May 2003

# Modular Code Generation from Hybrid Automata based on Data Dependency

Jesung Kim
*University of Pennsylvania*

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

# Modular Code Generation from Hybrid Automata based on Data Dependency

**Abstract**
Model-based automatic code generation is a process of converting abstract models into concrete implementations in the form of a program written in a high-level programming language. The process consists of two steps, first translating the primitives of the model into (approximately) equivalent implementations, and then scheduling the implementations of primitives according to the data dependency inherent in the model. When the model is based on hybrid automata that combine continuous dynamics with a finite state machine, the data dependency must be viewed in two aspects: continuous and discrete. Continuous data dependency is present between mathematical equations modeling timecontinuous behavior of the system. On the other hand, discrete data dependency is present between guarded transitions that instantaneously change the continuous behavior of the system. While discrete data dependency has been studied in the context of code generation from modeling languages with synchronous semantics (e.g., ESTEREL), there has been no prior work that addresses both kinds of dependency in a single framework. In this paper, we propose a code generation framework for hybrid automata which deals with continuous and discrete data dependency. We also propose techniques for generating modular code that retains modularity of the original model. The framework has been implemented based on the hybrid system modeling language CHARON, and experimented with Sony's robot platform AIBO.

**Comments**
Copyright 2003 IEEE. Reprinted from *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, pages 160-168.
Publisher URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=27075&page=1

# Modular Code Generation from Hybrid Automata based on Data Dependency*

Jesung Kim and Insup Lee
*Department of Computer and Information Science*
*University of Pennsylvania*
*jesung@saul.cis.upenn.edu, lee@cis.upenn.edu*

## Abstract

*Model-based automatic code generation is a process of converting abstract models into concrete implementations in the form of a program written in a high-level programming language. The process consists of two steps, first translating the primitives of the model into (approximately) equivalent implementations, and then scheduling the implementations of primitives according to the data dependency inherent in the model. When the model is based on hybrid automata that combine continuous dynamics with a finite state machine, the data dependency must be viewed in two aspects: continuous and discrete. Continuous data dependency is present between mathematical equations modeling time-continuous behavior of the system. On the other hand, discrete data dependency is present between guarded transitions that instantaneously change the continuous behavior of the system. While discrete data dependency has been studied in the context of code generation from modeling languages with synchronous semantics (e.g.,* ESTEREL*), there has been no prior work that addresses both kinds of dependency in a single framework. In this paper, we propose a code generation framework for hybrid automata which deals with continuous and discrete data dependency. We also propose techniques for generating modular code that retains modularity of the original model. The framework has been implemented based on the hybrid system modeling language* CHARON*, and experimented with Sony's robot platform* AIBO*.*

## 1 Introduction

Developing software for real-time embedded systems requires fundamentally different approach due to unique characteristics of the system that are not common in general purpose computers [8, 9]. First, they usually interact with the physical world and are based on mathematical models. Second, they are in many cases safety critical, making high assurance of correctness essential. Model-based automatic code generation is promising in this domain since design can be formally verified in the level of models using formal verification techniques, and implementation can be free from program errors due to manual coding.

A computer system interacting with an analog environment can be best modeled by hybrid automata [1, 10]. Hybrid automata combine the traditional finite state machine-based model of discrete control with continuous dynamics of the physical world. In hybrid automata, a set of differential equations and algebraic equations specify dynamics of the system, and the finite state machine specifies discrete change of dynamics of the system from one set of equations to another. This model is useful for describing systems that interact with the physical world where the input and output are continuous trajectories, rather than discrete samples, of variables. For example, the input to a robot tracking an object is a trajectory of the position of the object, and the output is a trajectory of the position of the head that minimizes the angle between the direction towards the object and the line of sight. Such trajectories can be conveniently modeled by differential equations that specify evolution of variables with respect to time. That is, differential equations reflect stimuli and reactions of the model. This concept is generally not well supported in other languages based on discrete events. Thus, developing such systems using traditional programming languages becomes unnecessarily complicated and hard to validate.

Automatic code generation rectifies the situation by converting mathematical models automatically into programs written in a system-level programming language such as C. Our code generation process consists of two

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

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

phases. First, each element of hybrid automata is converted into a piece of code. The challenge in this phase is to discretize time-continuous actions such that they can be executed by a digital computer. Second, the pieces of code generated in the first phase are combined into a single program. This phase resolves concurrency inherent in the model by interleaving the code for each element at the granularity of the period determined in the first phase. Here, the challenge is to determine an interleaving that is consistent to the mathematical model. In our approach, we address this issue by generalizing the concept of data dependency to the continuous-time domain and interleaving the code according to a data dependency order. We divide the data dependency of hybrid automata into continuous part and discrete part of the model and address separately. This approach is based on the semantics of hybrid automata where discrete actions occur instantaneously without any observable time passage. This means that time is conceptually stopped while discrete actions are performed. Time advances only while continuous activities change the state. Therefore, we can analyze dependency between discrete actions independently of the continuous part of the model, and vice versa. Once the dependency is resolved separately and the code is generated accordingly, we can simply concatenate both.

