7-13-2003

Testing and Monitoring Model-based Generated Program

Li Tan
University of Pennsylvania

Jesung Kim
University of Pennsylvania

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

Postprint version. Published in *Electronic Notes in Theoretical Computer Science*, Volume 89, Issue 2, October 2003.
Publisher URL: http://www.sciencedirect.com/science/journal/15710661

Testing and Monitoring Model-based Generated Program *

Li Tan, Jesung Kim, and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA

1 Introduction

Automatic code generation from hybrid automaton models attracts much research interest recently [14, 15, 5]. Besides the significant cut of the development cost, the benefits of automatic code generation also include that the system can be verified on the model level and the generated code can be free of human errors which otherwise may be introduced by manual translation. Recently, code generation from hybrid system models has been used for embedded systems. Industry has followed the trend by providing their own tools (cf. Real-time workshop for Simulink [14]). Nevertheless, the validation of the generated code still much relies on traditional techniques designed for code created by conventional means. We intend to fill this missing link by showing how the model-based code generation paradigm may also benefit the validation of generated code.

The validation techniques explored in this paper include testing and runtime verification. Testing is a well-studied domain. Usually testing checks the behavior of the tested system in response to a test suite. In contrast to static verification techniques like model checking [9], testing in general cannot provide full assurance of the tested system. Nevertheless, it is well studied and widely accepted in the field, and testing can work on the implementation level, in our case, on a hardware/software integrated embedded system. A benefit of model-based design is that a test suite may be generated from the model [13]. For this purpose, we implemented a simulation-based test-suite generator for hybrid automata. Our simulation-based test-suite generator produces a test suite in the form of a set of traces which satisfies the desired coverage criteria. The generated test traces are then loaded into the targeted embedded system to perform a test. However, a problem with such an approach is that traces are often prohibitively big to be fit into the targeted embedded system. In this paper, we advocate an alternative approach: instead of directly generating and applying a test suite, we create a testing automaton which can produce the desired test trace during its execution. Since testing automata themselves are also hybrid automata, the same code generation mechanism may be exploited to generate a tester from a testing automaton. The tester can then be linked with the rest of the generated code to supply the test trace when the code is running on the targeted embedded system. Once testing is done, the tester can be eliminated by simply recompiling the code without the tester.

To check the behavior of generated code during testing, we apply the idea of runtime verification (monitoring). Runtime verification is a technique to check in real time whether an execution of the program violates the given properties. In our approach, system properties are encoded as a deterministic Büchi timed automaton, from which a monitoring automaton may be synthesized. The monitoring automaton may be encoded as a hybrid automaton, and hence the same code generation mechanism can again be used to generate a monitor from the monitoring automaton. Figure 1 shows the overall flow of our toolset.

The code generation process we are using supports modular compilation [5]. Each component of the resulting hybrid model – the system model, the testing automaton, and the monitoring automaton – may

*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.

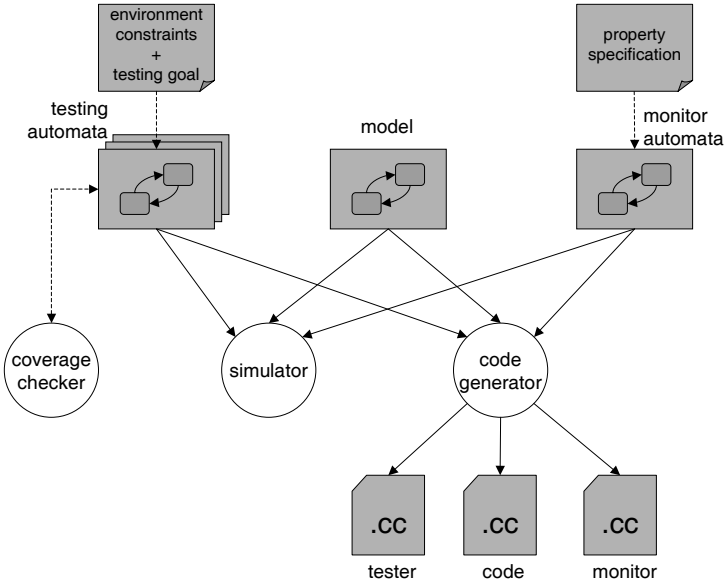

Figure 1: Framework for testing and monitoring model-based generated code

be generated and compiled separately, and linked only when testing and/or monitoring task is requested. We will illustrate our approach by going through an example built on a SONY AIBO robotic dog [7]. The original model is specified in Charon [2], a modeling language for hybrid automata which supports both behavioral and structural hierarchy as well as resource hiding. These features bring extra benefits to our approach: the hierarchical structure allows us to put the tester and the monitor to the exact level of hierarchy which we want to validate, and resource hiding allows us to specify what may or may not be seen by the tester and the monitor.

The rest of paper is organized as following. Section 2 prepares the notions and definitions. It also includes a brief introduction of Charon and the code generation process [15] which we will use. Section 3 discusses the issue of generating a test suite and integrating it as a part of the model being tested. In Section 4 we synthesize a monitor from temporal specification. Section 5 discusses our experiment on a SONY AIBO robot. We show that our approach may be used both on the model level by the simulation technique, or on implementation by plugging into the targeted hardware platform the generated code as well as the tester and the monitor. Finally, we conclude with a discussion on future directions.

Related works. Our work on synthesizing monitors is close in spirit to previous works on runtime verification based on formal methods, for instance, MaC [16] and Java PathExplorer [11]. Both tools work on the code level. They are capable of instrumenting Java bytecodes, observing the events emitted by the program, and comparing them with formal specification. However, our approach is capable of working on both the model level and the code level, that is, we may also combine the monitor model with the system model and run the compositional model on a simulator. In [12] authors show how to synthesize a monitor program directly from formal specification, while we take a different route: we synthesize monitors as hybrid automata, and leave the generation of actual monitor programs to code generators. In [8] and [10] authors show how to synthesize automaton-based monitors (test oracle) from temporal logics for systems with discrete events. However, in our approach, we are more interested in handling continuous dynamics in hybrid systems.

