model $\dot{x} = f(x, u)$ precisely, sampling introduces discretization errors, and there are numerical errors in computing g. However, the discrepancy can be bounded if we assume that the system state follows the model closely, there is a bound on numerical errors, and the control law is robust.

Typical controllers, however, are rarely purely continuous. Discreteness arises due to a variety of reasons such as communication, concurrency, and multiple modes of operation. An appropriate mathematical model is a *hybrid system*. A hybrid system combines the traditional FSM-based model of discrete control with continuous models of differential and algebraic equations [1, 24]. Hybrid systems is the focus of increasing research in control theory as well as in formal modeling and verification in recent years (c.f.[4, 23]). Consider a system with two modes. Initially the system is in mode M_1 with dynamics $\dot{x} = f_1(x, u)$. It can stay in the mode M_1 as long as the invariant $a(x)$ holds, and switches to mode M_2 if the condition $p(x)$ holds. The dynamics is $\dot{x} = f_2(x, u)$ in mode M_2. Suppose we design the controllers $u = g_1(x)$ and $u = g_2(x)$ for the two modes separately. The software corresponding to this controller samples the system state x every Δ time-units. It has a mode variable which is initially M_1, and it is updated to M_2 if $p(x)$ evaluates to true. The control output u is computed by evaluating either $g_1(x)$ or $g_2(x)$ based on the value of the mode variable. In terms of discrepancy between the high-level model and the code, in addition to errors in implementing continuous controllers in individual modes, now there can be errors in switching from mode M_1 to M_2 which can cause significant problems. There is no general theory of approximation and robustness of controllers in presence of switching. If a switch is missed, the resulting trajectory can be entirely different. Detecting switching events as accurately as possible has been a topic of research for simulation of hybrid systems (c.f.[14]), but such techniques cannot be implemented in real-time. In this paper, we initiate the study of formulating and limiting discrepancies between the model and the generated code for hybrid systems. We show that if the invariant $a(x)$ of a mode and the guard $p(x)$ of a switch out of this mode overlap for a duration greater than the sampling period, then the code will not miss the switching event. Such a condition can be checked statically, at least for systems with linear dynamics. It is worth noting that this requirement implies that the model is inherently non-deterministic: semantically, the switch may happen at any time in the duration for which the invariant and the guard overlap. This is in contrast with the hypothesis that modeling languages for reactive systems should have deterministic reactions to external inputs to be implementable (c.f. [7, 22]).

The second focus of this paper is generating the code in a modular fashion from hierarchical descriptions. Modern software design paradigms, such as UML, promote *hierarchy* as one of the key constructs for structuring complex specifications [8, 25]. We are concerned with two distinct notions of hierarchy. In *architectural hierarchy*, a system with a collection of communicating agents is constructed by parallel composition of atomic agents, and in *behavioral hierarchy*, the behavior of an individual agent is described by hierarchical sequential composition. The former hierarchy is present in almost all concurrency formalisms, and the latter, while present in all block-structured programming languages, was introduced for state-machine-based modeling in STATECHARTS [16]. The logical concurrency in architectural hierarchy leads to a set of tasks, and issues such as scheduling and distribution among multiple processors have been well studied in the real-time systems community (c.f.[9]). We are concerned with compiling a hierarchically structured mode in a modular manner so that sub-modes can be compiled separately allowing reuse.

Our ideas are demonstrated in the context of the modeling language CHARON, a design environment for specification and analysis of embedded systems [2]. 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 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 [3]. For analysis it supports simulation, and formal verification of safety properties for a restricted subset (where discrete state is finite, and continuous dynamics is linear) [2].

We exploit the hierarchical semantics of CHARON to generate the code in a modular fashion. Each mode is compiled as a `C++` class, and the discrete and continuous update methods for a mode simply call the corresponding methods for sub-modes in a hierarchical manner. As a case study, we have developed a compiler from CHARON to Sony's AIBO robots (see `www.aibo.com`). The code generation, of course, has to address all the details in mapping the logical constructs. The specific model described in this paper corresponds to coordinating the four legs to make the robot walk. The individual leg has four modes and the switching conditions demonstrates the flexibility offered by high-level modeling in mixing time-triggered and event-triggered switching: some switches are triggered updates of discrete variables by other agents, some are triggered by elapse of time for a specified duration, while some are triggered by conditions on continuous variables. This example also

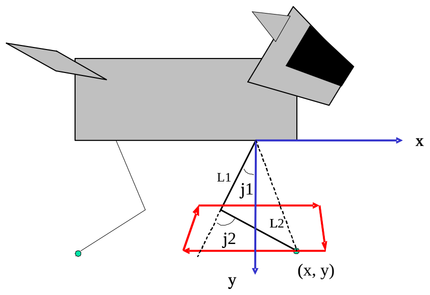


Figure 1: Leg model

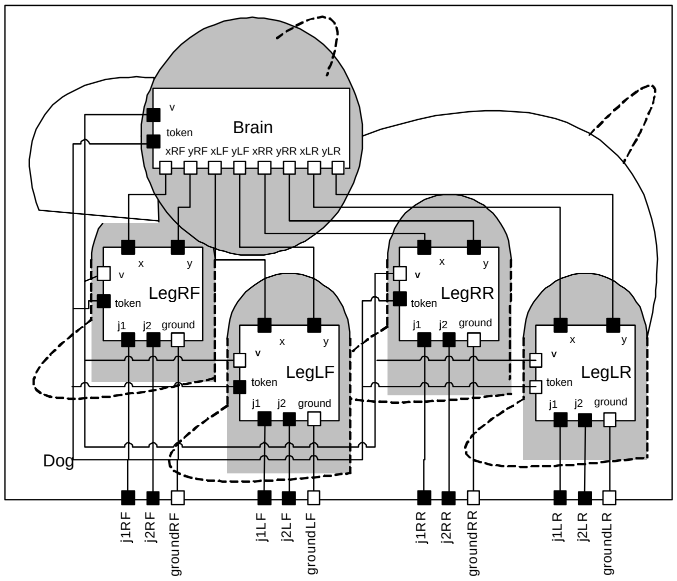
shows benefits of modeling: the CHARON model is simple and small, hides all the messy platform-dependent details of programming AIBO, and can be subjected to simulation and reachability analysis to prove formally safety properties.