Our automatic code generation technique supports modularity of the generated code. Generating modular code is crucial when the original model consists of a large collection of hierarchical or concurrent components. The generated code captures modularity of the original model in two senses. First, the code consists of components, each of which can be separately compiled for a different target platform. Second, each component of the generated code is valid even when other part of the model is modified. These properties facilitate reuse of components of the generated code in different application context and target hardware platforms.

Our code generation framework is implemented in the context of the hybrid system modeling language CHARON [2], and tested in Sony's robot platform AIBO. We have experimented our framework with numerous examples, including the modeling of robot's behavior presented in Section 2.

**Related works.** Commercial modeling tools such as Simulink and RationalRose support automatic code generation, but it is not formally described and the consistency between model and code is not addressed explicitly. Modeling languages for reactive systems such as ESTEREL [4], LUSTRE [6], and STATECHART [7] also support automatic code generation, but they do not support modeling of continuous dynamics.

## 2 Modeling

A hybrid automaton [1, 10] consists of locations each of which has a set of differential equations and algebraic equations, and transitions between locations. When a hybrid automaton stays in a location, variables are updated continuously according to the differential equations and algebraic equations of the location, until a transition is taken or the invariant condition of the location is violated. A transition can be taken whenever the associated condition (guard) is true. When a transition is taken from one location to another, differential equations and algebraic equations belonging to the destination become effective immediately, and the variables continuously evolve according to those new equations. Transitions may have optional assignments to variables that are performed instantaneously when the transition is taken. Formally, we define hybrid automata as follows.

**Definition 1 (Hybrid automata)** *A hybrid automaton $H$ is a tuple $\langle S, V, I_0, T, G, W, D, A, I, C\rangle$, where*

- *$S$ is a set of locations.*
- *$V$ is a set of real variables.*
- *$I_0$ is the initial state, which is a tuple $\langle s_0, V_0\rangle$, where $s_0 \in S$ is the initial location and $V_0 : V \to \mathbb{R}$ is a function that assigns the initial value of the variables in $V$, where $\mathbb{R}$ is the set of real numbers.*
- *$T \subseteq S \times S$ is a set of transitions between two locations. An element $(s, s') \in T$ is said active when $(s'', s) \in T$ is the last transition that was taken, or $s = s_0$.*
- *$G$ assigns to each $(s, s') \in T$ a guard, denoted as $G(s, s')$, which is a predicate over $V \times \mathbb{R}$. A transition $(s, s') \in T$ is said enabled when $G(s, s')$ is true.*
- *$W$ assigns to each $(s, s') \in T$ a reset, denoted as $W(s, s')$, which is a function from $V_{(s,s')}$ to $\mathbb{R}$, where $V_{(s,s')} \subseteq V$. $G$ and $W$ collectively define discrete behavior of $H$. A reset changes the value of variables in $V_{(s,s')}$ to $W(s, s')$ instantaneously when $(s, s')$ is taken.*
- *$D$ is a set of differential equations in the form of $\dot{x} = f(X)$, where $x \in V$, $X = \{x_1, x_2, ..., x_n\}$ is a vector of variables $x_i \in V$, and $f(X)$ is the first derivative of $x$ with respect to time (i.e., $dx/dt = f(X)$).*
- *$A$ is a set of algebraic equations in the form of $u = g(X)$, where $u \in V$, $X$ is a vector of variables in $V \setminus \{u\}$.*