2 Preliminaries

2.1 Hybrid Automata

In this paper we consider hybrid systems, that is, systems with both discrete and continuous behaviors. Formally, hybrid systems are modeled by hybrid automata [1, 18]. We use the following definition of hybrid automata for our discussion.

Definition 2.1 *A hybrid automaton $\mathcal{A}$ is a tuple $\{S, V, T, G, W, D, A, I, C, I_0\}$, where*

- *S is a set of locations.*
- *V is a set of variables defined on $\mathbb{R}$.*
- *$T \subseteq S \times S$ is a set of transitions between two locations.*
- *G assigns to each $t \in T$ a guard, denoted as $G(t)$, which is a predicate over V. A transition $t \in T$ is said enabled when $G(t)$ is true.*
- *W assigns to each $t \in T$ an assignment, denoted as $W(t)$, which is a partial function from $V(t) \subseteq V$ to $\mathbb{R}$. G and W collectively define discrete behavior of $\mathcal{A}$. An assignment changes the value of variables in $V(t)$ to $W(t)$ instantaneously when t 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.*
- *A is a set of algebraic equations in the form of $x = g(X)$, where $x \in V$, X is a vector of variables in $V \setminus \{x\}$.*
- *I is a set of predicates over V 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.*
- *I_0 is the initial state, which is a tuple $\{s_0, V_0\}$, where $s_0 \in S$ is the initial location and $V_0 : V \to \mathbb{R}$ is the initial valuation of the variables in V.*

A run of a hybrid automaton can be defined similarly to the traditional finite state machine, except that variables are changing continuously according to differential equations and algebraic equations corresponding to the current location. Also note that hybrid automata may have more than one possible run, since the transition may occur any time when the guard is true, provided that the invariant is true.

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 to specify switching control laws. These are also 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 2 shows a simple hybrid automaton modeling a robot dog panning its head. It consists of two locations, each of which specifies constant increase/decrease ($\pm 10\,°/s$) 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 ($\pm 45\,°$). 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 non-deterministic). The invariant of each location specifies that the switch should occur before the head moves beyond its allowed range ($x \leq 46$ and $x \geq -46$). Once the automaton is compiled into a programming language and the variable x 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.

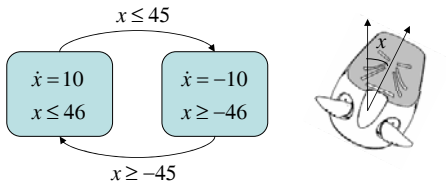

Figure 2: Hybrid automaton modeling a robot dog panning the head.

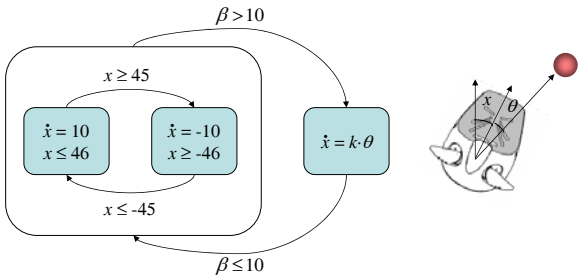

Figure 3: Hierarchical hybrid automaton modeling a robot dog tracking an object.

Hybrid automata can be composed hierarchically and/or concurrently to model more complex systems. In a hierarchical hybrid automaton, a location can be a hybrid automaton, or another hierarchical hybrid automaton. Figure 3 shows a hierarchical hybrid automaton modeling a robot dog tracking an object. The variable θ indicates the difference between the direction of the head and the object, and the variable β is the degree of visibility of the ball. When β is greater than a certain threshold, the robot attempts to move the head towards the object, as modeled by a differential equation $\dot{x} = k \times \theta$. However, if β is below the threshold, the robot gives up tracking the object, and continues panning the head. Note that the same model in Figure 2 is reused to model the movement of the head.

Figure 4 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 3 with another automaton for wagging the tail, which is very similar to the model shown in Figure 2 and the details are omitted. The automaton shown in Figure 3 is slightly modified such that it assigns to the variable v a value greater than zero when the dog detects the object. This triggers the tail to wag.

A hybrid automaton can be identified by a set of possible traces of states. The state of a hybrid automaton is expressed as a tuple $\{s_i, V_i\}$, where $s_i \in S$ is the current active location and V_i is a valuation of the variables in V.

Definition 2.2 *A trace of a hybrid automaton is a sequence of tuples* $\rho = \langle s_0, \mathcal{V}_0, \mathcal{I}_0 \rangle \langle s_1, \mathcal{V}_1, \mathcal{I}_1 \rangle \cdots$ *where* $\mathcal{I}_i = [\mathcal{I}_i^l, \mathcal{I}_i^h)$ *represents a time interval with* $\mathcal{I}_0 \mathcal{I}_1 \cdots$ *partitions* $[0, \infty)$ *and* $\mathcal{V}_i : \mathcal{I}_i \to (V \to \mathbb{R})$. *In addition,*

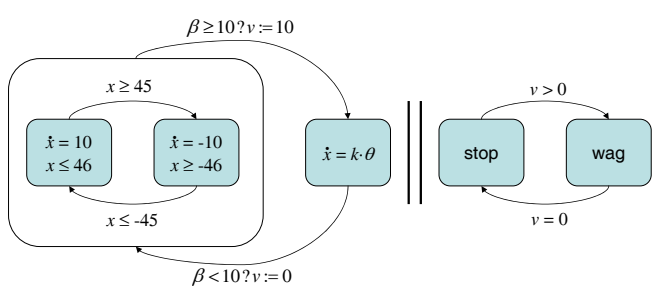

Figure 4: Concurrent hierarchical hybrid automaton modeling a robot dog wagging the tail.

it satisfies the following,

1. $\mathcal{V}_0(0) = V_0$.
2. *For each pair of* $\langle s_i, \mathcal{V}_i, \mathcal{I}_i\rangle\langle s_{i+1}, \mathcal{V}_{i+1}, \mathcal{I}_{i+1}\rangle$, $\langle s_i, s_{i+1}\rangle \in T$, $G(\langle s_i, s_{i+1}\rangle)(\mathcal{V}_i(I_i^h)) = true$, *and* $\mathcal{V}_{i+1}(\mathcal{I}_{i+1}^l) = W(\langle s_i, s_{i+1}\rangle)$.
3. $\mathcal{V}_i$ *satisfies all the constraints enforced by* $C(s_i)$.