Related work. Commercial modeling tools such as RationalRose and Simulink support code generation, but it is not formal and the discrepancy between model and code is not addressed. The code generation from synchronous languages for reactive systems such as STATECHARTS [16], ESTEREL [7], and LUSTRE [15], is significantly more rigorous. However, these languages do not support specification of continuous activities. Issues such as efficiency of generated code and dealing with logical concurrency and communication are addressed, but they do not exploit sequential hierarchy for modular compilation. SHIFT is a language for dynamic networks of hybrid automata [11], and it supports code generation, but the focus is not on modularity and correctness issues. 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 [19, 20]. 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 [13] and GME supports integration of multiple views of the system [21].

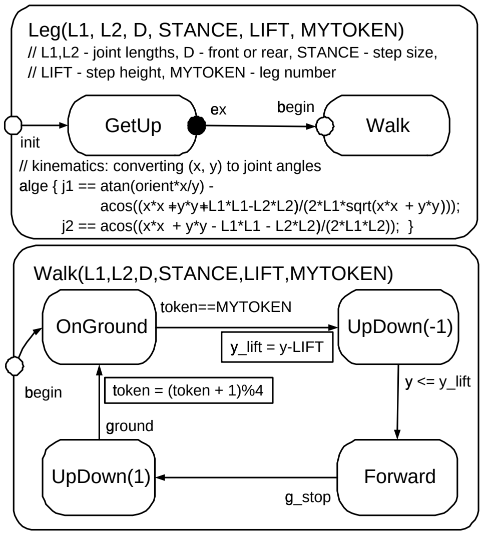
2 Modeling Language

We define the formal modeling language CHARON, illustrating it with an example, and give the intuition for its semantics. More details can be found in [3]. To enhance presentation, we use a pictorial view of language constructs. The language supports both visual and textual representations. Throughout the paper we use a recent case study to illustrate the modeling concepts of the language, the salient aspects of our code generation approach, and how they relate to each other. The case study models the walking process of a four-legged robot and uses this model to generate the code for the robot dog AIBO, manufactured by Sony. The controller for walking is described in detail in [17].

The conceptual model for a leg is shown in Figure 1. The controller assumes that each leg has the hip and the knee joints that can be controlled by giving the desired angular position of the joint (i.e., angles $j1$ and $j2$ in Figure 1). The control objective for each leg is to ensure that the leg moves in such a way that the paw (i.e., the end of the knee joint)



(a) Architectural hierarchy



```
// L1,L2 - joint lengths, D - front or rear, STANCE - step size,
// LIFT - step height, MYTOKEN - leg number
```

```
// kinematics: converting (x, y) to joint angles
alge { j1 == atan(orient*x/y) -
            acos((x*x +y*y+L1*L1-L2*L2)/(2*L1*sqrt(x*x  + y*y)));
       j2 == acos((x*x  + y*y - L1*L1 - L2*L2)/(2*L1*L2));  }
```

(b) Behavioral hierarchy

Figure 2: Model hierarchy

follows the shown trajectory. The global control objective is to ensure that only one leg is up in the air at any moment and that the center of mass for the robot is within the triangle given by the three legs on the ground.

2.1 Agents and Architectural Hierarchy

An *agent* represents an autonomous entity that operates by communicating with other agents via shared variables. We distinguish between *atomic* and *composite* agents. A composite agent $\langle SA, V\rangle$ consists of a set of variables V and a non-empty set of sub-agents SA. An atomic agent $\langle M, V\rangle$ does not have any sub-agents and its behavior is given by a *mode* M, as described in the next section. Figure 2(a) shows the architecture of the model represented as the top-level agent `Dog`. It contains five concurrent sub-agents representing the legs and the brain of the dog. The brain agent serves as the controller for the leg agents. Note that the leg agents are instances of the same agent `Leg`, which we describe below. The variables of an agent are partitioned into private, internal to the agent, and input and output variables. Each agent has a well-defined interface which consists of its typed input and output variables, represented visually as blank and filled squares, respectively. Connections between variables represent data flows between the agents in the model. The `Brain` agent reads variables x and y, representing leg positions, from the `Leg` agents, renaming them appropriately. That is, the variable x of agent `LegRF` is renamed to xRF in the agent `Brain`, and so on. The agent `Brain` provides the desired speed of the dog represented by the variable v, which is read by the `Leg` agents. The variable *token*, shared between all agents and used to tell which leg is currently in the air, can be modified by each of them. All of these variables, however, are internal to the `Dog` agent. The interface variables of the `Dog` agent are eight output variables that represent commands sent to the joint motors in each leg, and four input variables that represent ground contact sensors in each leg. The agent `Leg` is atomic. Its interface contains the position variables x and y, the joint commands $j1$ and $j2$, the *token* variable, and two input variables, v provided by the `Brain` agent, and *ground* provided by the external sensor.

2.2 Modes and Behavioral Hierarchy

Modes represent behavioral hierarchy in the system design. Each mode describes a continuous behavior and a single thread of discrete control. A mode can be active or inactive during an execution, depending on whether the position of discrete control is within the mode or not. Formally, a mode M is a tuple $\langle E, X, V, SM, Cons, T\rangle$, where E is a set of *entry control points*, X is a set of *exit control points*, V is a set of variables, SM is a set of sub-modes, $Cons$ is a set of constraints, and T is a set of transitions. Each mode has a well-defined data interface consisting of typed global variables used for sharing state information, and also a well-defined control interface consisting of entry and exit points, through which discrete control enters and exits the mode. A top-level mode, which is activated at the start of an execution and is never deactivated, has a special entry point `init`. Each mode has a default, unnamed entry and exit point. The set SM can contain a number of *sub-modes* connected by *transitions* from the set T. We distinguish between *entry* transitions, leading from an entry point of M to an entry point of a sub-mode of M, *exit* transitions, leading from an exit point of a sub-mode to an exit point of M, and *internal* transitions, connecting exit points of sub-modes to entry points of sub-modes. Each transition has a *guard* and an *action*, and is used to transfer discrete control from one sub-mode to another. During an execution, transitions occur instantaneously and can be taken when its guard is satisfied. When the transition is taken, an associated action is executed, assigning new values to the variables of the mode. In addition to discrete steps, the variables of the mode continuously evolve with the passage of time according to the set of constraints $Cons$. A mode can contain three kinds of constraints. Continuous trajectories of a variable x can be given by either an *algebraic constraint* A_x, which defines the set of admissible values for x in terms of values of other variables, or by a *differential constraint* D_x, which defines the admissible values for the first derivative of x with respect to time. Additionally, $Cons$ can contain an *invariant* I, which is a boolean predicate over the mode variables. Only those trajectories are allowed that continuously satisfy the invariant of the mode.