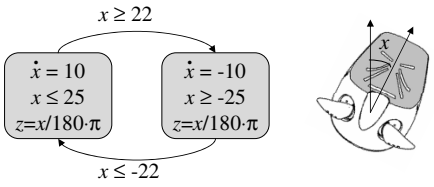


**Figure 1. Hybrid automaton modeling a robot dog panning the head.**

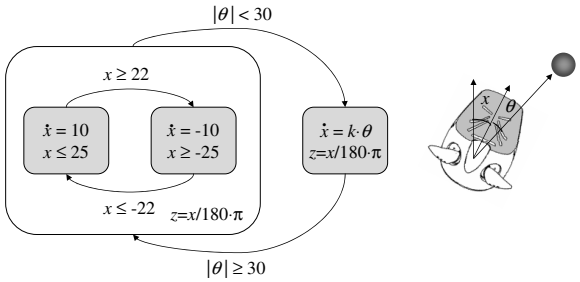


**Figure 2. Hierarchical hybrid automaton modeling a robot dog tracking an object.**

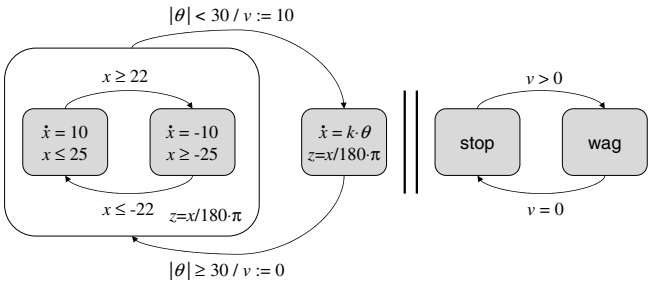


**Figure 3. Concurrent hierarchical hybrid automaton modeling a robot dog wagging the tail.**

- *$I$ is a set of predicates over $V \times \mathbb{R}$ that are the invariant conditions.*
- *$C$ assigns to each $s \in S$ constraints, a subset of $D \cup A \cup I$ that defines continuous behavior of the location. An element in $C(s)$ is said active when $(s', s)$ is the last transition that was taken, or $s = s_0$.*

Hybrid automata have been widely used for modeling and simulating control systems consisting of multiple control laws. In such a system, differential equations and finite state machines are essential for specifying transition of control laws. Hybrid automata are also very useful for programming robots, where one of the main tasks is to update the angle of each joint periodically to simulate a continuous action. For example, Figure 1 shows a simple hybrid automaton modeling a robot dog panning its head. It consists of two locations, each of which specifies constant increase/decrease ($\pm 10\,^\circ/s$) 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 location (and hence dynamics) when the head is moved beyond a certain position ($\pm 22\,^\circ$). Note that in hybrid automata transitions can be taken any time while the guard is true (i.e., the time when the transition is taken can be nondeterministic). The invariant of each location specifies that the switch should occur before the head moves beyond its allowed range ($x \leq 25$ and $x \geq -25$). Each location also has an algebraic equation that translates the degree to the radian ($z = x/180 \cdot \pi$). Once the automaton is compiled into a programming language and the variable $z$ is mapped to a hardware device or a device driver that actually controls the position of the head, the head will move as expected from the model.

Hybrid automata can be composed hierarchically and/or concurrently to model more complex systems. In hierarchical hybrid automata, a location can be a hybrid automaton, or another hierarchical hybrid automaton. Figure 2 shows a hierarchical hybrid automaton modeling a robot dog tracking an object. The model assumes continuously updated input variable $\theta$ that indicates the position of the object relative to the head. When $\theta$ is within a certain threshold ($|\theta| < 30$), the robot attempts to move the head towards the object, as modeled by a differential equation $\dot{x} = k \cdot \theta$ in the rightmost location of the model. However, if $\theta$ is beyond the threshold, the robot gives up tracking the object, and continues panning the head. Note that the same model in Figure 1 is reused in modeling movement of the head.

Figure 3 shows concurrent hierarchical hybrid automata modeling a robot dog wagging its tail when it detects an object. It simply combines the automaton shown in Figure 2 with a new automaton for wagging the tail, which is very similar to the model shown in Figure 1 and the details are omitted. The automaton shown in Figure 2 is slightly modified such that it assigns to the variable $v$ a value greater than zero when the dog detects the object. This triggers wagging of the tail.

A hybrid automaton can be identified by a set of possible traces of the variables. A trace $R$ of a hybrid automaton is one possible trajectory of values of variables that satisfies constraints $C(s)$ when the automaton stays in a location $s$ at time $t$, and has a discrete jump to $W(s, s')$ when there is a transition $(s, s')$ at time $t$. We denote the values of variables in a trace $R$ at time $t$ as $R(t)$. In this paper, we translate a hybrid automaton into

a program whose trace is sampling of some trace of the original automaton at discrete times. We define a *discretization* of a hybrid automaton as a basis for formal definition of the automatically generated code.