Composing two hybrid automata to form a new one is much like the case of composing two discrete automata. The composition of the automata $\mathcal{B} = \{S_b, V_b, T_b, G_b, W_b, D_b, A_b, I_b, C_b, I_{b0}\}$ and $\mathcal{C} = \{S_c, V_c, T_c, G_c, W_c, D_c, A_c, I_c, C_c, I_{c0}\}$ yields a hybrid automaton $\mathcal{B}||\mathcal{C} = \{S, V, T, G, W, D, A, I, C, I_0\}$, where,

1. The locations $S = S_b \times S_c$ is the product of $\mathcal{B}$ and $\mathcal{C}$'s locations
2. $t = \langle\langle s_b, s_c\rangle, \langle s_b', s_c'\rangle\rangle \in T$ if $s_b = s_b' \wedge \langle s_c, s_c'\rangle \in T_c$, in which case $G(t) = G_c(\langle s_c, s_c'\rangle)$ and $W(t) = W_c(\{s_c, s_c'\})$, or $s_c = s_c' \wedge \langle s_b, s_b'\rangle \in T_b$, in which case $G(t) = G_b(\langle s_b, s_b'\rangle)$ and $W(t) = W_b(\langle s_b, s_b'\rangle)$.
3. $V = V_b \cup V_c$, $D = D_b \cup D_c$, $A = A_b \cup A_c$, and $I = I_b \cup I_c$.
4. $C(\langle s, s'\rangle) = C_b(s) \cup C_c(s')$, for all $\langle s, s'\rangle$ reachable from $\langle s_0, s_0'\rangle$.

Let ρ be a trace of $\mathcal{B}||\mathcal{C}$ and $\rho\lceil\mathcal{B}$ be the project of ρ on $\mathcal{B}$, then clearly $\rho\lceil\mathcal{B}$ is also a trace of $\mathcal{B}$. However, the reverse doesn't hold. Composing traces of component automata may not necessarily yield a trace for the compositional automaton, since the composed trace may not satisfy the constraints imposed by both $\mathcal{B}$ and $\mathcal{C}$.

2.2 Modeling Language Charon

Charon [2] is a modeling language for hierarchial hybrid automata. We provide a very brief introduction to Charon with the emphasis on the features related to the techniques addressed in this paper. For the description of full features of Charon, please refer to [2].

The behavioral hierarchy of Charon is encoded as the hierarchy of modes. Each mode describes a continuous behavior and a single thread of discrete control. Each mode has its own constraints in terms of differential equations, algebraic equations, and invariants, just as a location in an ordinary hybrid automaton. Nevertheless, a mode in a Charon model may also contain a set of submodes. At any given moment, at most one submode is active in an active mode. Transitions link a mode with its sibling modes, parent mode, and child modes. The constraints imposed by an active mode is a collection of constraints of the modes and all its active descendant modes.

The architectural hierarchy of Charon is implemented by agents. Each agent stands for a hybrid automaton. An agent may be compositional, in which case it contains several subagents, or atomic. Atomic agents are building blocks of architectural hierarchy, and a compositional agent functions as the composition of all its descendant atomic agents.

Charon also provides the capability of resource hiding, which is implemented by defining the scope of variables. At any level of hierarchy, a mode or an agent may specify the attributes of variables it can assess as "read", meaning that a variable defined in a higher level can be read in the current agent or mode, "write", meaning that a variable defined in a higher level may be read and written to, and "private", meaning that a new variable is introduced. Resource hiding will help define the interface between the tester/monitor and the system by specifying what variables may or may not be seen by the tester/monitor. The skeleton of Charon model for tester, monitor, and controller which we will use in our case study is given in Figure 5.

```
agent DogHead() {
    private analog real head_pan;
    private analog real ball_pan;
    private analog real vision;
    ......
    agent headAgent = head();
    agent testerAgent = ball();
    agent monitorAgent = monitor();
    ......
}

agent head() {
    write analog real head_pan;
    read analog real ball_pan;
    read analog real vision;
    ......
    mode topMode = headMode();
    ......
}

agent ball() {
    write analog real ball_pan;
    write analog real vision;
    mode topMode = ballMode();
    ......
}

agent monitor {
    read analog real ball_pan;
    read analog real head_pan;
    read analog real vision;
    .....
    mode topMode = monitorMode();
}
```

Figure 5: Skeleton of CHARON model used in SONY AIBO example

```
class M : public mode {
    var v1, v2, ...;      // variables
    mode *m1, *m2, ...;   // submodes

    void diff() { ... }   // differential
    void alge() { ... }   // algebraic
    bool inv() { ... }    // invariant
    mode *trans() { ... } // transition
};
```

Figure 6: C++ class skeleton for a mode.

2.3 Generating Code from Charon

The code generator transforms a 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 periodically at every Δ time unit. We chose C++ as an intermediate target language, mainly because the object-oriented features of the language best suit Charon and make the code/test generation process simpler, and also because the language has been deployed in many real systems, including SONY AIBO Robotic dog.

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 submodes, as shown in Figure 6. Readers are referred to [5] for a full description of the code generation algorithm.