We represent modes visually as state machines with transitions between them. Transitions are labeled by guards and actions. To make it easier to visually distinguish between guards and actions, actions are boxed. Entry and exit points are denoted as blank and filled circles, respectively. Transitions incident to a default entry or exit point, which are not shown on the picture, are visually attached directly to the box representing the mode.

The mode `LegMode`, the top-level mode of the agent `Leg`, is shown in Figure 2(b). Invariants as well as the complicated expression for the guard `g_stop`, are omitted to avoid cluttering the picture. The mode contains five sub-modes. The sub-mode `GetUp` is entered during initialization and ensures that the dog is standing before walking begins. It has its own internal structure, which we do not discuss here. The other four modes correspond to the four segments of the leg trajectory in Figure 1. Note that the two sub-modes that move the leg up and down are instances of the same mode with different parameter values.

```
mode UpDown(real dir) {
  read real v;
  write real x, y;
  diff { d(x) == 3*v; d(y) == dir*3*v; }
  inv { y ≥ y_limit; y ≤ y_upper_limit; }
}
```

Figure 3: An atomic mode

To ensure stability of the robot, only one leg can be in the air at any time. We use the shared variable `token` to switch legs. A leg can lift off the ground only if the token is equal to its number (given as the mode parameter `MYTOKEN`). The leg then moves diagonally upwards until the desired height is reached, and the mode is switched to begin horizontal movement. When the leg is moved forward enough, another mode switch happens and the leg is moved diagonally down. When the leg reaches ground, a signal from the paw sensor sets the variable `ground`, the mode switch occurs and the token is passed to the next leg by the action of the transition.

At the lowest level of the behavioral hierarchy are atomic modes. They describe purely continuous behaviors. For example, Figure 3 illustrates the behavior prescribed by the mode `UpDown`, which specifies the desired trajectory for the paw moving diagonally up or down by means of a differential constraint that asserts the relationship between the horizontal and vertical velocities of the paw, represented as the first time derivatives of the paw coordinates `x` and `y`, and the input variable `v`, representing the desired speed. The trajectory is also constrained by the invariant specifying a range of valid vertical positions.

2.3 Semantics

The CHARON language has modular trace-based formal semantics. That is, the semantics prescribes how to construct the set of executions of an agent or mode based on its sub-agents (or sub-modes) and constraints within the mode. Instead of presenting the formal semantics for modes and agents, which can be found in [3], we give an informal description of an admissible execution here. Later, in Section 4, we present a simulator that agrees with the semantics at certain discrete points in time, and constructs the execution in a precise (albeit non-modular) fashion.

An execution of a mode M is constructed as follows. The state of a mode includes the values of the mode variables and, in a non-atomic mode, an additional variable that records the active sub-mode currently having the control. A mode becomes active when control is transferred to one of its entry points. The mode can remain active as long as its invariant is satisfied. As soon as the invariant is violated, time cannot progress any further and the mode is forced to transfer control to one of its exit points by means of an exit transition. If the invariant is satisfied, the mode can take a *continuous step*, during which time progresses and the state of the mode is continuously updated according to the differential and algebraic constraints of the mode and its active sub-mode. Discrete control is not affected by a continuous step. Alternatively, if a mode has an enabled transition t, it can execute a discrete step, during which time does not progress and t is executed. Mode variables are updated according to the action of t and, if the target of t is an entry point of a sub-mode m, m becomes the active sub-mode. A transition t is enabled if the guard of t is satisfied and control has been transferred to the control point that is the source of t. A transition whose source is the default exit point of a sub-mode is enabled whenever its guard is satisfied, which allows us to implement preemption.

Consider the example in Figure 2(b). The transition in mode `Leg` from sub-mode `GetUp` to the sub-mode `Walk` can be taken whenever `GetUp` completes its execution by transferring control to its exit point `ex`. By contrast, when the leg touches ground and the variable `ground` is set, the transition interrupts the execution of sub-mode `UpDown(1)` and transfer control to `OnGround`.

An execution of an agent is constructed by either taking a discrete step in one of its sub-agents or by taking a continuous step in all sub-agents simultaneously. The execution stops if time cannot be advanced (that is, one of the modes has a violated invariant) and no mode has an enabled transition. In this case, the model is deadlocked. If the model cannot be deadlocked, that is, whenever an invariant is violated, there is an enabled transition in one of the modes, we call it a non-blocking hybrid system.

3 Code Generation

In this section, we present the code generator that compiles CHARON models for the target platform. The process can be decomposed into two phases as shown in Figure 4. The front-end transforms the CHARON model into a high-level language representation. One of the main differences between CHARON models and high-level language programs is that in the former the state is defined in the continuous-time domain whereas in the latter the state changes in a discrete fashion. We approximate the continuous behavior by updating the state of the continuous model at every period Δ. Obviously, we lose certain properties of the model through the approximation, but we can guarantee that transitions are not missed if the period Δ is small enough. (We will come back to this issue later in Section 4.)

The code generated by the front-end is platform-independent and needs to be ported to the execution environment of a specific target platform. The back-end performs platform-specific adaptation of the code to bind abstract objects of the model to concrete objects of the platform. The resulting code can be compiled into a platform-specific binary form using a target compiler, as we will explain in Section 3.2.

Faithful implementation of the CHARON semantics in a code generation algorithm is complicated by the fact that, conceptually, executions of agents proceed concurrently, while on a single-processor platform they will by necessity be executed sequentially. Therefore, we have to ensure that the order of evaluation of the agents is consistent with the dependencies between variables in the model. For example, if an algebraic constraint for variable x in a mode contains variable y in its right-hand side, the constraint updating y, possibly in a mode of another agent, must be processed before the constraint for x. The same applies for the evaluation of guards: before the guard of a transition is evaluated, we have to ensure that all variables it uses have been updated. Of course, these dependencies can change dynamically as the execution moves from mode to mode. Variable dependencies will have to be updated with each mode switch. To make manipulation of dependencies easier, we assume that there are no cyclic dependencies in any state of the system during its execution.

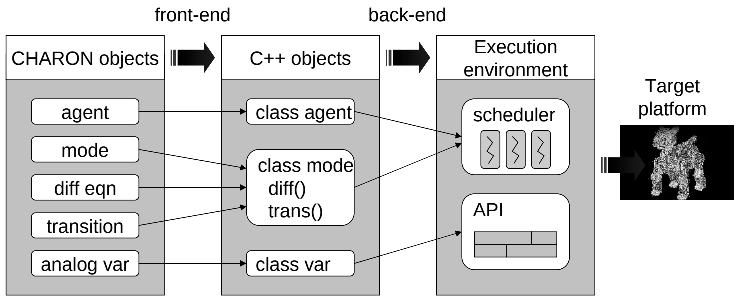


Figure 4: Code generator.

