Model-based Testing and Monitoring for Hybrid Embedded Systems*

Li Tan Jesung Kim Oleg Sokolsky Insup Lee

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
{tanli, jesung, sokolsky, lee}@ saul.cis.upenn.edu

ABSTRACT

We propose an integrated framework for testing and monitoring the model-based embedded systems. The framework incorporates three components: 1) model-based test generation for hybrid system, 2) run-time verification, and 3) modular code generation for hybrid systems. To analyze the behavior of a model-based system, the model of the system is augmented with a testing automaton that represents a given test case, and with a monitoring automaton that captures the formally specified properties of the system. The augmented model allows us to perform the model-level validation. In the next step, we use the modular code generator to convert the testing and monitoring automata into code that can be linked with the system code to perform the validation tasks on the implementation level. The paper illustrates our techniques by a case study on the Sony AIBO robot platform.

1. INTRODUCTION

An embedded system is a system that reacts to its environment and whose behavior is subject to the physical constraints imposed by the environment. Although embedded systems are increasingly becoming pervasive, the development of responsive embedded systems still remains challenging. To mitigate development difficulties, there has been a spate of model-based design efforts in recent years. The promise of the model-based design paradigm is to develop design models and subject them to analysis, simulation, and validation prior to implementation. Performing analysis early in the development cycle allows one to detect and fix design problems sooner and at a lower cost.

Tools have been developed both in academia [5, 10, 2] and in industry [12] to facilitate the model-based system design. These tools support the limited ability of validation, usually in form of invariant checking. Other more advanced validation and verification techniques such as hybrid system model checking are also being studied, but the scalability of these state-of-art techniques does not yet match the needs of embedded system design. For example, model checking of hybrid systems can be carried out only for very small systems because of the complexity of model checking algorithms. So, to use such model checkers, embedded system models need to be heavily abstracted, and it is not easy to establish that assumptions made in the abstraction process are always valid in the target system. It is therefore highly desirable to obtain a verification and validation technique that is capable of checking properties beyond simple invariants, can handle system models of realistic size, and can be applied both on the model and implementation levels.

The approach we propose in this paper is to integrate the testing and runtime verification into the model-based hybrid embedded system design. We generate the tests using a simulation-based test generator, but the focus of our approach is how to apply such tests to a system model as well as to an implementation, and verify the safety properties during the testing using the runtime verification. We propose the techniques to generate a model-based monitor from a formal specification of the system properties and a model-based tester by the test requirement. Our approach applies to both the design level and the implementation level. For the former, we compose the model-based monitor and tester with the instrumented system model to form a self-testing and self-monitoring model. For the latter, we use the existing model-based code generation mechanism to convert the composed model to executable code, including the code for the tester and the monitor. The salient aspect of our framework is that the design model and the implementation can be evaluated under the same test and runtime monitor. We believe that our framework is necessary in helping to narrow the gap between designs and implementations.

Figure 1 illustrates the overview of our framework. Embedded systems are modeled as hybrid automaton [17] in our framework. For model-based testing, we start with a nondeterministic hybrid automaton that emulates an environment under which the embedded system is to operate. The environment automaton supplies inputs to the system model. The goal of test generation is to fix a subset of environment behaviors, or in terms of environment automaton, a subset of its traces which satisfies the given testing criteria. For a particular test, our framework generates a testing automaton from the environment automaton. Unlike the environment automaton, the testing automaton is deterministic and its only trace is the prescribed test case. Model-based runtime verification is introduced to check in real time whether the execution of a model violates given properties.