**Definition 2 (Discretization of hybrid automata)** *A hybrid automaton is said discretizable with respect to time step $h$, if there exists a trace $R$ such that $R(t)$ is continuous for all $t \in \mathbb{R} \setminus \{\tau_i | \tau_i = i * h, i = 0, 1, 2, ...\}$. A discretization of a discretizable hybrid automaton with respect to time step $h$ is an automaton whose trace is a sequence $\langle V_0, V_1, V_2, ...\rangle$ of values of variables, such that there exists a trace $R$ of the hybrid automaton satisfying $V_i = R(i \times h)$ for all $i \geq 0$.*

Informally, in a discretizable hybrid automaton, it is possible that transitions are taken only at times multiple of $h$. A discretization of such a hybrid automaton assigns to each variable a value that is the same as the original hybrid automaton at every time $h$. In the remainder of this paper, we will describe our framework for generating code that is equivalent to the automaton of Definition 2.

## 3 Translation of primitives

We now explain how each primitive of hybrid automata can be translated into a piece of code that discretizes the primitive defined in the continuous-time domain. We first present translation of continuous actions specified by differential equations and algebraic equations. We then explain translation of discrete actions specified by guarded transitions.

### 3.1 Continuous actions

A differential equation in 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. In this study, we consider the fourth-order Runge-Kutta method that averages a number of approximate values [11]. A general form of the Runge-Kutta method for a differential equation $\dot{x} = f(x)$ with a step size $h$ is as follows.

$$k_1 = f(x_t) \cdot h \quad (1)$$
$$k_2 = f(x_t + k_1/2) \cdot h \quad (2)$$
$$k_3 = f(x_t + k_2/2) \cdot h \quad (3)$$
$$k_4 = f(x_t + k_3) \cdot h \quad (4)$$
$$x_{t+h} = x_t + k_1/6 + k_2/3 + k_3/3 + k_4/6 \quad (5)$$

```
double diff_x(double h)        /* x' = 2x */
    /* returns the value of x at time t+h */
    k1 = 2*x*h;
    k2 = 2*(x+k1/2)*h;
    k3 = 2*(x+k2/2)*h;
    k4 = 2*(x+k3)*h;
    return x + k1/6+k2/3+k3/3+k4/6;
}
```

**Figure 4. Translation of a single-variable differential equation**

A translation of a differential equation $\dot{x} = 2x$ using this method is shown in Figure 4. Continuous change of variable $x$ due to the differential equation can be simulated by invoking the function periodically at every time step $h$. Note that this equation involves only one variable, making it self-contained. In general, dynamics of a system can be specified by a set of differential equations that have dependency. For example, let's consider a model of an object that moves at the acceleration given by $a$. The position $x$ of the object can be modeled as a system of two differential equations: $\dot{x} = v$ and $\dot{v} = a$. The Runge-Kutta method for a set of differential equations has the same form except that each variable is a vector. That is, $k_2 = f(x_t + k_1/2) \cdot h$ can be interpreted as $k_2^j = f_j(x_t^1 + k_1^1/2, x_t^2 + k_1^2/2, ..., x_t^n + k_1^n/2) \cdot h$ for all $j$, and so on.

In our approach, differential equations are translated in two different ways depending on their *dependency* relations. We define that differential equation $d_i \in D$ has dependency on another differential equation $d_j \in D$, denoted as $d_i \rightarrow d_j$, when the right-hand side of $d_i$ contains the variable at the left-hand side of $d_j$. Data dependency is called *cyclic* if $d_1 \rightarrow d_2$, $d_2 \rightarrow d_3$, ..., $d_{n-1} \rightarrow d_n$, and $d_n \rightarrow d_1$, for some differential equations $d_1, d_2, ..., d_n \in D$ ($n \geq 2$).

When differential equations have cyclic dependency, they are translated based on the vectorized Runge-Kutta method. Figure 5 shows code for two differential equations $\dot{x} = f_x(x, y)$ and $\dot{y} = f_y(x, y)$ based on the vectorized Runge-Kutta method. A drawback of the code is that differential equations are tightly coupled. This means that different code needs to be generated for each possible combination of differential equations that can be active simultaneously. We can decompose the code by encapsulating each intermediate step as a function and cross-referencing when the result of the intermediate step of another differential equation is needed. Figure 6 shows an extra implementation. However, this code introduces overheads due to frequent and redundant function calls.

