Model-Based Self-Monitoring Embedded Programs With Temporal Logic Specifications*

Li Tan
The MathWorks Inc.
3 Apple Hill
Natick, MA 01760
ltan@mathworks.com

ABSTRACT

We propose a model-based framework for developing self-monitoring embedded programs with temporal logic specifications. In our framework the requirement specification of an embedded program is encoded in the temporal logic MEDL. We propose an algorithm that synthesizes a model-based monitor from a MEDL script. We also introduce a technique that instruments a system model to emit events defined in the model-based primitive event definition language mPEDL. The synthesized model-based monitor may be composed with the instrumented model to form a self-monitoring model, which can be simulated for design-level verification; the composed self-monitoring model can also be used to generate a self-monitoring embedded program, which can monitor its own execution on the target platform in addition to its normal functions. Our approach combines the rigidness of temporal logic specifications with the easy use of a toolkit M^{2}IST that we developed to automate the process of building a self-monitoring embedded program from a system model and its requirement specification.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*

General Terms

Algorithms,Design,Verification

Keywords

Run-Time Verification, Embedded Systems, Temporal Logic

*Part of the work was done when the author was visiting the University of Pennsylvania. This research was partly supported by NSF CCR-0086147, NSF CCR-0209024, and ARO DAAD19-01-1-0473

ASE'05, November 7–11, 2005, Long Beach, California, USA.

1. INTRODUCTION

Embedded systems are now part of our everyday life. Ensuring the correct functioning of these systems also becomes ever challenging. Model-based design [2, 8, 9] is embraced by research and engineering communities as a promising approach to develop dependable embedded systems. In model-based design engineers start with the model of a system, hence they can analyze the design before its implementation. Model-based design also facilitates rapid prototyping by means of automatic model-based code generation. Tools such as Simulink/Stateflow [8], GME[9], and Charon [2] have been developed in academia and industry to support model-based design. It is important to ensure that an embedded system model as well as its implementation meets its requirements. We propose a model-based framework for building self-monitoring embedded systems, which checks its own execution against a temporal logic-based specification.

Our work is inspired by software runtime verification. Runtime verification checks the execution of a program against a formal requirement [3, 4, 7]. Expanding it to the domain of model-based embedded systems presents several challenges: first, since many embedded systems consist of both analog and digital devices, they are often modeled as hybrid automata whose executions contain both discrete and continuous elements, in contrast to discrete executions of software programs. Temporal logics initially used in the discrete domain must be extended to reflect such shift. Second, Key techniques in software runtime verification such as runtime monitor and program instrumentation have to be replaced by new techniques targeted for model-based design. Last but not least, we need to devise a process that can be fully automated so industrial practitioners may potentially take advantage of this new technique.

In our framework, requirements are encoded in a temporal logic MEDL to avoid ambiguity. Our model-based approach integrates a runtime verifier (monitor) to a system model: we design an algorithm that synthesizes a model-based monitor from a MEDL script. Next, we instrument a system model to emit primitive events. Finally, we compose the instrumented model with the monitor to form a self-monitoring model. The self-monitoring model may be simulated on an existing model simulator for design-level verification and can be used to generate a self-monitoring embedded program for implementation-level verification. We also developed a toolkit M^{2}IST to automate the process. M^{2}IST provides two utilities: M2C synthesizes a monitor from a MEDL script, and P2C instruments a model and composes it with the monitor to form a self-monitoring model.

The rest of paper is organized as below. Section 2 provides a brief introduction to hybrid automaton and the modeling language CHARON. Section 3 defines the temporal logic MEDL and its semantic extension in context of hybrid automaton. It also introduces the model-based primitive event definition language mPEDL. Section 4 discusses the techniques essential for building self-monitoring models. These techniques include a model-based monitor synthesis algorithm and a model instrumentation technique. Section 5 discusses design-level and implementation-level runtime verifications facilitated by our model-based approach. Section 6 introduces M^{2}IST, a toolkit we developed to automate the building process. Finally, in section 7 we conclude this paper. The proofs in this paper are removed to save space and may be found in the full version of the paper at www.cis.upenn.edu/∼tanli/papers/modelmonitor-full.ps