We describe the details of our framework in context of CHARON. CHARON is a visual language for modeling hierarchical hybrid automaton. CHARON toolkit has a model simulator and a code generator which can translate CHARON to C++ code. In our framework, system properties are en-

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

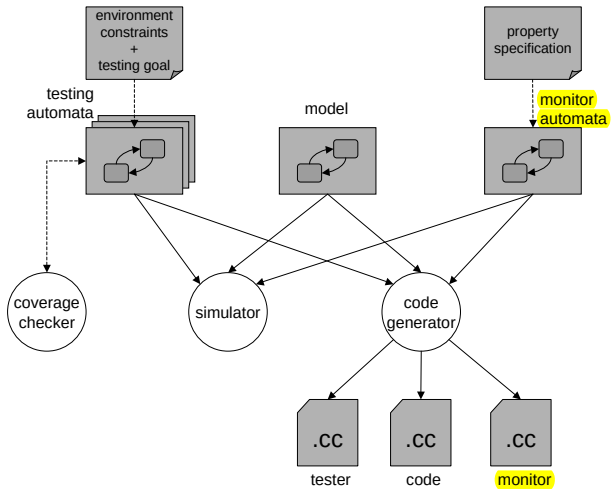


Figure 1: The framework for testing and monitoring model-based generated code

coded in MEDL, a linear temporal logic for specifying safety properties [15]. We provide an algorithm and a tool M^{2}IST that can synthesize a hybrid automaton as a model-based monitor from the MEDL specification. It is then composed with the instrumented system model to check the execution of the system model against the MEDL specification.

The rest of the paper is organized as follows. Section 2 introduces the notations and definitions used in the paper. It also includes a brief introduction to CHARON. Section 3 covers the issue of generating a model-based tester. Section 4 discusses techniques related to model-based runtime verification. It also explains our toolkit M^{2}IST for model instrumentation and model-based monitor synthesis. Section 5 illustrates how this framework can be used for both design-level and implementation-level validation with a case study on a SONY AIBO robot. The last section concludes the paper with discussions on future directions.

Related work. Our work on synthesizing monitors is inspired by the previous research on runtime verification based on formal methods, for instance, MaC [15] and Java PathExplorer [8]. Both tools work on the code level. They are capable of instrumenting Java bytecode, observing events emitted by a running program, and comparing them with formal specification. The approach described in this paper can work at both the model level and the code level; that is, our approach may also combine the monitor model with the system model and run the composed model on a simulator for design-level validation in addition to code-level run-time validation. In [9], the authors show how to synthesize a monitor program directly from formal specification. In contrast, our approach is to synthesize monitors as hybrid automaton and then leave the generation of actual monitor programs to the automatic code generator [4, 13]. In [6] and [7], the authors show how to synthesize automaton-based monitors (test oracle) from temporal logics for systems with discrete events, while we are more interested in handling continuous dynamics of hybrid systems.

2. PRELIMINARIES

2.1 Modeling language CHARON

CHARON is a formal language for modeling hybrid systems [2]. CHARON builds upon the formalism of hybrid automaton [1, 17], extending it in several ways. Hybrid systems are modeled as hierarchical collections of concurrent *agents*, that is, each agent can contain a number of concurrent sub-agents. Each agent is a hierarchical collection of *modes*. A mode is a hybrid state machine made of sub-modes connected by transitions. Sub-modes can, in turn, contain further levels of hierarchy. In addition, agents and modes contain I/O interfaces similar to hybrid I/O automaton of [16]. Interfaces allow us to model open systems and ensure compositionality of the semantics. Precise definition of CHARON and its formal semantics can be found in [3]. For simplicity, in this paper we will identify an agent with its top-level mode and refer to both as a hybrid automaton. For the purpose of this paper, we will use the following definition of a hybrid automaton.

