Model-Based Self-Adaptive Embedded Programs with Temporal Logic Specifications [*]

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

ABSTRACT

We propose a model-based framework for developing a self-adaptive embedded program, which monitors its own execution and reconfigures itself at runtime to avoid failure and improve performance. Our approach uses formal methods at different design stages to reduce the complexity of developing a self-adaptive embedded program. In our framework system requirement is rigidly encoded in temporal logics, and the original embedded system behavior is captured in a hybrid automaton-based model. We introduce the reconfiguration specification language REDL to specify reconfiguration requirements, and define a formal semantics of reconfiguration in context of hybrid automaton. Using formal methods also helps automate design and implementation: we use model-based runtime verification techniques introduced in [19] to extend a system model to a self-monitoring model based on its temporal logic requirements; we then extend the self-monitoring model with a reconfiguration mechanism based on its REDL specification. Our approach works with models, and hence it may be incorporated into existing model-based design workflow: the resulting self-adaptive model can be analyzed using an existing model simulator and may be used to generate a self-adaptive embedded program for targeted platform.

1. INTRODUCTION

Since their debut in the early 60s, embedded systems play an indispensable role in many mission-critical applications such as NASA Apollo mission. They are expected to function correctly under often harsh and sometimes even unexpected environment. Many embedded systems either lack the interfaces for software upgrade, or their missions inherently require real-time adaptivity [13]. There are growing interests in self-adaptive embedded software for applications such as Unmanned Aerial Vehicles (UAV), DARPA in 1997 proposed an initiative which calls for more research on self-adaptive software. Two key features of a self-adaptive software are (1) the ability to evaluate its own execution; and (2) the ability to reconfigure itself based on its execution status. The idea of self adaption may be dated back to the adaptive control theory in the 1960s and its biggest success is still limited to the control theory field. Despite of its merits, developing self-adaptive embedded software turns out to be notoriously difficult. Self adaption adds one more dimension of complexity to system design, which makes it much harder to implement and validate a self-adaptive system design. Just as a yardstick on how hard the problem could be, it took the control society over 20 years to prove the stability of self-adaptive controllers. Lack of a formal process also contributes to difficulty in developing high quality self-adaptive embedded software.

We propose a formal approach to rigidly encode reconfiguration requirement and automate most part of self-adaptive embedded program development. Figure 1 shows an overview of our approach. We introduce the reconfiguration configuration language REDL to formally specify requirements for reconfiguration, and use the temporal logic MEDL to specify events triggering reconfiguration. The approach uses model-based runtime verification technique we developed in [17] to extend a system model with self-monitoring capability. Events generated by the self-monitoring model are used to trigger reconfiguration. The approach provides a formal process to further extend a self-monitoring model with reconfiguration capability based on a REDL specification. Our approach makes the following contributions:

1. We introduce the reconfiguration configuration language REDL and define its semantics in context of hybrid automaton-based models. As a replacement for ad-hoc specifications, a REDL script rigidly defines reconfiguration requirements. REDL enables formal analysis on reconfiguration requirements and helps automate the process of building a self-adaptive system. We formalize a process that extends a system to a self -adaptive model based on its REDL specification.

2. Our approach can be integrated into existing model-based design workflow. Our approach transforms a regular design to a self-adaptive design based on reconfiguration requirements. The resulting self-adaptive model can be analyzed using existing model-based tools such as a model simulator. It can also be used to generate self-adaptive embedded systems using an existing model-based code generator.

3. We use model-based runtime verification we developed in [17] to build runtime evaluation mechanism. We specify system requirements in the temporal logic MEDL and define primitive events - the meta events directly tied to the status of an embedded system - in the event definition language mPEDL. MEDL identifies a triggering event as a temporal pattern exhibited by sequences of primitive events. The expressive power of

[*]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 CR-0209024, and ARO DAAD19-01-1-0473.

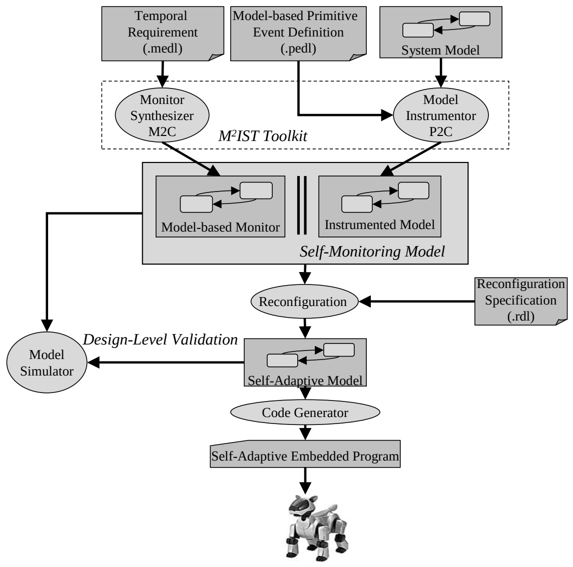


Figure 1: A model-based framework for building self-adaptive embedded programs

> MEDL goes beyond simple failure detection in most self-adaptive system implementation. It allows us to define triggering events with subtle temporal patterns. Model-based runtime verification also provides a tool [11] to automate the process of building a self-monitoring model from the original system model.