2. PRELIMINARIES

An embedded system generally has both discrete and continuous behaviors. Such a system is often modeled as a hybrid automaton [1, 5]. A hybrid automaton is a tuple $\{S, V, \rightarrow, G, W, s_0, \mathbf{v_0}, D, I\}$, where $\langle S, V, \rightarrow, G, W, s_0, \mathbf{v}_0\rangle$ is an Extended Finite State Machine (EFSM) with a set of locations (modes) S, a set of variables V, the transition relation $\rightarrow$, the guards G as predicates over $\rightarrow$, the assignments W setting new values to V upon a transition, the initial location s_0, and the initial valuation $\mathbf{v}_0$. There are two elements which distinguish a hybrid automaton from a EFSM: D associates each mode with a set of differential equations, and I imposes on a mode an invariant as a predicate over V. A state $\langle s, \mathbf{v}\rangle$ of a hybrid automaton is identified by its location and valuation of variables. A run of a hybrid automaton is a hybrid trace, which 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 $\mathcal{V}_i$ is the flow of variables' values at a mode s_i and $\mathcal{I}_i$ is the duration of ρ's stay at s_i. $\mathcal{V}_i$ must also respect s_i's differential equations $D(s_i)$ and satisfy its invariant $I(s_i)$.

Hybrid automata can be composed parallel to model the architectural hierarchy of a system. Composing two hybrid automata is much like composing two EFSMs. Modes of the composed automaton $M_0||M_1$ are the products of modes of each component automaton. They also inherit differential equations and invariants imposed on both M_0 and M_1. Hybrid automata may also be composed hierarchically to model the behavioral hierarchy. In a hierarchical hybrid automaton, a mode may be another hierarchical hybrid automaton. Hybrid automata have been used for modeling and simulating control systems consisting of multiple control laws. Hybrid-automaton-based modeling languages such as CHARON and their companion tools have been successfully used to model and develop embedded systems.

Modeling Language CHARON CHARON [2] is a modeling language based on hybrid automata. CHARON supports both structural and behavioral hierarchy by allowing the parallel and hierarchical composition of hybrid automata. CHARON design environment [2] provides, among other functionalities, a simulator and a code generator. The code generator can convert a CHARON model to C/C++ code. We choose CHARON as the target modeling environment in which we implement our framework and demonstrate its feasibility and benefits.

$\langle C\rangle$ ::= $[\langle E\rangle, \langle E\rangle) \mid \langle C\rangle \&\& \langle C\rangle \mid \langle C\rangle||\langle C\rangle \mid \langle Q\rangle \square \langle Q\rangle$
$\langle E\rangle$ ::= $e \mid \mathsf{start}(\langle C\rangle) \mid \mathsf{end}(\langle C\rangle) \mid \langle E\rangle||\langle E\rangle \mid \langle E\rangle \&\& \langle E\rangle$
$\mid \langle E\rangle \mathsf{when} \langle C\rangle$
$\langle Q\rangle$ ::=$\mathsf{time}(\langle E\rangle) \mid q \mid \langle Q\rangle \diamond \langle Q\rangle$

where q is a constant, $\diamond \in \{+, -, *, /\}$ and $\square \in \{>, <, =\}$ are arithmetical and relational operators, respectively

Table 1: The syntax of MEDL[4]

3. SPECIFYING SYSTEM REQUIREMENTS IN MEDL AND mPEDL

We specify system requirements in the temporal logic MEDL and the model-based primitive event definition language mPEDL. A MEDL script encodes a system requirement as temporal patterns of primitive events and an mPEDL script defines primitive events as changes of states in a hybrid automaton. In Section 3.1 we extend the semantics of the original MEDL [4] from the discrete-time domain to the continuous-time domain. In Section 3.2 we extend PEDL to mPEDL in context of hybrid automaton.

3.1 The Temporal Logic MEDL and Its Semantics Extension

The syntax of MEDL is given in Table 1. The building blocks of MEDL are *events* $\langle E\rangle$- things that occur at some time instance; and *conditions* $\langle C\rangle$- facts that last for certain duration. MEDL is interpreted on sequences of primitive events, which represent the status changes of a monitored system. MEDL is originally introduced for specifying requirements of software programs whose executions can be discretized by steps, and hence primitive events are emitted on a discrete time line. In the case of hybrid automaton, however, primitive events are emitted on a *continuous* time line: a sequence of primitive events occurring during a hybrid trace ρ is denoted by $\epsilon(\rho) = \langle \mathcal{E}_0, t_0\rangle\langle \mathcal{E}_1, t_1\rangle\cdots$, where $t_0 t_1 \cdots$ is a strictly increasing sequence on $\mathbb{R}^+$, and $\langle \mathcal{E}_i, t_i\rangle$ is a set of primitive events $\mathcal{E}_i$ emitted at t_i. Definition 1 extends the semantics of MEDL to the continuous time domain. We use the following notations: we write $\mathcal{I}^l$ and $\mathcal{I}^h$ for the start and the end of an interval $\mathcal{I}$. We denote $\rho\{\mathcal{I}\}$ and $\rho(t)$ for part of a hybrid trace ρ during an interval $\mathcal{I}$ and at a time t, respectively. $\rho\{\mathcal{I}\} \models C$ and $\rho(t) \models E$ indicate that on the hybrid trace ρ the condition C holds during the interval $\mathcal{I}$ and the event E occurs at the time t. The value of an expression Q at a time t is denoted by $[Q](t)$.