A hybrid automaton $\mathcal{A}$ is a tuple $\{S, X, T, G, W, D, Inv, h_0\}$, where S is a set of *locations*. Locations, in turn can contain automata. X is a set of real-valued *variables*. X is partitioned into sets I, O, P of input, output, and internal variables, respectively. $T \subseteq S \times S$ is a set of *transitions*. The set of guards G assigns to each transition $t \in T$ a guard, denoted as $G(t)$. A guard is a predicate over X. A transition $t \in T$ is enabled when $G(t)$ is true. The set of resets W assigns to each $t \in T$ a reset function. $W(t)$ is a partial function from X to $\mathbb{R}$, specifying how variables of the automaton are changed when a transition is taken. G and W collectively define discrete behavior of $\mathcal{A}$. The set of *flow constraints* D assigns each location a set of differential equations in the form of $\dot{x} = f(X)$[1]. D defines the continuous trajectories for the automaton variables while the automaton stays in a given location. The set of *invariants* Inv assigns each location a set of predicates over X. The automaton can stay in location s as long as $Inv(s)$ is true. A state is defined by a location and a valuation of the variables. $h_0 = \langle s_0, V_0 \rangle$ is the initial state, where $s_0 \in S$ is the initial location and is the initial valuation of the variables. We require that the automaton cannot control its input variables neither by its resets, nor by flow constraints. We also require that for each variable $x \in X - I$, each location defines a flow constraint for x.

An execution of a hybrid automaton A is given by its flow constraints and resets. While A stays in one location s, continuous trajectories for output variables satisfy the flow constraints in s, while input variables may follow an arbitrary piecewise-continuous function. When A takes transition t from one location to another, $W(t)$ specifies the change of some output variables, while all other retain their values. Two automata may be composed in parallel, in which case they synchronize with each other via the shared variable during continuous steps, while discrete transitions can be taken by the automata independently of each other.

Figure 2 shows a hierarchical hybrid automaton modeling a robot dog tracking an object. The variable θ indicates the angle between the head and the red ball, and the variable β is the degree of visibility of the ball. On the top level the automaton has two locations. When the visibility of the ball is greater than the threshold 10, the control jumps to the right top location. The movement of the dog head

[1]CHARON also allows algebraic constraints to be used in defining flows, however this feature is not used here.

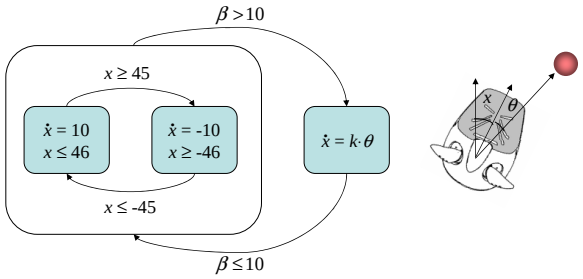


Figure 2: Hierarchical hybrid automaton modeling a robot dog tracking an object.

is controlled by a differential equation $\dot{x} = k \times \theta$, which forces the dog to move its head towards the ball; When β is below the threshold 10, the control jumps to the left top location. This model will be revisited as a simple case study to illustrate our framework step by step.

The CHARON tool set includes a simulator, a code generator, and a simulation-based test generator. Code generation is modular, which allows each component automaton in the model to be translated separately and be linked at the compilation time. In our case study, the model in Figure 2 has been used to generate the controlling program for the Sony AIBO dog.

2.2 Runtime Verification

To check the execution of system models and their implementations, we use a novel formal technique called *runtime verification*. Runtime verification was initially proposed to check whether an execution of a software program violates its safety requirement [15, 8]. A typical runtime verification framework contains three stages. First, a requirement to be checked is formally expressed, for example, in some temporal logic. Second, the program is instrumented to send relevant information to the runtime checker. Finally, when the program is running, the checker will detect any violation of the requirement and raise alarms. In this paper, we build upon a framework for runtime verification that we have previously developed and successfully used in a number of case studies [15]. In particular, we use Meta Event Definition Language (MEDL) as the temporal logic to encode safety requirements.