Variable. Variables in Charon are either private or shared. Each private variable is translated into a variable class instance, while each shared variable 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 private variable. Variables are represented by instances of class `var` that has methods `read()` and `write()`, which are used to get the value and assign a new value to the variable. Top-level variables need to be handled differently, since they can be mapped either to platform specific APIs or to the tester. This mapping is done by overriding `read()` and `write()` methods in a derived class of `var`, that are used to get the value from and put the value to the environment, respectively. This allows us to associate variables to the tester without modifying the automatically generated code.

Differential equation. A differential equation of the form $\dot{x} = f(X)$ declares that a variable x should evolve continuously at a rate given by the expression $f(X)$ over variables which may be continuous. 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. This approximation, known as *Euler's method*, is efficient to compute and produces good results when dynamics is not changing rapidly. More advanced, but more expensive methods can be used to improve accuracy [15]. Interested readers are referred to [6], in which the effect of numerical errors for sound simulation of hybrid automata is addressed formally.

Algebraic equation. An algebraic equation declares that an equation involving variables should be satisfied at all times. In Charon, an algebraic equation is specified as the form of $y = f(X)$, where $f(X)$ is an expression of variables other than y. Such an equation can be translated into an assignment statement.

Invariant. An invariant 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 run-time checking of properties that are not part of the model through a monitor as explained in Section 4.

Transition. Transitions specify the control flow of the model. In Charon, transitions can be non-deterministic. Non-determinism comes from two sources. First, a transition can be enabled for a duration of time in which the transition can be taken at any time instance. Second, more than one transition can be enabled at the same time. Non-determinism in the Charon model implies more than one valid

implementations. In our implementation, we use an *urgent branching* policy, meaning that a transition is taken as soon as it is detected to be enabled. Such a policy can be implemented simply by translating it into an if-then statement where the if-block contains the guard and the then-block contains the optional discrete actions. More sophisticated approach can also be considered (e.g., [6]), but it is beyond the scope of this paper. Furthermore, the second source of non-determinism is resolved by ordering the if-then statements according to priority. The priority of a transition is implicitly encoded as the order of appearence in the textual form of the CHARON model.

Mode. The class for modes has two methods, `continuousStep()` and `discreteStep()` that perform evaluation of differential/algebraic equations and transitions, respectively. Each method is invoked by the same method of the parent mode. These methods are implemented in a base class `mode` and shared by all the modes. The class also contains run-time information such as the pointer to the currently active submode. This pointer constitutes a linked list of active submodes from the top-level mode to some leaf mode. The methods of the top-level mode are invoked by the same 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 subagents 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 all the subagents. Note that, since the code for each agent is modularized, code for testing and monitoring agents can be easily coupled or decoupled, without modifying the code for the model.

3 Generating Tester

Typically testing involves checking how a tested system responses to a test suite, that is, a finite set of finite traces selected according to the required coverage criteria. Some sample coverage criteria for discrete systems include state coverage, transition coverage, and modified condition coverage etc. In our case study, we are interested in *mode coverage*. Mode coverage requires that test suite traverse all the modes in a CHARON model.

We implemented a simulation-based test-suite generator for hybrid automata written in CHARON. The heart of our test-suite generator is a simulation routine [2]. To generate a test suite for a hybrid automaton $\mathcal{A} = \{S, V, T, G, W, D, A, I, C, I_0\}$ with $I_0 = \{V_0, s_0\}$, test suite generator simulates the behavior of $\mathcal{A}$, starting with the initial valuation V_0 and the initial location s_0. Let $\langle s_0, \mathcal{V}_0, [0, u_0)\rangle \cdots \langle s_i, \mathcal{V}_i, [u_{i-1}, u)\rangle$ be the trace explored so far, the test suite generator may have more than one of the following choices,

- Continuous update. If $\mathcal{V}_i(u)$ satisfies invariants imposed by $C(s)$, then the test-suite generator can stay for another integration step δ at location s_i and extends $\mathcal{V}_i$ to $\mathcal{V}'_i$ such that $\mathcal{V}_i(m) = \mathcal{V}'_i(m)$ for $m \in [u_{i-1}, u]$, and $\mathcal{V}'_i(m)$ satisfies all the constraints imposed by $C(s)$ for $m \in (u, u+\delta]$. [1]
- Discrete update . If there is a transition $t = \{s_i, s_{i+1}\}$ such that $\mathcal{V}_i(u)$ makes true the guard $G(t)$, the test-suite generator may choose to take t.

Unless an enable transition is urgent, i.e., any prolonged stay at the current location will mean the violation of invariants, the test-suite generator may choose between continuous update and discrete update, and in the case of continuous update the size of integration step, or in the case of discrete update which enabled transitions to be taken. The deviation on these choices means a different simulation trace. The test-suite generator logs the transitions and modes the trace has covered to check whether the traces reach the required coverage criteria.

[1] Integration routine in simulator may introduce numerical error, which may prevent detecting the potential violation of invariants during an integration step. Nevertheless, this is not in the scope of this paper. Interested readers are referred to [6].