3.1 Front-end

The role of the front-end is to parse the given CHARON model into an abstract syntax tree and map each node of the tree into an object of the target programming language. We chose `C++` as an intermediate target language, mainly because the object-oriented features of the language best suit CHARON and make the code generation process simpler, and also because the language has been deployed in many real systems, including AIBO.

Modularity of the original model is captured by aggregating objects belonging to the same mode in a `C++` class that can be compiled separately. The `C++` class consists of methods implementing equations and transitions, pointers to the external variables and the sub-modes. The code generator produces a `C++` class for a given abstract syntax tree of a mode $M = \langle E, X, V, SM, Cons, T\rangle$ consisting of entry points E, exit points X, variables V, sub-modes SM, constraints $Cons$, and transitions T, as described in Algorithm 1 in Appendix. The algorithm makes a recursive call for each sub-mode $m \in SM$ to generate separate classes. Note that the algorithm does not reference any elements of upper-level modes nor any elements of sub-modes except for the sub-mode interface. This implies that the generated code is modular and can be compiled and executed independent of other modes. In the next, we describe the algorithm in more detail for each element of a mode. To simplify the algorithm description, we assume a utility function GenStmt() that produces a syntactically correct `C++` statement from given inputs.

Variable. Variables in CHARON are either local or global. Each local variable $v \in V_l$ is translated into a variable class instance, while each global variable $v \in V - V_l$ is translated into a reference to a variable class instance that is instantiated at an upper-level mode where the same variable is declared as a local variable. Variables are represented by instances of class `var` that has methods `read()` and `write()`, used to get the value and assign a new value to the variable. Top-level variables need to be handled differently, since they are mapped to platform specific APIs. This mapping is done by overriding `read()` and `write()` methods in a derived class of `var` by the back-end, without modifying the code produced by the front-end.

Differential constraint. A differential constraint D_x of the form $\dot{x} = f_{D_x}$ declares that a variable x should evolve continuously at a rate given by the expression f_{D_x} over variables which may be continuous. Theoretically, this requires evaluation of the expression f_{D_x} and valuation of the variable x at every infinitesimal period. We approximate this specification into an assignment statement that is executed at every period to increment the variable in proportion to the length of the period, which is given by the parameter `Delta` of the

```
void UpDown::diff(double delta) {
    x += (3*v) * delta;    // d(x) == 3*v;
    y += (dir*3*v) * delta;  // d(y) == -dir*3*v;
}
```

Figure 5: Generated code for differential equations.

function. For example, Figure 5 shows the code for differential equations of mode `UpDown` ($\dot{x} = 3v; \dot{y} = dir \cdot 3v$) given in Figure 3. This method, known as *Euler's method*, is efficient to compute and produces good results for our models. More advanced, but more expensive methods can be used to improve accuracy.

Algebraic constraint. An algebraic constraint declares equality between variables that should be satisfied at all times. In CHARON, an algebraic constraint $A_x \in Cons$ of a variable x is specified in the form of an equation $x = f_{A_x}$, where f_{A_x} is an expression of other variables. Such constraints are translated into assignment statements, which are evaluated in the dependency order. This requires a dynamic dependency graph between equations that is updated by mode switches. A dependency tracking mechanism is implemented in the base class for modes and does not depend on the generation algorithm. We omit the implementation details to conserve space.

Invariant. An invariant $I_x \in Cons$ declares a condition that should be satisfied at all times while the mode is active. In general, violation of an invariant means that the implementation is not faithful to the specification, or the model is infeasible. We translate each invariant to an assertion statement for run-time checking of correctness. Our framework also provides a means for static analysis of invariant violations as explained in Section 4.

Transition. Transitions specify the control flow of the model. A transition $t \in T$ is translated into an if-then statement where the if-block contains the guard g and the then-block contains the optional discrete actions α as shown separately in Algorithm 2 in Appendix. In addition to the guard, when the transition specifies a specific control point in the source mode, the if-block also checks whether the control has been transferred to the control point. When the guard is true and the control has been transferred to the source control point, the associated discrete action in the form of a set of assignments is executed. Note that this implementation enforces the transition is taken as soon as it is detected enabled. While the CHARON semantics allows us not taking enabled transitions deliberately, we take this approach because it makes verification easier as we will explain in the next section. After executing the discrete action, control is passed to the destination control point by invoking the corresponding method of the destination mode. When the transition does not specify a specific control point in addition to the source mode (i.e., default exit control point), it is a *preemption*, and it can be taken if the guard is true, even when the sub-mode still has the control. When this happens, the local variables in the source mode and the pointer to the currently active sub-mode are preserved as they are so that it can resume from the same state when the control enters through the default entry control point. Evaluation of guards also proceeds in the order of variable dependencies as described above. When more than one transition are enabled and there is no dependency between them, the code executes the transitions in the order chosen by the code generator non-deterministically. Note that any arbitrary choice among enabled transitions is valid provided that the model is non-blocking (i.e, taking the transition does not lead to violation of the invariant).

Control Point. The code generator produces for each entry point $e \in E$ a corresponding method e`()` that implements the entry transition (see Algorithm 3 in Appendix). Each generated method checks the guard g, performs the associated discrete actions α when the guard is true, and invokes the method corresponding to the destination entry point $m'.e'$`()` to trigger a cascade of entry transitions towards a leaf mode. In addition, it updates the pointers of each variable that represent the data dependency. On the other hand, for each exit point $x \in X$, a method is generated that tests whether control has been transferred to the control point. The method checks a flag `exitCode` that is set by the function `trans()` when an exit transition is performed.

Mode. The class for modes has two methods, `continuousStep()` and `discreteStep()` that performs evaluation of the mode. Each method is invoked by the same method of the parent mode. They are implemented in a base class `mode` since they are common to all the modes. The class also contains run-time information such as the pointer to the currently active sub-mode. This pointer constitutes a linked list of active sub-modes from the top-level mode to some leaf mode. The methods of the top-level mode are invoked by the corresponding methods in the class for agents.