MEDL defines properties in terms of events and conditions. Intuitively, conditions hold for a certain duration during the execution, while events occur instantaneously. Each event is labeled with the time of its occurrence. Primitive events are supplied by the system during its execution. Complex events and conditions are defined in terms of primitive events. Predicates over event time stamps form primitive conditions and allow us to express real-time requirements. If `e1` and `e2` are events, then `time(e2) - time(e1) < 5` is a condition that is true if the last occurrence of `e2` was within 5 time units after the last occurrence of `e1`. Two distinct events, `e1` and `e2`, can define a condition `[e1,e2)`, which is true from an occurrence of `e1` until the next occurrence of `e2`. Similarly, any condition `c` defines two events, `start(c)` and `end(c)`, which occur when `c` becomes true and stops being true, respectively. Any event can be designated as an alarm, so that an occurrence of this event can be reported to the user as a violation of a requirement during an execution. The formal semantics of MEDL define the outcome of the evaluation of a MEDL formula over a stream of primitive events. A precise definition of MEDL syntax and its formal semantics are defined in [15]. A sample MEDL formula used in our case study can be seen in Section 4.

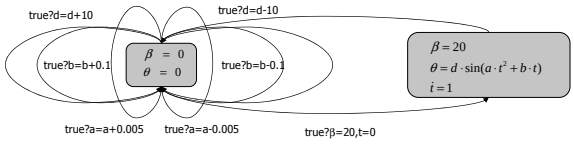


Figure 3: The environment automaton for testing Sony AIBO dog

We use an auxiliary language, Primitive Event Definition Language (PEDL), to define the relationship between low-level run-time observations such as variable assignments and primitive events used in a MEDL formula. PEDL is dependent on the way the system is specified. In Section 4, we will discuss mPEDL, a variant of PEDL suitable for definition of primitive events in the model-based setting, when the system model is defined in CHARON.

3. GENERATING MODEL-BASED TESTERS

The first step of model-based testing is to generate a test suite. For an open system, we will close it by providing an environment automaton that specifies "reasonable" behavior for the environment. The automaton has all input variables of the system model as its output variables.

Consider an environment automaton for the system of Figure 2, shown in Figure 3. In our case study, the environment is essentially the movement of a red ball. The ball may change its visibility as well as its position in the space. The environment automaton models a chaotic environment with some degree of control: the red ball may switch between visible and invisible at any time; the red ball swings before the dog when it is visible. Its speed and acceleration are controlled by random variable a, b, and d, which can change while the ball is invisible, but remain constant once it appears.

A test suite is defined as a finite set of executions of the environment model. We implemented a simulation-based test case generator for CHARON. It simulates the behavior of the automaton starting with the initial valuation V_0 and the initial location s_0. During the simulation, the test generator tries to achieve the desired coverage by resolving the non-determinism in the model in different ways using a randomized algorithm. A test case is obtained from the simulation trace by projecting it on the input variables of the system model. The framework of this paper, however, does not depend on a particular test generation approach, so that we may be able to use other available test case generators for hybrid systems such as [11].

Once the test suite is chosen, we create a testing automaton for each test case in the suite. The purpose of the testing automaton is to supply the test case during the execution of the system model and later its implementation. Given a test case, the testing automaton is the result of restricting the environment automaton to the behavior exhibited by the test case. Interested readers may refer to [18] for a description of the algorithm for constructing a testing automaton from the environment automaton.

One of the generated test cases for the AIBO case study

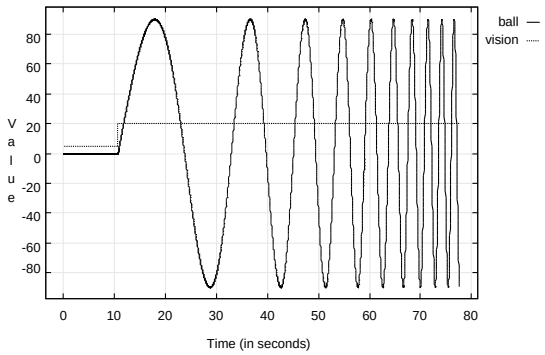


Figure 4: The movement of a red ball: a generated test case

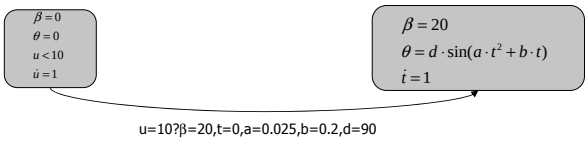


Figure 5: The testing automaton for Sony dog

is shown in Figure 4, where vision and ball are the names of variables in the CHARON models representing the visibility β and the position θ of the red ball. The movement of the ball has two phases. Initially, it is invisible; after 10 seconds, it starts to be visible and waves in front of the dog. This test covers all the locations in the 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. Figure 5 shows the testing automaton. The new clock variable u ensures that the transition from invisible location to visible location is taken at precisely 10 seconds. The only run of the testing automaton is the test case in Figure 4.