Most of semantic definitions in Definition 1 are straightforward: for instance, a condition $C_0 \&\& C_1$ holds when both C_0 and C_1 hold. An event E_0 && E_1 occurs when E_0 and E_1 occur *simultaneously*. The condition $[E_0, E_1)$ holds from the event E_0 to (not including) the event E_1.

DEFINITION 1. *Let ρ be a hybrid trace and $\epsilon(\rho) = \langle \mathcal{E}_0, t_0\rangle \langle \mathcal{E}_1, t_1\rangle \cdots$ a continuous primitive event sequence occurring during ρ, then,*

- **(Conditions)**
  1. *$\rho\{\mathcal{I}\} \models C_0 \&\& C_1$ iff $\rho\{\mathcal{I}\} \models C_0$ and $\rho\{\mathcal{I}\} \models C_1$*
  2. *$\rho\{\mathcal{I}\} \models C_0 || C_1$ iff $\rho\{\mathcal{I}\} \models C_0$ or $\rho\{\mathcal{I}\} \models C_0$.*

3. $\rho\{\mathcal{I}\} \models \neg C_0$ *iff* $\rho\{\mathcal{I}\} \not\models C_0$.
4. $\rho\{\mathcal{I}\} \models [E_0, E_1)$ *iff there exists a* $\mathcal{I}'$ *such that* $\mathcal{I} \subseteq \mathcal{I}'$, $\rho(\mathcal{I}'^l) \models E_0$, *and* $\nexists t \in \mathcal{I}'.\rho(t) \models E_1$.
5. $\rho\{\mathcal{I}\} \models Q_0 \square Q_1$ *iff* $\forall t \in \mathcal{I}.[Q_0](t) \square [Q_1](t)$.

- **(Events)**

1. $\rho(t) \models E_0||E_1$ *iff* $\rho(t) \models E_0$ *or* $\rho(t) \models E_1$.
2. $\rho(t) \models E_0\&\&E_1$ *iff* $\rho(t) \models E_0$ *and* $\rho(t) \models E_1$.
3. $\rho(t) \models \mathsf{start}(C)$ *iff* $\lim_{u\to t^-} \rho\{[u,t)\} \not\models C$ *and* $\lim_{u\to t^+} \rho\{(t,u]\} \models C$.
4. $\rho(t) \models \mathsf{end}(C)$ *iff* $\lim_{u\to t^-} \rho\{[u,t)\} \models C$ *and* $\lim_{u\to t^+} \rho\{(t,u]\} \not\models C$.
5. $\rho(t) \models E$ when C *iff* $\rho(t) \models E$ *and* $\lim_{\delta\to 0^+} \rho\{[t-\delta, t+\delta)\} \models C$.
6. $\rho(t) \models e$ *if* $\exists i.(t = t_i) \wedge (e \in \mathcal{E}_i)$.

- **(Expressions)**

1. $[\mathsf{time}(E)](t) = t'$ *such that* $\rho(t') \models E$ *and* $\nexists u \in [t', t).\rho(u) \models E$, *that is,* $\mathsf{time}(E)$ *records last occurrence of* E *before* t.
2. $[Q_0 \diamond Q_1](t) = [Q_0](t) \diamond [Q_1](t)$.

A condition C *holds on* ρ *iff* $\rho([0,\infty)) \models C$ *and an event* E *occurs on* ρ *iff* $\exists t.\rho(t) \models E$.

MEDL as a script language has a richer syntax. For instance, a MEDL script $\mathcal{P}$ has a special type of events called *alarm*s. The occurrence of an alarm indicates the violation of safety properties in $\mathcal{P}$. Therefore, a hybrid automaton M violates $\mathcal{P}$ if an alarm raises on a hybrid trace of M.