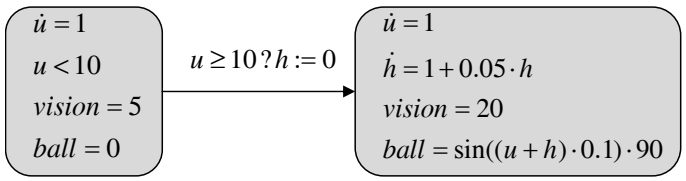

Figure 7: Environment (ball) model

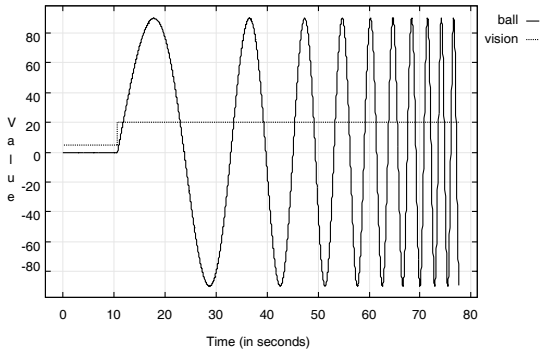

Figure 8: Movement of the ball

Embedded system interacts with the environment with a defined I/O interface. In a hybrid model for embedded system I/O interface is defined by a set of inputs variables V_I and output variables V_O such that $V_I \subseteq V$, $V_O \subseteq V$, and $V_I \cap V_O = \emptyset$. For the formal description of hybrid I/O automata, users may refer to [17], but the rough idea is that the hybrid model has no control over input variables V_I, that is, no input variable $v \in V_I$ appears in the left side of differential or algebra equations. If we directly put such automata on our test-suite generator, the test-suite generator will choose the random value for these input variables since no constraints are imposed on them, which reflects a completely chaotic environment. However, in many applications one may want to test the system under more controlled environment. Some common reasons for testing under controlled environment include (1) the embedded system is designed to function correctly under certain assumption on environment, or (2) we are more interested in how the system reacts to certain environment. In the case of testing SONY AIBO Robot, we are more interested in seeing the reaction of the dog to the accelerated movement of the ball.

To test the system under a controlled environment, we model the environment itself as a hybrid automaton. The output variables of the environment model are the input variables of the system model. The hybrid automaton for the controlled environment (i.e. ball) is given in Figure 7. The model basically describes the movement of the ball. Figure 8 shows the simulated trace of the ball's movement. The movement of the ball has two phases: Initially, it remains invisible; after exactly 10 seconds, it starts to be visible and waved in front of the dog. This controlled environment is designed to test all the modes in SONY AIBO model. The initial invisible phase tests the dog's behavior when the ball is invisible, and the second phase tests how well the dog tracks the ball. The introduction of non-linear factor h is to accelerate the ball's movement.

After each run of simulation, the test-suite generator checks whether the current trace could contribute to the requested coverage criteria, and if so, the test-suite generator keeps the current traces and continue test-suite generation if the requested coverage criteria has not been met. A final test suite may be obtained by projecting the traces to the input variables. In our example, a single test trace in Figure 8 already achieves the desired mode coverage. In general, however, the mode coverage on the mode level may not imply the mode coverage on the code level because of the semantics difference between the model and the generated code. This is a topic of future research.

Last but not least, we need to test the generated code with the obtained test suite. A traditional way to do so is to write an input/output routine to read the trace in and feed it into the generated code. Given the fact that the most of embedded systems have very limited internal memory, loading the test suite to the targeted system is not always possible. In our example, a single test trace for an 80-second test contains 8000 lines of information, or roughly 240 KB in the size, and this is just with a moderate setting on the integration step size. With a longer test and a finer integration process, the test suite can easily exceed the capacity of memory in the targeted embedded system.

We choose a different approach: since test traces are essentially an execution of the environment model, we may use a tester, a program which reassembles the function of the environment model, and link the tester with the code generated from the system model. Moreover, since we already have the ability of automatic code generation, we can directly generate the tester from the environment model. This is exactly what we did on our SONY AIBO example. We generate the tester from the environment model shown in Figure 7, link it with the code generated from the system model and the monitor (which will be introduced shortly), and then load it to the robot. The generated tester has 472 lines of `C++` code and slightly increases the binary code by 18 KB. Note that the tester remains the same size regardless of the duration of tester or the size of integration step.

As summary, the approach we advocate for generating and applying test suite is following: first we analyze the test requirement and the constraints of hardware/software environment to create a model for controlled environment. Next, we bounded the environment model with the system model and put them on a test-suite generator. Third, we refine the environment model to produce the exact test trace we want to test on the targeted system. Finally, the same code generator used to generate the code from the system model may be used to generate a tester from the refined environment model and link it with the rest of generated code.

4 Synthesizing Monitor

4.1 Encoding Properties

In our approach, the system property is encoded as passive timed automata.

Definition 4.1 *A passive timed automaton is a tuple* $\{S, s_0, V, X, T, G, W\}$*, where*

1. S *is a set of locations.*
2. $s_0 \in S$ *is the initial location.*
3. V *is a set of variables.*
4. X *is the set of clocks.*
5. $T \subseteq S \times S$ *is the transition relation.*
6. G *assigns each transition a guard, which is a predicate on* $X \cup V$*.*
7. $W : T \to 2^X$ *associates a transition with a set of clocks which need to be reset of taking the transition.*

A passive timed automaton is deterministic if for each pair of transition $t_1 = \{s, s_1\} \in T$ *and* $t_2 = \{s, s_2\} \in T$*,* $G(t_1) \wedge G(t_2)$ *is unsatisfiable.*

Definition 4.2 *A trace of a passive automaton* $\mathcal{B} = \{S, s_0, V, X, T, G, W\}$ *is a sequence* $\rho = \langle s_0, X_0, \mathcal{V}_0, \mathcal{I}_0\rangle \langle s_1, X_1, \mathcal{V}_1, \mathcal{I}_1\rangle \cdots$ *where* X_i *are the evaluation of clocks upon entering location* s_i*,* $\mathcal{V}_i : \mathcal{I}_i \to (V \to \mathbb{R})$ *records the change of values of variables at location* s_i*, the time interval* $\mathcal{I}_i$ *represents time spent at location* s_i*. In addition,* ρ *satisfies the following constraint: for each pair of* $\langle s_i, X_{i+1}, \mathcal{V}_i, \mathcal{I}_i\rangle\langle s_{i+1}, X_{i+1}, \mathcal{V}_{i+1}, \mathcal{I}_{i+1}\rangle$*,* $t = \langle s_i, s_{i+1}\rangle \in T$*,* $G(t)(\mathcal{V}_i(\mathcal{I}_i^h)) = true$*, and,* $X_{i+1}(x) = X_i(x) + \mathcal{I}_i^h - \mathcal{I}_i^l$ *for* $x \notin W(t)$ *and* $X_{i+1}(x) = 0$ *for* $x \in W(t)$*.*