4. SYNTHESIZING MODEL-BASED MONITOR

To apply the runtime verification concept in the model-based design, we need to change the last two stages in the standard runtime verification framework. The instrumentation is performed on the system model instead of on the code, and instead of constructing a stand-alone checker, we represent the checker as a hybrid automaton, encoded in the same modeling language as the system model.

Figure 6 presents a MEDL formula used in the AIBO case study. It describes a requirement for object tracking. When an object is visible, the alarm is raised if the dog loses track of the ball 50 seconds after the ball becomes visible. Events `isVisible` and `isInvisible` denote changes in visibility of the object, and event `lost` when the dog's head cannot follow the ball closely enough. Precise definition of these primitive events in terms of the variables of the system model is discussed below.

4.1 Model Instrumentation and mPEDL

MEDL formulas are evaluated over sequences of primitive events, which are emitted during the executions of

```
ReqSpec dogVision
import event isVisible, isInvisible, lost;
condition visible = [isVisible, isInvisible);
event becameTruelost = lost when visible;
alarm lostTrack = start (time(becameTruelost)
  -time(isVisible)>50);
End
```

Figure 6: MEDL script for monitoring SONY Dog

```
MonScr Dog
  /* Export section */
  export event isVisible, isInvisible, lost;
  /* Monitored objects */
  monobj real dog.beta, dog.theta, dog.x;
  /* Predicate definition */
  condition close=|dog.theta-dog.x|<10;
  /* Event definition */
  event  isVisible= start (dog.vision>10);
  event  isInvisible= end (dog.vision>10);
  event  lost = end (close);
End
```

Figure 7: mPEDL script for monitoring Sony AIBO Dog

monitored object, or in our case, in an execution of a hybrid automaton. In our framework primitive events are defined in model-based Primitive Event Definition Language, or mPEDL. mPEDL is a variant of PEDL which has been introduced in the tool Java-MaC [14] for defining primitive events on Java programs. Primitive events in mPEDL are defined as the changes on predicates over the variables of the monitored automaton. Figure 7 gives a sample mPEDL script which defines primitive events for MEDL script in Figure 6 with respect to the hybrid automaton in Figure 2.

A mPEDL script specifies primitive events that are *exported* to the checker. The monitored object section defines the variables which are used for defining primitive events. Consider the script in Figure 7. The script defines two primitive events that denote visibility of the ball, `isVisible` and `isInvisible`. The script also defines event `lost`, which occurs when the angular difference between the direction towards the ball (θ) and the position of the head (x) becomes too large.

To monitor the hybrid automaton, the original model needs to be instrumented to emit primitive events. Events are implemented as shared variables: for each primitive event we introduce a variable that records the time of the most recent occurrence of the event. In addition, a variable *newEvent* is used to signal to the checker that an event has been detected. An observer automaton is introduced for each predicate used in the mPEDL script. Each of such automaton has two locations: in one state the predicate is true and in the other location it is false. A transition between these locations occurs happen when the predicate changes its value and manipulates the time variables for the respective events.