We can get more efficient yet modular code, if data

```
void diff_xy(double h) {
    k1_x = h*fx(x, y);  /* x' = fx(x, y) */
    k1_y = h*fy(x, y);  /* y' = fy(x, y) */
    k2_x = h*fx(x+k1_x/2, y+k1_y/2);
    k2_y = h*fy(x+k1_x/2, y+k1_y/2);
    k3_x = h*fx(x+k2_x/2, y+k2_y/2);
    k3_y = h*fy(x+k2_x/2, y+k2_y/2);
    k4_x = h*fx(x+k3_x, y+k3_y);
    k4_y = h*fy(x+k3_x, y+k3_y);
    x += k1_x/6+k2_x/3+k3_x/3+k4_x/6;
    y += k1_y/6+k2_y/3+k3_y/3+k4_y/6;
}
```

**Figure 5. Non-modular implementation of the Runge-Kutta method.**

dependency between differential equations is not cyclic. The idea behind it is that, if data dependency is not cyclic, numerical integration of each differential equation can be performed separately one by one, without interleaving intermediate steps of different equations as implied by the original formula of the Runge-Kutta method. That is, if $d_i \rightarrow d_j$, integration of $d_j$ does not require the results of integration of $d_i$, and thus integrations of two differential equations can be performed sequentially. Thus, the tightly coupled code shown in Figure 5 can be decomposed into a component for each differential equation without overhead for redundant function calls of the code shown in Figure 6. We define a *discretization* of a differential equation with respect to step size $h$ as a procedure that produces the value of the left-hand side variable at time $t + h$ and its intermediate values (i.e., $k_1, k_2, k_3, k_4$), from the values of variables at the right-hand side at time $t$ and their intermediate values.

For example, Figure 7 shows modularized code for a pair of differential equations $\dot{x} = v$ and $\dot{v} = a$. Note that the code does not assume anything about dynamics of $a$. This makes the code for $\dot{v} = a$ valid regardless of dynamics of $a$, provided that the code for $a$ is executed before $v$. For example, when another concurrent automaton constrains $a$ by either $\dot{a} = 1$ or $\dot{a} = 2a$ depending on the state of the system, the code for $\dot{v} = a$ is valid provided that it is executed after the code for $a$ is executed.

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, a *discretization* of algebraic equation $y = g(X)$ is simply an assignment of the form $y := g(X)$.

We also define data dependency between algebraic equations $a \in A$. Algebraic equation $a_i$ has depen-

```
void diff_x(double h) {
    return x + (k1_x(h)/6 + k2_x(h)/3
              + k3_x(h)/3 + k4_x(h)/6);
}
double k1_x(double h) {
    return h*fx(x, y);
}
double k2_x(double h) {
    return h*fx(x+k1_x(h)/2, y+k1_y(h)/2);
}
double k3_x(double h) {
    return h*fx(x+k2_x(h)/2, y+k2_y(h)/2);
}
double k4_x(double h) {
    return h*fx(x+k3_x(h), y+k3_y(h));
}

void diff_y(double h) {
    return y + (k1_y(h)/6 + k2_y(h)/3
              + k3_y(h)/3 + k4_y(h)/6);
}
double k1_y(double h) {
    return h*fy(x, y);
}
double k2_y(double h) {
    return h*fy(x+k1_x(h)/2, y+k1_y(h)/2);
}
double k3_y(double h) {
    return h*fy(x+k2_x(h)/2, y+k2_y(h)/2);
}
double k4_y(double h) {
    return h*fy(x+k3_x(h), y+k3_y(h));
}
```

**Figure 6. Modular implementation of the Runge-Kutta method.**

```
double diff_v(double h) {  /* v' = a */
    v_k1 = a*h;
    v_k2 = (a+a_k1/2)*h;
    v_k3 = (a+a_k2/2)*h;
    v_k4 = (a+a_k3)*h;
    return v+(v_k1/6+v_k2/3+v_k3/3+v_k4/6);
}

double diff_x(double h) {  /* x' = v */
    x_k1 = v*h;
    x_k2 = (v+v_k1/2)*h;
    x_k3 = (v+v_k2/2)*h;
    x_k4 = (v+v_k3)*h;
    return x + (x_k1/6 + x_k2/3 + x_k3/3 + x_k4/6);
}

double diff_vx(double h) {
    v_tmp = diff_v(h);
    x_tmp = diff_x(h);
    v = v_tmp;
    x = x_tmp;
}
```

**Figure 7. Dependency-based implementation of the Runge-Kutta method.**

dency on another algebraic equation $a_j \in A$, denoted as $a_i \rightarrow a_j$, when the right-hand side of $a_i$ includes the left-hand side variable of $a_j$. That is, algebraic equation $z = g(Y)$ has a dependency on another algebraic equation $y = f(X)$ if $Y$ contains $y$. By executing discretizations of algebraic equations before executing discretizations of algebraic equations that have dependency on the former, equalities between variables inferred by the algebraic equations in the model can be satisfied. That is, the code satisfies $z = g(f(x))$ when algebraic equations $z = g(y)$ and $y = f(x)$ are active. Note that the equality may not be satisfied if the execution order is reversed. In this paper, we do not consider hybrid automata that have cyclic dependency in algebraic equations.