A trace is infinite if $|\rho|$ *is infinite, or* $|\rho|$ *is finite but* ρ *stays in the last location for infinite time.*

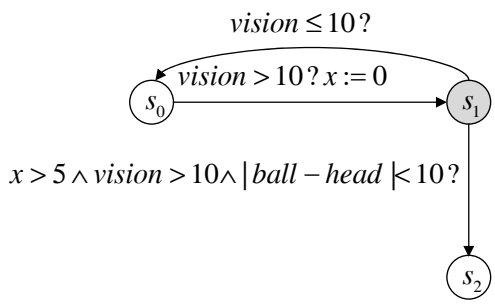

Figure 9: A Deterministic Passive Büchi Timed Automaton, where $\{s_1\}$ is the Büchi acceptance condition

Passive timed automata accept *timed words* on the valuation of variables. A timed word is a sequence $\eta = \{\mathcal{V}_0, \mathcal{I}_0\}\{\mathcal{V}_1, \mathcal{I}_i\}\cdots$, where $\mathcal{I}_i$ is a time interval such that $\mathcal{I}_i^h = \mathcal{I}_{i+1}^l$, and $\mathcal{V}_i(m)$ such that $m \in \mathcal{I}_i$ is the evaluation on the set of variables.

Definition 4.3 *A timed word $\eta = \{\mathcal{V}_0, \mathcal{I}_0\}\{\mathcal{V}_1, \mathcal{I}_1\}\cdots$ is accepted by a passive timed automaton $\mathcal{B} = \{S, s_0, V, X, T, G, W\}$ if there is a trace of $\mathcal{B}$: $\rho = \langle s_0, X_0, \mathcal{V}_0', \delta_0'\rangle\langle s_1, X_1, \mathcal{V}_1', \delta_1'\rangle\cdots$ such that, if $m \in \mathcal{I}_i$, then there is a l such that $m \in \mathcal{I}_l'$ and $\mathcal{V}_k(m) = \mathcal{V}_l'(m)$.*

A passive Büchi timed automaton is a passive timed automaton extended with a subset of locations as Büchi acceptance condition.

Definition 4.4 *A passive Büchi timed automaton is a tuple $\mathcal{B} = \{S, s_0, V, X, I, T, G, W, F\}$, where $\{S, s_0, V, X, I, T, G, W\}$ is a passive timed automaton and $F \subseteq S$ is the Büchi acceptance condition. A trace ρ is an accepting run of a passive Büchi timed automaton if and only if ρ is infinite and $inf(\rho) \cap F \neq \emptyset$, where $inf(\rho)$ is the set of locations ρ visits infinitely often, or the last location of ρ if $|\rho|$ is finite. $\mathcal{B}$ accepts a timed word η if $\mathcal{B}$ has an accepting run for η.*

Figure 9 shows a deterministic passive Büchi timed automaton (DP-BTA). It accepts a timed word η if $|ball - head| < 10$ holds within five second after whenever $vision > 10$ holds, unless $vision \leq 10$. Note that in our example of robotic dog, $vision > 10$ means that the ball is visible, and $|ball - head| \leq 10$ indicates that the angle between ball and head is close enough. The automaton in Figure 9 encodes the property that the dog shall closely chase the ball five second after the ball is visible.

Remarks. The definition of passive timed automata is quite similar to that of traditional timed automata [3]. However, the major difference between the two is that passive timed automata take predicates as the input from environment, while traditional timed automata take actions or events. In our application of monitoring hybrid automata, working on predicates directly allows us to save the extra work of interpreting events or actions as the changes on predicates, which otherwise may be required if we use the traditional notion of time automata.

4.2 Synthesizing Monitor from Properties

Now our question is, given a property specified as a DP-BTA, how we can synthesize a monitor to detect an erroneous execution. There are two problems: First, since the monitor sets to detect the erroneous behavior in an execution but an execution is always finite in practice, we need to translate DP-BTA to an automaton which accepts finite traces; Second, since our code generator works on hybrid automata, or more specifically, Charon models, the monitor should be expressed as a hybrid automaton.

Definition 4.5 *Given a DP-BTA $\mathcal{B} = \{S, s_0, V, X, T, G, W, F\}$, a monitoring automaton is a tuple $\mathcal{M} = \{S, s_0, V, X, T, G, W, F'\}$, where $s \in F'$ if there is no path in T from s to any $s' \in F$. A timed word $\eta = \langle\mathcal{V}_0, \mathcal{I}_0\rangle\langle\mathcal{V}_1, \mathcal{I}_1\rangle\cdots$ is accepted by $\mathcal{M}$ if and only if $\mathcal{M}$ has a trace ρ for η such that there is $s_i \in F'$ on ρ.*

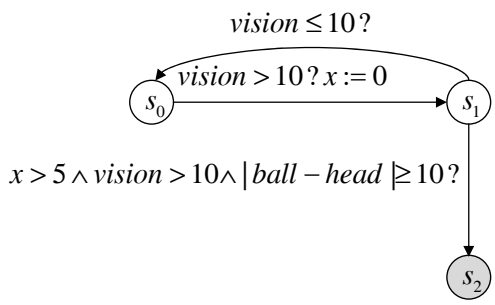

Figure 10: Monitor automaton for DP-BTA in Figure 9

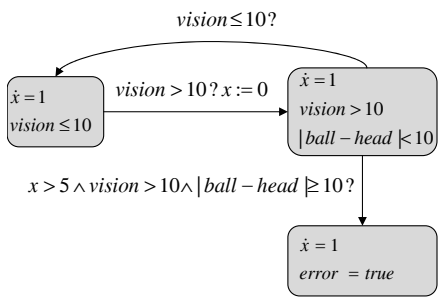

Figure 11: Hybrid automaton version of the monitoring automaton

Theorem 1 *Let $\mathcal{B}$ be a deterministic passive Büchi timed automaton and $\mathcal{M}$ be the monitoring automaton for $\mathcal{B}$, then timed words accepted by $\mathcal{M}$ cannot be accepted by $\mathcal{B}$.*

Figure 10 gives the monitoring automaton of DP-BTA in Figure 9. Both automata have similar structure, except that the monitoring automaton has $\{s_2\}$ as its acceptance set.