3.2 Model-Based Primitive Event Definition Language mPEDL

We define primitive events in the Model-Based Primitive Event Definition Language mPEDL, which extends the PEDL [4] in context of hybrid automaton. Figure 1 shows an mPEDL script used in our case study to define primitive events on a hybrid automaton model of a robotic dog chasing a ball. An mPEDL script consists of the following sections.

- Export section declares primitive events.
- Monitored object section declares variables and transitions being monitored. The names of these variables and transitions reflect the structural hierarchy of the target hybrid automaton. For instance, *dog.vision* refers to the variable *vision* defined at the top-level mode in a model *dog*.
- Condition section defines conditions as predicates over the monitored variables
- Event section defines primitive events as changes of predicates or conditions. For instance, the events *visible* and *invisible* are defined as on and off of the predicate $dog.vision > 10$.

Overall, the mPEDL script in Figure 1 defines four events: *lost*, *track*, *visible*, and *invisible*. The event *lost* (*track*) occurs when the dog moves away from (close to) the red ball; the event *visible* (*invisible*) occurs when the ball becomes visible (invisible).

```
export event isVisible, isInvisible, \
             lost, track;
monobj int dog.vision,dog.theta,dog.x;
condition tracking=(dog.theta-dog.x<10) \
       &&(dog.theta-dog.x>-10);
event  isVisible= start (dog.vision>10);
event  isInvisible= end (dog.vision>10);
event  track = start (tracking);
event  lost = end (tracking);
```

Figure 1: A sample mPEDL script

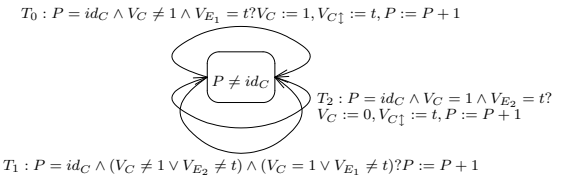

Figure 2: Translation rule for $C = [E_1, E_2)$

4. BUILD SELF-MONITORING MODELS

In traditional runtime verification, runtime verifier is integrated into the programming language's runtime environment. In context of model-based design, this would mean that a model simulator need be modified to include a runtime verifier, which is often costly if not all impossible, since many model simulators contain proprietary code not in the public domain. In [10] we extend a monitored model to include a model-based runtime verifier (monitor). The model-based monitor checks the primitive events emitted by an instrumented model. Here we formalize the process in [10] and introduce an algorithm that can synthesize a *monitoring automaton* $\mathcal{M}$ from a MEDL script $\mathcal{P}$.

4.1 Synthesize Model-Based Monitor from MEDL

Synthesis is carried out in a compositional fashion. Each term (event, condition, or expression) is related to some variables which record its history and value. For instance, Each condition C has two variables: V_C for C's current value and $V_{C\updownarrow}$ for the last time C changes its value. Each term T is translated to an automaton $\mathcal{M}_T$ which updates T's related variables upon incoming primitive events. We assign each term automaton an *id* based on the term's dependency in $\mathcal{P}$. All the term automata shares a token variable P which, together with *id*s, implement a Round-Robin ring: a term automaton $\mathcal{M}_T$ is enabled only if P matches $\mathcal{M}_T$'s *id*. Only by then all the variables related to the terms T depends on have already been updated. Figure 2 lists the translation rule for $C = [E_1, E_2)$ as an example. The rest of rules are similar and may be found in the full version of the papers.

The monitoring automaton $\mathcal{M}$ is a parallel composition of term automata and an *engine* automaton, which checks incoming primitive events and passes the *token* to the first term automaton on the ring to start event processing. Interested readers may refer to the full version of the paper for the details of translation and the proof for Theorem 1.

THEOREM 1. *Let* $M||\mathcal{M}$ *be a self-monitoring hybrid automaton where* $\mathcal{M}$ *is the monitoring automaton for the MEDL*

script $\mathcal{P}$, ρ be a hybrid trace of M, and $\epsilon(\rho) = \langle \mathcal{E}_0, t_0 \rangle \langle \mathcal{E}_1, t_1 \rangle \cdots$ be a primitive event sequence occurring during ρ, then,

(1) (Guaranteed event detection). For every event E in $\mathcal{P}$, $V_E = t_i$ sometime on ρ iff $\rho(t_i) \models E$.

(2) (Guaranteed condition checking). For every condition C in $\mathcal{P}$, $V_C = 1$ sometime on ρ iff there exists an interval $\mathcal{I}$ such that $\rho\{\mathcal{I}\} \models C$.