### 3.2 Discrete actions

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

We translate a transition into an if-then statement where the guard becomes the if-condition and the statement block contains the assignments, along with an additional statement that updates a variable storing the current location. Such a variable is needed to test whether a differential/algebraic equation is currently active. Conceptually, the if-block should be executed continuously (i.e., infinitely frequently), since continuous variables can be updated at any time. In the generated code, however, variables are updated synchronous to execution of discretizations of differential equations and algebraic equations. Therefore, the code for transitions is executed after continuous actions are performed at every step. Formally, we define that $p \rightarrow q$ if $p \in T$ and $q \in A \cup D$. Note that such a discretization guarantees that a transition is taken in a delay less than $h$ after it is enabled.

A transition may enable another transition through a discrete action as in the model shown in Figure 3. While non-deterministically specified hybrid automata allow us to leave such transition not taken until the next time step, our code generator enforces such transitions occur synchronously in the same step. The motivation is that we can eliminate delays of synchronous transitions if they are evaluated in a dependency order. We define a transition $(s_1, s_1')$ has dependency on another transition $(s_2, s_2')$ if $G(s_1, s_1')$ is true over $W(s_2, s_2')$. For example, in the model shown in Figure 3, transitions in the right automaton have dependency on transitions in the left automaton. If transitions in the left automaton are evaluated before transitions in the right automaton, wagging of the tail can start simultaneously when the dog starts tracking an object. Note that wagging of the tail can be delayed by $h$ when the evaluation order is reversed.

## 4 Scheduling

We now describe a process of combining discretizations of primitives explained in the previous section into a single program. The resulting program is a single-threaded code that executes discretizations of active primitives sequentially. An execution order, i.e., a schedule, is determined based on data dependency defined in the previous section.

Formally, given a set $P$ of primitives, i.e., $P \subseteq A \cup D \cup T$, a *schedule* of $P$ is a sequence $\langle p_1, p_2, ..., p_n \rangle$ of all primitives $p_i \in P$, i.e., a total order on $P$. *Scheduling* is a process of determining a schedule. Scheduling can be done either statically at code generation time or dynamically at run time. In static scheduling, a schedule determined at code generation time, called a *static schedule*, is used throughout execution, and the execution order never changes. On the other hand, in dynamic scheduling, schedules are determined at run time, and thus the execution order can be changed.

We enforce that a schedule be *consistent* and *complete*. A schedule $\langle p_1, p_2, ..., p_n \rangle$ is *consistent (to data dependency)* if $p_i \rightarrow p_j$ implies $i > j$ for all $i, j \leq n$. Let $P_\tau$ be a set of active primitives at time $\tau$. A schedule is said *complete* at time $\tau$ when it includes all $p \in P_\tau$. A schedule is complete if it is complete for all times.

A static schedule that is consistent exists when data dependency is not cyclic. If data dependency is not cyclic, the transitive closure of the data dependency relation defines a partial order on $A \cup D \cup T$. Any total order on $A \cup D \cup T$ that subsumes the partial order is consistent to data dependency, and can be used as a consistent static schedule. Such a schedule is complete since it includes all primitives.

Note that a static schedule that is consistent and complete is possible only when data dependency is not cyclic. In other case, dynamic scheduling is used. Given a set $P_\tau$ of active primitives at time $\tau$, a schedule that is consistent exists if the data dependency relation on $P_\tau$ is not cyclic. A schedule of $P_\tau$ is complete by definition. A consistent schedule can be obtained at time $\tau$ by

determining a total order that subsumes the partial order defined by the transitive closure of the data dependency relation on $P_\tau$. A schedule that is consistent exists at all times if the data dependency relation on $P_\tau$ is not cyclic for all $\tau$.

On the other hand, if the data dependency relation on $P_\tau$ is not cyclic for some time $\tau$, a consistent schedule does not exist.

We now define code of a hybrid automaton.