Note that a monitoring automaton also accepts finite words. By Theorem 1, the acceptance of a finite prefix of a timed word ρ by the monitor automaton implies that ρ will be rejected by the original DP-BTA.

Next, we need to encode monitoring automata as hybrid automata. There are two problems we need to solve: First, we have to find a way to handle clocks since hybrid automata don't explicitly have clock variables; Second, transitions in a timed automaton are *urgent*, i.e., whenever the guards of some transitions are satisfied, the automaton takes some enabled transition, while in the case of a hybrid automaton continuous update may still be taken even if the guard of some transitions are satisfied, given that the invariant of the current location is not violated.

The solution to the first problem is simple. For each clock variable we introduce a variable x in the hybrid automaton, and define the dynamics of x as $\dot{x} = 1$. For the second problem, we add to each location an invariant which is the negation of conjunction of the guards of all its outgoing transitions. Formally we define a translation procedure Π as below: Given a monitoring automaton $\mathcal{M} = \{S, s_0, V, X, T, G, W, F'\}$, $\Pi(\mathcal{M})$ is a hybrid automaton $H = \{S, V', T, G, W', D, A, I, C, I_0\}$ such that,

- $V = V \cup X \cup \{error\}$.
- $W'(t)(x) = 0$ for every $t \in T$ and $x \in W(t)$.
- $D = \{\dot{x} = 1 \mid x \in X\}$, $A = \{error = true\}$, and $I = \bigcup_{s \in S}(\bigvee_{\langle s,s' \rangle \in T} \neg G(\langle s, s' \rangle))$
- $C(s) = D \cup \{\bigvee_{\langle s,s' \rangle \in T} \neg G(\langle s, s' \rangle)\} \cup \{error = true \mid s \in F'\}$ for $s \in S$.

Figure 11 gives the hybrid automaton version of the monitoring automaton in Figure 10.

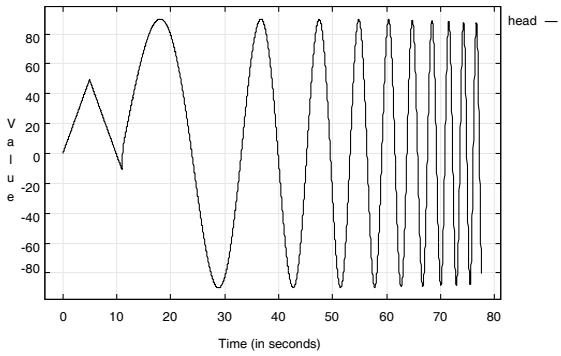

Figure 12: Movement of the head

Theorem 2 *Given a DP-BTA $\mathcal{B} = \{S, s_0, V, X, T, G, C, W, F\}$ with $\mathcal{M}$ as its monitoring automaton and a hybrid automaton $\mathcal{H}$, if ρ is a trace of $\mathcal{H}||\Pi(\mathcal{M})$ and error is true on ρ, then $\rho\lceil V$ cannot be accepted by $\mathcal{H}$. $\rho\lceil V$ is a timed word obtained by projecting ρ on the set of variables V.*

5 Case Study

To assess the feasibility of our approach, we test it on the SONY AIBO Robot dog. The robot consists of both 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 latter may be used for loading controlling program. The operating system is Sony's proprietary object-oriented real-time operating system known as Aperios.

In [15] authors develop a code generator which can generate controlling programs for embedded systems from Charon specification. The code has been generated to control head (tracking a ball) and leg (walking) movement. In our case study, we want to test and monitor the code generated for head movement. The dog has a two-dimension light sensor and two step motors, each of which controls the head's vertical or horizontal movement. The two-dimension light sensor can measure the relative angle between head and a bright object, in our case, a red ball. In [15] the automatic code generation process produces a controlling program to track the ball. The code takes the input from sensor and sends signal to motors to move the head towards the ball. The code controls both vertical and horizontal movement. However, for simplicity we only describe the control of vertical movement. The control model is a hierarchical hybrid automaton $\mathcal{H}$ given in Figure 3. It takes two inputs from the sensor: the relative angle (θ in Figure 3) between the ball and the head, and *vision* (β in Figure 3), the visibility of the ball. The goal of our task is to,

1. Test the functionalities of generated code specified by each mode in a hybrid automaton (i.e. mode coverage).

2. Test how well dog track the ball.

The hybrid automaton $\mathcal{T}$ for the tester is given in Figure 7.

The property of interest is that dog can closely track the ball whenever the ball is visible. The property is formally captured by DP-BTA in Figure 9. The hybrid automaton $\mathcal{M}$ for monitoring is given in Figure 11.

We did testing and monitoring on simulation level and hardware level. On the simulation level, we run $\mathcal{M}||\mathcal{H}||\mathcal{T}$ on Charon simulator. Figure 12 gives the head movement. In the first 10 second, the head swings to the right then back, indicating that the ball is invisible, then it starts to track the ball.

Figure 13 puts together the traces of the angle between the ball and the head, variable vision, and variable error. Note that after the initial ten seconds, the ball becomes visible and the head starts to track the ball.