4.2 Model Instrumentation

To emit primitive events, the original system model has to be instrumented. An event is emitted via a shared variable in the instrumented model which records its last occurrence. We propose an instrumentation technique called *model augmentation*, which extends the original model with *an observer*. This way the instrumentation introduces little disturbance to the structure of the original model.

The observer is a parallel composition of a collection of a simple hybrid automata, each of which represents a predicate(condition) in the mPEDL script and contains only two models: one for the case that the condition is true; and the other for the case it is false. The transitions between two modes emit primitive events by updating the related event variables.

5. MODEL-BASED RUNTIME VERIFICATION

As one of its benefits, our approach support both design-level and implementation-level runtime verification using existing model-based design tools: a self-monitoring model can be simulated for design-level verification on an existing model simulator; and it may also be used by an existing model-based code generator to generate a self-monitoring embedded program, which can execute on the target hardware for implementation-level verification.

In Section 4 we developed an automated process to synthesize a model-based monitor and instrument a system model. Parallel composing the synthesized monitor with the instrumented model forms a self-monitoring model. When simulated, the self-monitoring model emits events that are intercepted by the integrated monitor. Conditions and events can be observed by changes of their related variables' value. For instance, a value change on an *alarm* variable indicates an occurrence of the alarm. When this happens, we know that the model fails design-level verification.

Since a self-monitoring model itself is also a model in the chosen modeling language, it can be used by an existing model-based code generator to produce an embedded program. The generated program is already instrumented on model level and has an integrated monitor. During its execution on the target hardware, the generated self-monitoring embedded program monitors itself besides its normal functions. On the implementation level, however, we generally don't have access to real-time values of variables so we cannot check events or conditions by simply looking at values of their related variables. Instead, we use event (alarm) variables' values to trigger outputs on the target hardware. In this way, we will know if an event occurs and when.

6. TOOLS SUPPORT

We developed M^{2}IST, a toolkit that implemented the monitor synthesis algorithm in Section 4.1 and the model instrumentation technique in Section 4.2. M^{2}IST consists of the model instrumentation tool P2C and the model-based monitor synthesis tool M2C. It provides a "push-button" approach to build a self-monitoring model from a system model in CHARON and its requirements in MEDL and mPEDL. More information about M^{2}IST may be found at [6].

M^{2}IST has been successfully used in several case studies including the one on SONY AIBO Robot, in which we use P2C to instrument a CHARON model of the robot chasing a moving ball, and use M2C to synthesize a model-based monitor from requirements encoded in MEDL and mPEDL. We simulate the self-monitoring model on the CHARON simulator to observe when the robot fails to catch the ball. The self-monitoring model is also used to generate an embedded program executed on the SONY AIBO Robot. The alarm variable is used to trigger a flashing light on the robot to indicate failure during execution.

7. CONCLUSIONS

We proposed an model-based runtime verification framework with temporal logic requirements to monitor embedded system models and its implementations. Our framework has several benefits: first, using temporal logics MEDL and mPEDL allows a lucid and rigid representation of requirement specifications. Second, the process of building self-monitoring models can be fully automated by monitor synthesis and model instrumentation techniques introduced in Section 4. Finally, our model-based framework provides both design-level and implementation-level runtime verification by using existing model-based development tools. We also developed a toolkit M^{2}IST to support our framework. M^{2}IST can build a self-monitoring model from a CHARON model and its requirements in MEDL and mPEDL.

8. 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. *Theo. Comp. Sci.*, 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. *Proc. of IEEE*, 91:11– 28, 2003.

[3] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. In *RV'01*, 2001.

[4] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: a run-time assurance tool for Java. In *RV'01*, 2001.

[5] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *REX Workshop*, LNCS 600, 1991.

[6] M^{2}IST toolkit. University of pennsylvania. In *http://www.cis.upenn.edu/∼tanli/tools/mist.html*, 2003.

[7] A. K. Mok and G. T. Liu. Efficient run-time monitoring of timing constraints. In *RTAS'97*, 1997.

[8] Simulink and Stateflow. The MathWorks, Inc. In *http://www.mathworks.com*.

[9] J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE computer*, pages 110–111, 1997.

[10] L. Tan, J. Kim, and I. Lee. Testing and monitoring model-based generated program. In *RV'03*, volume 89 of *Electronic Notes in Theo. Comp. Sci.*, 2003.