**Definition 3 (Code)** *Code of a hybrid automaton is a program that executes discretizations of active primitives at time $\tau_i = i \times h, i = 0, 1, 2, ...$, such that*

- *execution starts from $I_0$ at time $\tau_0$ and continuous until the invariant is violated.*
- *there is a procedure that determines a set $P_{\tau_i}$ of active primitives at time $\tau_i$.*
- *there is a procedure that determines a schedule (i.e., a run-time scheduler) in the case of dynamic scheduling.*

*A trace of code is a sequence $\langle V_0, V_1, V_2, ... \rangle$, where $V_0$ is the initial values and $V_i, i > 0$ is the values produced by execution of discretizations at time $\tau_{i-1}$.*

Code based on static scheduling executes discretizations of primitives that are active in an order given by a static schedule. Since an execution order is determined at the code generation phase, a schedule can be encoded as a sequence of code. Thus, there is no run time overhead associated with scheduling. On the other hand, code based on dynamic scheduling determines execution order at run time. To support dynamic scheduling, it is required that the generated code be modular since components can be reordered at run time, It also requires a run-time scheduler that determines a schedule based on a data dependency relation on currently active primitives at every step. This dynamic scheduling can be implemented by encapsulating each component in a function and maintaining the pointers to the functions in an array. Figure 8 compares code based on static scheduling and code based on dynamic scheduling.

Now we present the properties of the code formally.

**Theorem 1** *The trace of the code of a discretizable hybrid automaton is equal to the trace of some discretization of the hybrid automaton, if*

1. *The schedules is consistent and complete at each step.*
2. *The code does not violate the invariant.*

```
code_static() {
   while (1) {
       if (active_d1()) diff_1();
       if (active_d2()) diff_2();
       ...
       if (active_a1()) alge_1();
       if (active_a2()) alge_2();
       ...
       if (active_t1()) trans_1();
       if (active_t1()) trans_2();
       ...
       assert(inv_1() && inv_2() && ...);
   }
}
```

(a) Static scheduling

```
code_dynamic() {
    while (1) {
        schedule(f);
        for (i = 0; i < N; i++) {
            (*f[i])();
        }
        assert(inv_1() && inv_2() && ...);
    }
}
```

(b) Dynamic scheduling

**Figure 8. Generated code skeleton.**

3. *The discretizations of differential equations are precise.*
4. *Execution of the step at time $\tau_i$ finishes before $\tau_{i+1}$.*

**Proof** Let $\langle V_0, V_1, V_2, ... \rangle$ be the trace of the code. And let $R_i$ be the $i$'th element of the trace $R$ of some discretization of the hybrid automaton. Initially, for all discretization, $V_0 = R_0$ by definition. Let's assume that there exists discretizations that satisfy $V_i = R_i$ for all $i \leq k$. Then among such discretizations, there exists discretizations that also satisfy $V_i = R_i$ for all $i \leq k+1$ because (1) discretizations of differential equations produce the same values as in $R_{k+1}$ by the assumptions 1 and 3, and because (2) discretizations of algebraic equations produce the same values as in $R_{k+1}$ by the assumption 1. In addition, $V_{k+1} = R_{k+1}$ and the assumption 3 imply that transitions enabled in the code are also enabled in the hybrid automaton, and thus any transition occurred in the code is possible in the hybrid automaton. And instantaneous transition semantics is preserved since the values used to test guards and the values produced by discrete actions both represent the values at time $\tau_{i+1}$. Finally, the assumption 4 indicates that the values at $\tau_{i+1}$ are readily available at $\tau_{i+1}$, satisfying timely production of values. □

Note that the code generated from the model shown in Figure 1 satisfies all three conditions given by The-

orem 1, since data dependency is not cyclic, differential equations are zero-order and thus can be solved precisely, and transitions can be taken before the invariant is violated. However, in general, our code generation framework guarantees only the first condition of Theorem 1. That is, even if we can generate the code that is consistent to the data dependency when there is no cyclic dependency, it is not guaranteed that the code produces the exactly same values of variables as the original mathematical model, since discretization of differential equations introduces numerical errors and discretization of transitions may cause a transition miss. Note that the latter two conditions are the matter of robustness of the model against discretization, and should be analyzed at the level of the model possibly with feedback information from the code (c.f. [3, 5]). For example, the model can be analyzed whether guards are enabled longer than a certain duration before the invariant is violated [3]. The effect of numerical errors can also be analyzed if the error bound of the numerical method is given [5].

## 5 Implementation

We have implemented our code generation framework in the context of CHARON, the hybrid system modeling language [2]. Using CHARON, designers can formally describe sophisticated behaviors of hybrid systems using a language construct such as modes and agents. To experiment our framework in a real system, we used Sony's four-legged robot, AIBO, as the target platform (http://www.aibo.com). The robot is a typical example of hybrid systems, consisting of analog devices for inputs and outputs and a digital control system to control the devices. 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 operating system is Sony's proprietary object-oriented real-time operating system known as Aperios. There is an additional layer of software called OPEN-R that hides system-level details.