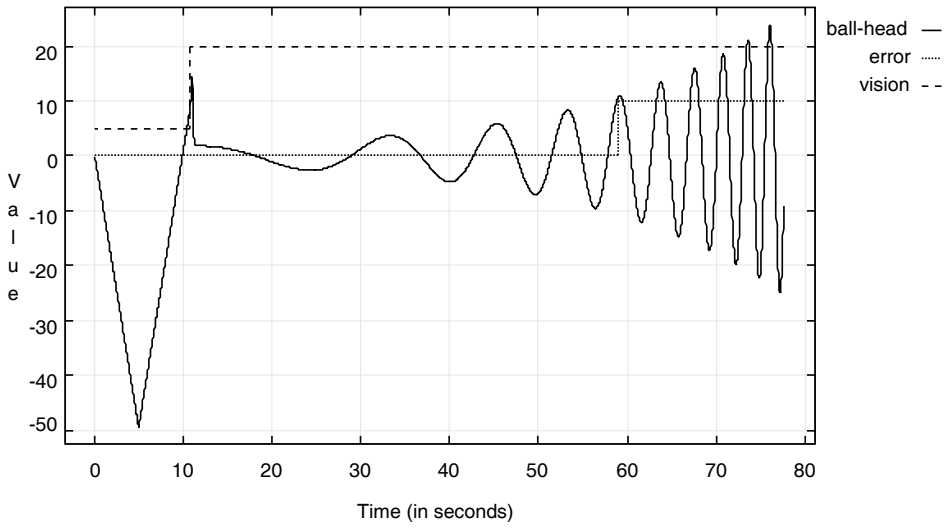

Figure 13: Monitoring head movement

Table 1: Sizes of the tester, the monitor, and the controller

	Tester	Monitor	Controller	+Tester	+Tester+Monitor
CHARON (lines)	81	52	109	190	242
C++ code (lines)	495	466	622	1117	1583
Binary (bytes)	15,496	16,988	661,184	676,680	693,668

However, with the speed of the ball getting faster and faster, there is increasing difficulty for the dog to follow the ball, indicated by increasing distance between the ball and the head, and finally the dog lost track of the ball ($|ball - head| > 10$).

One may be tempted to encode the property of interest as an invariant $vision > 10 \Rightarrow |ball - head| \leq 10$ and reduce monitoring to invariant checking. However, Figure 13 gives us a reason why we need the technique introduced in Section 4. Note that during the time interval $[10.7, 11]$ there is a peak on the curve $ball - head$, and both $vision > 10$ and $|ball - head| > 10$ on this peak. The peak is introduced because the ball was just being seen and the dog has been given time to track the ball. However, if we had simply used invariant checking, the peak would have been reported as the sign that dog cannot track the ball, a "false" alarm one may want to avoid.

Finally, we use the model-based code generator in [15] to generate the code for tester, controller, monitor from $\mathcal{T}||\mathcal{H}||\mathcal{M}$. Because the code generator supports modular compilation, we can compile each module separately and link all or some of them at our will. Table 1 shows the size of the model and the generated code. In [15] the input variables *vision* and *ball* in generated code were mapped to the registers for the sensor unit. In our experiment, these variables are mapped to the output variables of the code for tester. In addition, we link the variable *error* to the input of LEDs on the head, hence when the monitor sees an error, the LEDs on the dog's head starts to blink. We have loaded the generated code into the SONY AIBO dog. The dog acts as if it were chasing a virtual ball, and after 59 second, the LEDs starts to blink, indicating that the dog fails to follow a fast-moving virtual ball.

6 Conclusion

We have proposed an integrated approach to test and monitor model-based generated code. We start with the techniques of generating and encoding tester and monitor in the same language as the system model, then use the same code generating process to synthesize a tester and a monitor from these models. In our approach, the hybrid automaton model for the tester is to model a controlled testing environment we want to test the system, and the model is refined according to both the coverage criteria and the testing goal. The tester supplies test traces on-the-fly, which eliminates the need of loading a relatively big static test suite into the limited memory space of embedded systems. We also introduce a technique to mechanically synthesize monitors from behavior specifications encoded in deterministic passive Büchi timed automata. The tester and the monitor may be used both on the simulation level and on the code level to validate the tested code. Finally, modular compilation supported by our code generator allows us to link the tester and/or the monitor as needed with the tested code. Our approach essentially shows another benefit of model-based design and code generation, that is, it allows the user to rapidly prototype testing and monitoring tasks in the same modeling language, and use the same code generation mechanism to generate code for testing and monitoring purposes.

Our works may be further extended in many ways. First, it would be interesting to see how testers can address the constraints imposed by physical systems. In addition, in this paper the system property is encoded as a variant of timed automata, but our approach may be extended to other logics like MEDL [16] and metric LTL [4]. We are also interested in automatic generation of testing automata from given environment constraints.

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. 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 and D. L. Dill. The theory of timed automata. *TCS*, 126(2), 1994.

[4] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. In *Journal of the ACM*, volume 43, pages 116–146, 1999.

[5] R. Alur, F. Ivančić, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchial hybrid models. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[6] 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.

[7] SONY Corporation. Entertainment Robot AIBO. http://www.aibo.com.

[8] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specification. In *the Fourth ACM SIGSOFT Symposium on the Foundation of Software Engineering*, 1996.

[9] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, Berlin, 1981. Springer-Verlag.

[10] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering*. IEEE Computer Society, 2001.

[11] K. Havelund and G. Rosu. Java pathexplorer - a runtime verification tool. In *Proceedings of The Sixth International Symposium on AI, Robotics, and Automation in Space*, May 2001.

[12] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2002.

[13] Reactive Systems Inc. Reactis. http://www.reactive-systems.com, 2003.

[14] The MathWorks Inc. Real-time workshop for Simulink. http://www.mathworks.com.

[15] J. Kim and I. Lee. Modular code generation from hybrid automata based on data dependency. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS 2003)*, 2003.

[16] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[17] N. A. Lynch, R. Segala, and F. W. Vaandrager. Hybrid I/O automata. In *82*, page 16. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1995.

[18] 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.