4.2 Generating model-based monitor from MEDL

In our approach a hybrid automaton is synthesized to monitor the system w.r.t. the given MEDL script. This monitoring automaton is composed with the instrumented system automaton and reacts to the events emitted by the system via shared variables. Each event E in the MEDL for-

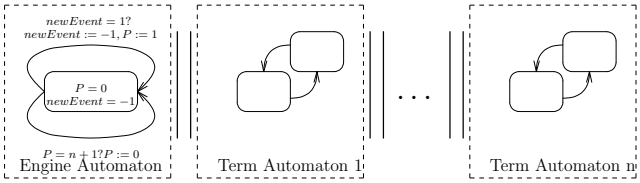


Figure 8: The Monitoring Automaton

mula has a variable V_E in the monitoring automaton recording the last time E occurs. Each expression Q also has a variable V_Q to store its current value. Each condition C has two variables: V_C records the current value of C and $V_{C\updownarrow}$ records the last time C changes its value. The synthesizing process is modular: each term in the formula (condition, expression, or non-primitive event) is translated to a separate automaton which processes the related variables. The automata then synchronize with each other by passing around a *token* that ensures that each term automaton executes only after all its input variables have been processed by other term automata. The engine automaton is triggered by each primitive event and passes the token to the first term automaton. The monitoring automaton is the parallel composition of the term automata and the engine automaton, as shown in Figure 8. The rules to generate the term automata from a given MEDL formula are given in [18].

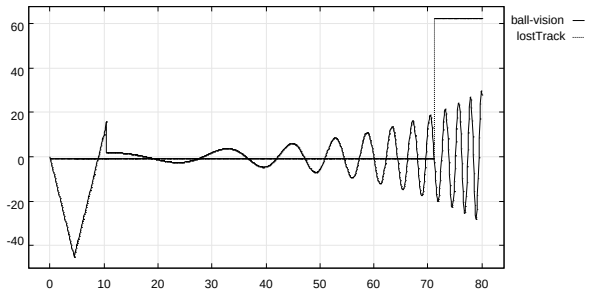
5. VALIDATING MODEL-BASED EMBEDDED SYSTEMS

In this section we describe in more detail the case study on the SONY AIBO Robotic dog, in which we performed validation of the object tracking code generated from the model in Figure 2. The robot consists of both analog devices for inputs and outputs and a digital control system. 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. The dog has a two-dimension light sensor and two step motors, which control the head's vertical and horizontal movement. The two-dimensional light sensor can measure the relative angle between head and a bright object, in our case, a red ball. Input variables in the CHARON model are mapped to platform sensors, and output variables are mapped to motor actuators.

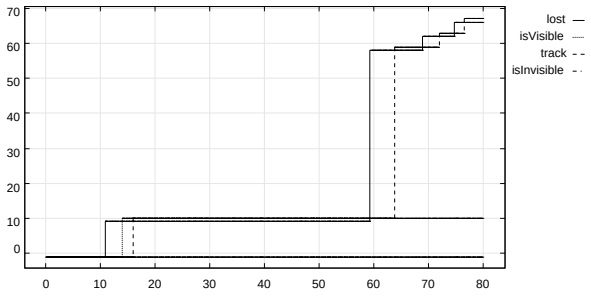
5.1 Design-level Validation

We validate the system model using the CHARON simulator. We concurrently compose the instrumented model with the generated tester (Figure 5) and monitor. An alarm may be detected by observing the value changes on its event variable. Note that the instrumentation introduced into the model passively observes the values of model variables and do not constrain the execution of the system model. Thus an execution of the instrumented model, projected to the variables of the system model, is an execution of the system model.

The CHARON model for the hybrid automaton in Figure 2 has 221 lines of code. The model is instrumented by the tool P2C. P2C generates an observer from the mPEDL



(a)



(b)

Figure 9: Design-level validation for the SONY AIBO dog

script in Figure 7 and adds the observer to the right place in the CHARON model. The instrumented model contains 240 lines of code. The monitor is synthesized from the MEDL script in Figure 6 using the tool M2C, also a part of the M^2IST toolkit. The monitor has 295 lines of CHARON code. The generated tester has 81 line of CHARON code. The size of the composed self-testing and self-monitoring model has 622 lines of CHARON code.