Agent. We have implemented a single-threaded code generation scheme, since hybrid models generally have much finer granularity concurrency than that is supported by the traditional multitasking mechanism of the operating system. That is, execution of concurrent sub-agents are interleaved at the granularity of the period Δ in a single thread of execution. The top-level agent has a single method `update()` that is called periodically at every Δ by the timer or a periodic task of the platform. It executes first the continuous steps and then the discrete steps of the sub-agents.

3.2 Back-end

The `C++` code generated by the front-end can be compiled into binary object code suitable for the target platform once a target compiler is given. The next step is to relate variables to specific objects in the target platform. For example, if the model denotes a joint of the head of the robot as variable x, we need to relate variable x to the servo motor that controls the position of the head. In other words, we need to *bind* objects in the model to objects in the target platform just like high-level language compilers bind variables to memory addresses.

While variables in programming languages are generally bound only to memory addresses, variables in the model may be bound to a hardware register, an I/O port, or a parameter or the return value of a system call/API, as well as a memory address, depending on the abstraction level of the program execution environment. These bindings require extra code that *glues* objects in the model and the platform. Compiled and linked together, the glue code allows the generated code to communicate with the platform transparently. The back-end generates the glue code when information on binding is given.

We use a Makefile-like script to describe relationship between objects in the model and objects in the platform. Specifically, the script consists of colon-separated dependency relations and optional rules (i.e., code fragments) to relate the two. For example, a script that relates variable x to an API function `syscall()` and a constant `HEAD_JOINT` used as a parameter is shown in Figure 6. The script shown in the figure lets the back-end translate

Script

```
x.write: HEAD_JOINT
        syscall(HEAD_JOINT, x);
HEAD_JOINT:
        #include "system.h"
```

system.h

```
#define HEAD_JOINT "PRM:/r1/c1-Joint2:j1"
void syscall(const char *, double);
```

Figure 6: Script for binding.

write access to variable `x` to a API call `syscall(HEAD_JOINT, x)`, with an additional parameter `HEAD_JOINT` defined in the API header file `system.h`. This code fragment is appended without modifying the code generated from the front-end by creating a derived class that overrides the default `write()` method.

3.3 Modular Compilation

The generated code is modular in the sense that each mode and agent is mapped to a `C++` class that can be compiled separately and reused in different contexts. Each module can be reused not only in the level of models but also in the level of code. For example, the code for the walking process can be used in a larger application without modifying the original model or the generated code.

In addition to reusability, modularity is salient in two aspects. First, hybrid system models in many cases contain both the controller and environment, and they need to be decoupled since only the former is subject to code generation. Our code generator allows code generation only for selected modes/agents, and the decoupling comes naturally. Second, modularity of the generated code is essential when the target platform is a distributed system consisting of multiple processing elements. We can port each module to a different execution environment, possibly using different target compilers and/or compilation options. The distributed modules can interact with each other when the API for communication is associated to the variable class, since variables are the only interface between modules.

4 Discretization Error Analysis

In this section, we analyze code generation from a given hybrid system model, that is, we analyze the accuracy of the generated code with respect to the hybrid system. There are a variety of errors that are introduced to the system during the generation of discrete code with fixed sampling step-size. An overview of the various classes of errors is given in Section 1. Here, we only consider the errors due to the discretization of a hybrid system. For this we will not consider the architectural hierarchy of the system, and hence assume without loss of generality, that our hybrid system consists of n concurrent atomic agents $A_1, A_2, \ldots, A_n$. For the sake of clarity of the following description, we assume that all transitions end in a leaf mode by executing entry transitions as necessary to reach a leaf mode. For an atomic agent $A = \langle M, V\rangle$ we denote the set of all variables in any sub-mode assuming naming conflicts have been resolved by $\mathcal{V}_A$, and the set of valuations of these by Σ_A. For each atomic agent we call the set of nested modes that are active (see Section 2.2) its *active modes* M_A. The active modes M_A of an agent A basically consist of a path from the top-level mode of the agent A to some leaf mode. A state of an atomic agent then consists of an active mode and a value to all its variables $\mathcal{V}_A$. The set of states for an atomic agent A is denoted by $\mathcal{X}_A$, and the initial set of states is denoted by $\mathcal{X}_A^0 \subseteq \mathcal{X}_A$. We can then define the set of *active constraints* $Cons(M')$ given a state $x = (M', v)$ as the union of all constraints in M', the set of *active invariants* $I(M')$ as the union of all invariants in M', and the set of *active transitions* $T(M')$ as the set of all transitions of modes in M', such that the source of the transition is an exit control point of a mode in M'. We denote the set of valuations of $\mathcal{V}_A$ that satisfy all invariants of an active mode M' with $\mathcal{I}(M') \subseteq 2^{\Sigma_A}$.

We want to check feasibility of the code generation task for a sampling period Δ. The environment of an agent plays a central role in determining this kind of feasibility. We consider closed system of agents, assuming without loss of generality that naming conflicts have been resolved. We define the set of *globally active modes* M_A for an agent $A = \langle\{A_1, \ldots, A_n\}, V\rangle$, where each agent A_i is atomic, as the cross-product of the active modes of its sub-agents. The set of all variables of A denoted by $\mathcal{V}_A$ is the union of all variables of its sub-agents, and the set of valuations of all variables is denoted by Σ_A. A state of the agent A then consists of a globally active mode and an evaluation of all its variables. The set of all states of an agent A is denoted by $\mathcal{X}_A$, and the set of initial states is denoted by $\mathcal{X}_A^0 \subseteq \mathcal{X}_A$. The set of *globally active constraints* $Cons(M')$ given a state $x = (M', v)$ is the union of all active constraints of its sub-agents, the set of *globally active invariants* $I(M')$ is the union of all active invariants of its sub-agents, and the set of *globally active transitions* $T(M')$ is the union of all active transitions of its sub-agents. We call a function $\Phi : \Sigma_A \times \mathbb{R}_{\geq 0} \to \Sigma_A$ an admissible flow for the globally active mode M', if $\forall v \in \Sigma_A : \Phi(v, 0) = v$ and $\Phi(v, t)$ is a solution to all globally active algebraic and differential constraints in M'.

We define a *fixed step-size simulator* for a given hybrid system as a first step towards the generated code. The fixed step-size simulator with period Δ for a given CHARON model can be seen as one computable approximation of the mathematical hybrid system model. Given an admissible initial state of an agent, we evaluate the behavior of an agent at time points $\Delta, 2\Delta, 3\Delta, \ldots$ As mentioned earlier, we assume that the dependency graph of atomic agents based on their globally active transitions is acyclic.