Our code generator is implemented on top of the parser of CHARON. It generates a C++ class for each module (mode/agent) of the model that can be compiled separately. Using the code generator, we can translate the models explained in Section 2 into C++ programs and compile them into executable code for the robot (see Figure 9). Since the generated code is virtually platform-independent, we are required to write additional code that interfaces the operating system, which should be done manually in part. We also implemented a run-time scheduler independently of the code generator that can be compiled and linked with automatically generated code. The scheduler is invoked periodically

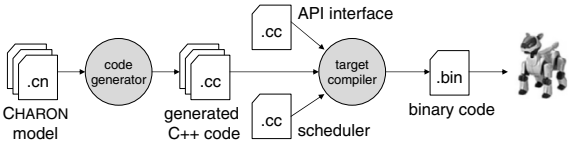


**Figure 9. Code generation process.**

**Table 1. Generated code evaluation.**

| | Execution time | Code size |
|---|---|---|
| PANNING THE HEAD (Figure 1) | | |
| Static scheduling | 5,910 msec | 44,992 bytes |
| Dynamic scheduling | 8,370 msec | 47,449 bytes |
| TRACKING AN OBJECT (Figure 2) | | |
| Static scheduling | 23,740 msec | 59,789 bytes |
| Dynamic scheduling | 32,690 msec | 60,173 bytes |
| WAGGING THE TAIL (Figure 3) | | |
| Static scheduling | 31,180 msec | 60,960 bytes |
| Dynamic scheduling | 40,500 msec | 62,413 bytes |

by the operating system of the robot. All the generated code were compiled and run smoothly as intended by the model.

We measured performance of the code to evaluate the cost of dynamic scheduling. The measurement is done in the host system, since measuring tools for the target platform were not available. Table 1 shows the result obtained by compiling and running the code generated from the models explained in Section 2 in the host system based on the 1.8 GHz Intel Pentium 4 processor running the Linux operating system. The execution time indicates the CPU time consumed to execute the application up to 100,000 steps. The results show that run-time overheads of dynamic scheduling range from 30% to 40% compared to the statically scheduled code in our implementation. This indicates that static scheduling improves the performance significantly, and thus should be preferred wherever possible. However, static scheduling requires that data dependency should be resolved at the stage of code generation, and is limited to models that do not have potential cyclic dependency. The results also show the overhead of the code size due to dynamic scheduling, but overhead is much less significant.

## 6 Conclusion

We have presented a framework of automatic code generation for embedded real-time systems from models specified in hybrid automata. The automatic code generation process is decomposed into two phases: one translating each primitive into a piece of code and the other scheduling the pieces of code consistent to data depen-

dency. We have shown that data dependency analysis can be done separately in two domains, and that code can be modularized if the dependency in each domain is acyclic. The framework generates modular and efficient code suitable for single-threaded execution even when the model has arbitrarily complex hierarchy and concurrency.

The framework is implemented in the hybrid system modeling language CHARON and tested in a robot platform AIBO. We feel that the model-based approach is promising especially in hybrid systems. Robot programming, for example, generally requires implementation of finite state machines and periodic update of variables, and in many cases these are hand-crafted using traditional programming languages. Debugging is more difficult because reasoning is done at the level of code, rather than at the level of the abstract model. In contrast, automatic code generation improves productivity since it eliminates errors due to tedious manual coding and allows the designer to be devoted to higher level design issues.

Our code generation framework is based on a formal language, and we have defined the relation between the model and the generated code formally. There, however, still exists discrepancy between them, when a numerical solution of differential equations is not precise, and when discretizations are executed in a distribute system where communication delays are present. We are currently addressing these issues by considering correctness of the generated code in more general cases. This paper has focused on semantic relationship between the model and the generated code, and we have largely ignored a possible performance gap between the two. We are also studying on improving performance of the generated code by exploiting the hierarchical structure of the model.

**Acknowledgements.** The authors would like to thank Rajeev Alur, Franjo Ivančić, and Oleg Sokolsky for their various contributions to the code generation framework for CHARON. We are also grateful to Jin-Young Choi and Yerang Hur for their invaluable comments.

## 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. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 2003.

[3] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. Technical Report MS-CIS-03-07, Dept. of Computer and Information Science, University of Pennsylvania, 2003.

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

[5] J.-Y. Choi, Y. Hur, and I. Lee. IHA: Ensuring sound numerical simulation of hybrid automata. Technical Report MS-CIS-03-06, Dept. of Computer and Information Science, University of Pennsylvania, 2003.

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

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Sc. of Comp. Programming*, 8:231–274, 1987.

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

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

[10] 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.

[11] 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.