Figure 9 shows the result of the design-level validation. Figure 9 (a) shows the simulation trace of the composed model: during the initial 10 seconds, the dog swings its head because the ball invisible. After 10 seconds the dog starts to chase the ball. With the speed of the ball increasing, the dog has increasing difficulty in chasing the ball, as indicated by the growing angle between the dog head and the ball ($ball - vision$ in Figure 9), and finally the jump on the event variable $lostTrack$ at time 70 indicates the occurrence of the alarm $lostTrack$. Figure 9 (b) shows the primitive events emitted during the simulation. Again, each jump on event variables indicates the occurrence of an event, and their values indicate when the event occurs.

5.2 Implementation-level Validation

The code generator [13] we are using supports modularity compilation, which allows each component automata in a CHARON model to be compiled separately and linked as needed. Instead of the straightforward way to generate embedded code from the aforementioned self-testing and self-monitoring model, we generate the system code, the monitor code, and the tester code separately from the instru-

	Controller	+Tester	+Tester+Monitor
CHARON (lines)	221	303	622
C++ code (lines)	873	1278	3969
Binary (bytes)	470,886	485,327	544,259

Table 1: The size of the generated codes

mented system model, the synthesized monitor model, and the tester. They can be linked according to the validation plan. We may link the monitor code with the system code for a self-monitoring code. This self-monitoring code performs the same as the original system, except that the monitor is constantly checking the execution of the program and raises the alarm *lostTrack* if necessary. Or we may link all of three together to check the reaction of the dog on test inputs. Figure 1 gives the size of programs in different configurations.

On the design-level simulation, events are observed as the changes on the corresponding event variables; On the implementation level, we do not usually have the access to the values of variables in an embedded program. Nevertheless, we can do something more creative: the changes of event variables may be used to trigger some visible actions on the tested platform. In our case study event variable *lostTrack* has been used to activate the "play" function which makes the dog bark when event *lostTrack* occurs. We have loaded the generated self-testing and self-monitoring code to Sony AIBO dog. The dog moves its head as predicted by Figure 9 (a) and starts to bark at 70 seconds, just as expected.

6. CONCLUSIONS

We have proposed an integrated framework to test and monitor model-based hybrid embedded programs. Our approach works directly on models, hence it does not require changes to the existing design tools. We discuss the set of techniques necessary for supporting this new approach: in model-based testing we generate a model-based tester which supplies a test case to the system model; in model-based monitoring we instrument the system model and synthesize a model-based monitor from the MEDL specification of the safety properties. Our framework provides both design-level and implementation-level validations for hybrid embedded programs. For the design-level validation, we compose the instrumented model with the synthesized monitor and run the composed self-monitoring model on a simulator. For the implementation-level validation, the existing code generation mechanism is deployed to generate the self-monitoring executable code from the aforementioned model. Our approach yields a hardware-specific tester and runtime verifier which can run on the targeted hardware platform for "on-board" validation. Last but not least, we also provide the tools support for model-based runtime verification.

Our work may be extended in several ways. For example, it would be interesting to see how our approach works on some industrial model-based design tools like Simulink and Stateflow. We are also trying to optimize the tester-generating algorithm to produce smaller testing automata.

7. 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*, 91(1):11–28, 2003.

[3] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, March 2001.

[4] 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.

[5] A. Chutinan and B.K. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems: Computation and Control, Second International Workshop*, LNCS 1569, pages 76–90, 1999.

[6] 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.

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

[8] K. Havelund and G. Rosu. Monitoring Java programs with JavaPathExplorer. In *Proceedings of the Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.

[9] 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.

[10] T.A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1, 1997.

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

[12] The MathWorks Inc. Simulink, stateflow, and real-time workshop. http://www.mathworks.com.

[13] 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.

[14] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In *Proceedings of Workshop on Runtime Verification (RV'2001)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, July 2001.

[15] 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.

[16] N. A. Lynch, R. Segala, and F. W. Vaandrager. Hybrid I/O automata. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1995.

[17] 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.

[18] Li Tan. Model-based self-monitoring embedded programs. In *submitted for publication*, 2004.