Definition 1 *A* **fixed step-size simulator** *with period Δ given a closed agent $A = \langle SA, V\rangle$ of atomic sub-agents $SA = \{A_1, \ldots, A_n\}$ computes a potentially partial function $f_A : \mathbb{N} \to \mathcal{X}_A$. The function f_A is defined as $f_A(0) \in \mathcal{X}_A^0$, and $f_A(k+1) = f_n(f_A(k))$, with*

1. *$f_0(M, v) = (M, \Phi(v, \Delta))$ where Φ is an admissible flow in M, such that $\forall t \in [0, \Delta] : \Phi(v, t) \in \mathcal{I}(M)$; and*

2. *there exists an admissible ordering $o : \{1, \ldots, n\} \to SA$, that corresponds to a full ordering of the partial order given by the dependency graph of atomic agents based on their active transitions, such that one of the following two evaluations is used for $1 \leq i \leq n$:*

(a) if the invariants of the active modes $M_{o(i)}$ of atomic sub-agent $o(i)$ are not violated, then $f_i(M, v) = f_{i-1}(M, v)$; or

(b) if there exists an enabled active transition $t \in T(M_{o(i)})$, that is $guard_t(v) = \mathtt{true}$, where t is switching to the globally active mode M', then $f_i(M, v) = (M', actions_t(f_{i-1}(M, v)))$.

Please note that a fixed step-size simulator with period Δ for a non-blocking hybrid system H may deadlock. In fact, there are non-blocking hybrid systems such that there is no step-size Δ, that would produce a non-blocking fixed step-size simulator. It should also be noted that this definition describes the computation of a function with non-deterministic choices. A fixed step-size simulator thus can compute a set of admissible functions according to this definition. The goal of the forthcoming analysis is to assure the feasibility of computing one such admissible function using the generated code. Lastly, it should be noted that the invariant of some active mode may be violated in the case 2.(b). Part 2. models instantaneous transitions jumps between modes, which are allowed to pass through intermediate, zero time invariant violations. However, for a non-blocking behavior of the fixed step-size simulator, one needs to assure that $f_A(k)$ is not violating any invariant for all $k \in \mathbb{N}$.

This model assumes that the simulator can sense the world, compute necessary updates, and act accordingly all in zero time. This is not a realistic model of embedded systems. An embedded system needs to accommodate a time delay for sensing the world, as well as computation and execution time. We will describe properties of the fixed step-size simulator though as a first approximation of a model for our generated code. We define a class of hybrid systems for which we can prove that a fixed step-size simulator is an appropriate execution model. We denote an execution to be appropriate if it corresponds to a valid trace of the hybrid system constrained to sampling points. We now define a class of closed agents for which we can show that it can be faithfully simulated by the aforementioned fixed step-size simulator. Intuitively, we consider the class of closed agents for which guards and invariants over continuously updated variables overlap for a duration greater than the sampling period. We use the non-determinism of the continuous flow to allow a discrete and deterministic simulation a decision criterion when to switch modes using enabled transitions. We define a function $\mathtt{Post}_\Phi : 2^{\Sigma_A} \times \mathbb{R}_{\geq 0} \rightarrow 2^{\Sigma_A}$ for an admissible flow Φ of an agent A of atomic sub-agents as: $\mathtt{Post}_\Phi(X, \tau) = \{v \in \Sigma_A \mid \exists x \in X, 0 \leq t \leq \tau : \Phi(x, t) = v\}$.

Definition 2 *Given a globally active mode M for a closed agent $A = \langle SA, V\rangle$ of atomic sub-agents $SA = \{A_1, \dots, A_n\}$ and an admissible flow Φ, define the guards set $\mathcal{G} \subseteq \mathcal{I}(M)$ as the set of valuations of $\mathcal{V}_A$ such that at least one globally active transition t is enabled. A globally active mode M for the agent A is called an ε-*lookahead mode*, iff*

$$\mathtt{Post}_\Phi(\mathcal{I}(M) \setminus \mathcal{G}, \varepsilon) \subseteq \mathcal{I}(M). \tag{1}$$

*An ε-**lookahead agent** $A = \langle SA, V\rangle$ is a closed agent of atomic sub-agents such that all its globally active modes are ε-lookahead modes.*

It can be shown that a Δ-lookahead agent can be faithfully simulated by a fixed step-size simulator with period Δ; that is, the fixed step-size simulator as defined above computes a trace at steps $0, 1, 2, 3, \dots$ that corresponds to a real trace of the Δ-lookahead agent at the time points $0, \Delta, 2\Delta, 3\Delta, \dots$ It is clear that *urgent switching*, that is taking a transition whenever a guard is enabled, guarantees non-blocking simulation using period Δ.

Theorem 1 *A non-blocking Δ-lookahead agent A can be faithfully simulated by a fixed step-size simulator with period Δ; that is for any admissible trace $r_A : \mathbb{R}_{\geq 0} \to \mathcal{X}_A$ of the Δ-lookahead agent A there exists a simulation trace f that can be computed by a fixed step-size simulator, such that $\forall k \in \mathbb{N} : r(k\Delta) = f(k)$.*

Although Theorem 1 guarantees faithful simulation of a Δ-lookahead agent, it does not mean that generated code embedded in a physical system will produce a faithful trace. One still needs to address issues such as timing delays introduced through sensing, computation and actuation. However, if we consider the case that we are trying to discretize an agent using a period Δ, when the agent is not a Δ-lookahead agent, it is apparent that even a fixed step-size simulator *cannot* guarantee a faithful simulation if condition (1) is not met for some reachable pair of guards set and invariant set. The condition (1) can be tested efficiently for systems with linear continuous dynamics using over-approximations [6]. Additionally, it should be noted that it is enough to prove that a mode is an ε-lookahead mode, if we can show that all pairs of active transition guard sets and invariant sets exhibit a big enough overlapping following an analogous definition. This modular proof technique is used in Section 5 to show feasibility of the code generation approach for Sony's robotic dog AIBO.

5 Case Study

To experiment our framework in a real system, we used Sony's four-legged robot, AIBO, as our experimental platform. The robot is a typical example of a hybrid system, consisting of analog devices for input and output and a digital control system to control the device. The control system is an embedded computer based on a MIPS microprocessor running at 384 MHz, and equipped with 32 MB main memory and 16 MB flash memory. The robot contains servo motors controlling position of the joints in the legs and the head, an LED display to simulate emotional expression, a speaker for voice, and input devices such as camera, microphones, and touch sensors. In this study, we are using the servo motors to make the dog walk and the touch sensors to detect ground contact. Applications can actuate motors so that the joints are positioned at a desired angle by sending a message containing the command. The system can process a vector of commands to the motors as frequently as once every eight milliseconds (i.e., $\Delta \geq 0.008$).

A typical program for the robot is coded as a `C++` class that contains methods invoked whenever a message arrives to the object. These methods typically implement a finite state machine that determines a behavioral mode of the system. In each state, it composes and sends a message containing the desired angle of each joint that is determined by the dynamics of the current state and the elapsed time since the last update of the joint. Since there is no explicit means to deal with time and dynamics in `C++`, code becomes easily awkward and hard to understand. In addition, the message-driven execution style tends to cause code for iterative jobs unstructured because control of execution leaves at every iteration. As the behavior becomes more complicated, code may become unmanageably huge for hand coding and debugging.

In contrast, CHARON supports hierarchical state machines, explicit time manipulation, differential equations describing dynamics, and static/dynamic analysis for correctness, mak-

```
void Walk::Ready(const OReadyEvent& event) {
    rgn = FindFreeRegion();

    charon->update(DELTA);

    subject[event.SbjIndex()]->SetData(rgn);
    subject[event.SbjIndex()]->NotifyObservers(rgn);
}

class joint : public var {
public:
    joint(int id) { this->id = id; }
    virtual void write(double value) {
        SetJointValue(rgn, id, value, newValue);
    }
private:
    int id;
};
```

Figure 7: Generated code.

ing it ideal for modeling event and time synchronized behavior of robots. Figure 7 shows an example message handling routine that utilizes the generated code replacing hand-coded if-then-else statements and joint value calculation routines. The method `update()` traverses active modes to trigger evaluation of equations, which assign a new value to the left-hand side variable. As explained in Section 3, the assignment operator is redefined such that it triggers a call to the method `write()`, which can be overridden by a customized function that finally calls platform specific APIs as shown in the figure. To demonstrate modularity of the generated code, we combined a sample program that comes with the official SDK for the robot (`www.aibo.com/openr`) with code generated from a CHARON model. The original sample program moves the legs to a set position and then controls the position of the head towards an object (pink ball). We added our walking model to this program, by slightly modifying the if-then-else statements of the original program such that it has an additional state that invokes the generated CHARON code. The dog then tracks the ball while walking. Interested readers are referred to our web site (`www.cis.upenn.edu/~rtg/codegen`), which contains full descriptions of this and several other examples, including motion pictures of the moving dog.

The error analysis as it has been described in section 4 can be used to show that our walking model can be faithfully simulated on the mathematical model of a fixed step-size simulator. Consider, for example, the sub-mode `UpDown(-1)`: The invariant `y>=y_limit; y<=y_upper_limit` and the guard `y<=y_lift` overlap assuming that `y_limit` $\leq$ `y_lift` $\leq$ `y_upper_limit`, while the dynamics are $\dot{y} = -3v$. Clearly, a fixed step-size simulator with time step Δ, such that $3v\Delta \leq$ `y_lift` $-$ `y_limit` will guarantee a faithful simulation. On the other hand, if `y_lift` $-$ `y_limit` is too small for a given Δ, even a fixed step-size simulator cannot guarantee a faithful discretization, when $3v\Delta >$ `y_lift` $-$ `y_limit`. Notice the inherent duality of this approach: A given time-step suggests a minimum guard-invariant overlap, while a given overlap suggests a maximum time-step. Also notice that if the model is deterministic (`y_lift` $-$ `y_limit` $= 0$ in this example), a fixed step-size simulator is unrealizable because it implies that Δ should be zero.

6 Conclusion

We presented a framework for automatic code generation of embedded software from high-level hybrid systems models and its implementation for a robotic platform. Our claim is that a model-based approach for embedded software development is essential for complex hybrid systems. Traditionally, software development for robot control includes a lot of hand-crafting to ensure correct timing and desired performance. Furthermore, debugging is more difficult because reasoning is done in the level of code, rather than in the level of the abstract model. In contrast, automatic code generation should result in faster development with higher quality code since it eliminates errors, which are often the result of manual coding. In addition, it is easy for the designer to concentrate on higher-level design issues, such as more efficient walking style. We spent only a few days to make the walking dog example work, even though we have very little experience of robot programming and walking is known to be one of the complex behaviors to program.

Several aspects of the code generation framework have been left for future work. 1) Ensuring adequate performance of generated code. The approach taken here is sufficient for controlling the dog, but more dynamic targets may require back-end optimization. 2) More systematic generation of glue code that connects platform-independent generated code to the target platform. Ideally, we need a platform specification language that will capture, in addition to the platform API, the resources of the platform such as the number and types of processors, available memory, communication bandwidth, etc. The code generation back-end can use this information to generate more efficient code and consider various implementation trade-offs. 3) Better understanding of the relation between continuous model and discretized code. In this paper, we considered errors under the assumption that sensing, computation and actuation are performed instantaneously at the beginning of each period. For an execution model with total period Δ that explicitly includes timing delays we need to require a (2Δ)-lookahead hybrid system. Assuming that sensing, computation and actuation can be performed within Δ, the system reacts to inputs in at most 2Δ time-units. To illustrate this, assume a non-zero time delay δ_S for sensing. The sensed inputs representing the values at some time $k\Delta$ are available at time $k\Delta + \delta_S$ only. If a computation delay of δ_C is assumed, the system can react only at time $k\Delta + \delta_S + \delta_C$. If no guard is enabled at time $k\Delta$, this implies that the system could react to a transition only at time $(k+1)\Delta+\delta_S+\delta_C$. Given that $\delta_S + \delta_C \leq \Delta$, to be safe, we require a (2Δ)-lookahead system.[1] 4) Multi-threaded and multi-processor code generation. We described generation of single-threaded implementations. We have also implemented multi-threaded implementation with tight synchronization between threads. It is interesting to consider generation of multi-threaded code with different peri-

[1]A similar (2Δ) bound has been derived for the implementability of switching behaviors on PLCs (programmable logic controllers), where Δ is the upper bound on the length of a PLC cycle consisting of polling inputs, computation and delivery of outputs using a timed-automata variant framework [12].

ods for different tasks. This will require us to explore the scheduling of agent threads and more sophisticated error analysis. Multi-processor code generation is a more long-term goal and will require us to map shared variables of the model into message passing and consider communication delays.

References

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

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

[3] 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.

[4] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III: Verification and Control.* LNCS 1066. Springer-Verlag, 1996. Proceedings of the Third International Workshop.

[5] P. Antsaklis and A. Michel. *Linear Systems.* McGraw Hill, 1997.

[6] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control, Third International Workshop*, LNCS 1790, pages 21–31. 2000.

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

[8] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison Wesley, 1997.

[9] G.C. Buttazo. *Hard real-time computing systems: Predictable scheduling algorithms and applications.* Kluwer Academic Publishers, 1997.

[10] C.T. Chen. *Linear System Theory and Design.* Oxford University Press, 1999. 3rd Edition.

[11] 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.

[12] H. Dierks. PLC-automata: A new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, December 2000.

[13] J. Eker, J. Janneck, E.A. 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.

[14] J. Esposito, V. Kumar, and G.J. Pappas. Accurate event detection for simulating hybrid systems. In *Hybrid Systems: Computation and Control, Fourth International Workshop*, LNCS 2034, pages 204–217, 2001.

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

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

[17] B. Hengst, D. Ibbotson, S. B. Pham, and C. Sammut. Omnidirectional locomotion for quadruped robots. In *RoboCup 2001: Robot soccer world cup V*, LNAI 2377, pages 368–373. Springer-Verlag, 2002.

[18] T. Henzinger and C. Kirsch, editors. *Embedded Software, First International Workshop.* LNCS 2211. Springer, 2001.

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

[20] T.A. Henzinger and C.M. Kirsch. The embedded machine: Predictable, portable, real-time code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 315–326, 2002.

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

[22] E.A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18–26, September 2000.

[23] N. Lynch and B.H. Krogh, editors. *Hybrid Systems: Computation and Control.* LNCS 1790. Springer, 2000.

[24] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600, pages 447–484. Springer-Verlag, 1991.

[25] B. Selic, G. Gullekson, and P.T. Ward. *Real-time object oriented modeling and design.* J. Wiley, 1994.

APPENDIX: Code Generation Algorithm

Algorithm 1 CodeGen ($M = \langle E, X, V, SM, Cons, T\rangle$)

```
GenStmt("class", $M$, ": public mode");
/* sub-modes */
for all $m \in SM$ do
  GenStmt("mode *", $m$);
  if $m$ is not visited then
    CodeGen ($m$);    /* recursive call */
/* variables */
for all $v \in V$ do
  if $v \in V_l$ then
    GenStmt("var&", $v$);
  else
    GenStmt("var", $v$);
/* differential equations */
GenStmt("void diff(delta)");
for all $D_x \in Cons$ do
  /* integration */
  GenStmt($x$, "+=", $f_{D_x}$, "* delta");
/* algebraic equations */
GenStmt("int alge(int id)");
GenStmt("switch (id)");
for all $A_x \in Cons$ do
  GenStmt("case", $A_x.id$)
  GenStmt($x$, "=", $f_{A_x}$); /* assignment */
  GenStmt("return 0")
GenStmt("return -1")
/* invariants */
GenStmt("bool inv()");
for all $I_x \in Cons$ do
  GenStmt("if (assert(", $f_{I_x}$, "))"
  GenStmt("return false");
GenStmt("return true");
/* transitions and control points */
CodeGenTrans($T$);    /* Algorithm 2 */
CodeGenCtrlPoint($E, X$);    /* Algorithm 3 */
```

Algorithm 2 CodeGenTrans (T)

```
GenStmt("mode *trans(mode *h)");
for all $t = \langle m.x \xrightarrow{g|\alpha} m'.e\rangle \in T$ do
  /* internal transition from $m.x$ to $m'.e$ */
  GenStmt("if (h ==", $m$, "&&", $m.x()$, "&&", $g$, ")");
  for all $a_x \in \alpha$ do
    GenStmt($x$ "=" $f_{a_x}$); /* discrete actions */
  for all $a_x \in \alpha$ do
    GenStmt($x$,".sync()"); /* synchronization */
  /* update dependency links */
  for all $t' = \langle m.x \xrightarrow{g|\alpha} m'.e\rangle \in T$ do
    for all $a_x \in \alpha$ do
      GenStmt($x$, ".unlink(", $M$, $t'.id$, ")")
  GenStmt($m'.e$, "()"); /* entry transition */
  for all $t' = \langle m'.x \xrightarrow{g|\alpha} m''.e\rangle \in T$ do
    for all $a_x \in \alpha$ do
      GenStmt($x$, ".link(", $M$, $t'.id$, ")")
  GenStmt("return", $m'$);

for all $t = \langle m.x \xrightarrow{g|\alpha} M.x'\rangle \in T$ do
  /* exit transition from $m.x$ to $M.x'$ */
  GenStmt(if (h ==", $m$, "&&", $m.x$, "()", "&&", $g$, ")");
  for all $a_x \in \alpha$ do
    GenStmt($x_a$, "=", $f_{a_x}$); /* discrete actions */
  for all $a_x \in \alpha$ do
    GenStmt($x$,".sync()"); /* synchronization */
  GenStmt("exitCode =", $m.x$); /* set exit code */
  /* remove dependency links */
  for all $t' = \langle m'.x \xrightarrow{g|\alpha} m''.e\rangle \in T$ do
    for all $a_x \in \alpha$ do
      GenStmt($x$, ".unlink(", $M$, $t'.id$, ")")
  for all $A_x \in Cons$ do
    GenStmt($x$, ".unlink(", $M$, $A_x.id$, ")")
  GenStmt("return null");
```

Algorithm 3 CodeGenCtrlPoint (E, X)

```
/* entry point */
for all $e \in E$ do
  GenStmt("void", $e$, "()");
  /* reset exit code */
  GenStmt("exitCode = null");
  for all $t = \langle m.e \xrightarrow{g|\alpha} m'.e' \rangle \in T$ do
    GenStmt("if (", $g$, ")");
    for all $a \in \alpha$ do
      GenStmt($a.LHS$, "=", $a.RHS$);
    GenStmt($m'.e'$, "()"); /* destination entry */
    /* update dependency links */
    for all $A_x \in Cons$ do
      GenStmt($x$, ".link(", $M$, $A_x.id$, ")")
    for all $t' = \langle m'.x \xrightarrow{g|\alpha} m''.e \rangle \in T$ do
      for all $a_x \in \alpha$ do
        GenStmt($x$, ".link(", $M$, $t'.id$, ")")
    GenStmt("return");

/* exit point */
for all $x \in X$ do
  GenStmt("bool $x$()");
  /* test if the mode exited through $x$ */
  GenStmt("return exitCode ==", $x$);
```