The rest of paper is organized as follows. Section 2 provides a brief introduction to hybrid automaton models for embedded systems. Section 3 discusses model-based runtime verification which facilitates on-the-fly evaluation of system status. Section 4 introduces the reconfiguration definition language REDL and its semantics in context of hybrid automata. Section 5 discusses the technique of extending a model with reconfiguration mechanism specified in a REDL script. Section 6 shows how our model-based approach can be incorporated into an existing design workflow to validate the design and facilitate automatic generation of a self-adaptive embedded program from the model. Finally Section 7 concludes this paper.

Related works. The idea of self adaption may be dated back to adaptive control theory in 1960s [12]. Until recently, ad-hoc approaches are still prevalent in self-adaptive software development. DARPA announced its initiative on self-adaptive software in 1997 which helped bring research attention to this important subject. In [13] authors outlined in a casual and informal way an infrastructure to support self-adaptive software but lacked a detailed account on techniques necessary to realize such infrastructure, while here we formalize a process from encoding requirements to finally building a self-adaptive embedded system. In [6] authors used StateChart to model adaptive logic, which they called a "supervisory-level" layer. Here we model the adaptive logic using hybrid automata, which allows us to work with the continuous-time domain. In [16] Strunk and Knight proposed *assured reconfiguration.* Their approach emphasized on architecture level and has a coarse-grained view of systems as a cluster of fail-stop computers. While their main interests were to formally assure reconfiguration for ultra-dependable systems and their abstraction made assured systems discrete and hence suitable for formal analysis, we target at embedded system design with continuous dynamics. We also provide a formal engineering approach which can be integrated into model-based design paradigm.

The real-time evaluation part of our framework is based on our model-based runtime verification technique. Runtime verification [5, 7] is to on-the-fly check the execution of a software program against its formal specification. In [18] we extended it to model-based design. Instead of using events emitted by a runtime verifier to indicate errors, we now use them to trigger reconfiguration.

2. PRELIMINARIES

An embedded system usually consists of digital components and analog devices. It has both discrete and continuous behaviors. Formally, such a system is modeled as a hybrid automaton [1, 10]. A hybrid automaton is an extension of traditional extended finite state machine (EFSM) with continues dynamics. Formally a hybrid automaton is a tuple $\mathcal{A} = \{S, V, T, G, W, D, I, h_0\}$. Just like its discrete counterpart, a hybrid automaton has a set of locations(modes) S, a set of variables V, a set of transitions T, and an initial state h_0. A transition $t \in T$ is also associated with a predicate over V as its guard $G(t)$ and an assignment action $W(t)$ which updates V's values upon the transition t. Unlike an EFSM, a hybrid automaton associates each state s with a set of differential equations $D(s)$ and an invariant $I(s)$. $D(s)$ specifies how variables change values when s is active, and the invariant $I(s)$ must hold when the control stays at s.

A state $\langle s, \mathbf{v}\rangle$ of a hybrid automaton shall specify the active mode s and the current valuation of V. Unlike traditional finite state machines that work in the discrete-time domain, hybrid automata are interpreted in the continuous-time domain. A trace of a hybrid automaton $\mathcal{A}$ can be expressed as a sequence $\rho = \langle s_0, \mathcal{V}_0, \mathcal{I}_0\rangle\langle s_1, \mathcal{V}_1, \mathcal{I}_1\rangle \cdots$, where $\mathcal{I}_i$ is the time interval when the control stays at s_i, and $\mathcal{V}_i : \mathcal{I}_i \rightarrow (V \rightarrow \mathbb{R})$ describe how variables change their value during $\mathcal{I}_i$. Interested reader may refer to [19] for a more detailed explanation of hybrid automaton.

Hybrid automata can be composed parallel. Modes of the composed automaton $\mathcal{A}_0||\mathcal{A}_1$ are the products of modes of each component automaton. When the control stays in a mode $\langle s, s'\rangle$, the flow of variables' values must respect differential equations and invariants with the state s of $\mathcal{A}_0$ and with the state s' of $\mathcal{A}_1$. Hybrid automata may also be composed hierarchically. In a hierarchical hybrid automaton, a mode may be another hierarchical hybrid automaton.

Hybrid automata have been widely used for modeling and simulating embedded systems. Figure 2 shows a hierarchical hybrid automaton modeling a SONY AIBO robot tracking a moving ball. θ is the angle of the ball, x is the angle of the robot's head, and β is the visibility of the ball. When the ball is visible ($\beta > 10$), the robot attempts to chase the ball, as modeled by a differential equation $\dot{x} = k \cdot (\theta - x)$. If the ball is invisible ($\beta \leq 10$), the control switches to the left-top mode, which in turn has two sub-modes. In the left sub-mode, the robot moves its head toward its right at the speed of 10 deg./sec. The right sub-mode is symmetric to the left sub-mode. The robot simply swings its head if it cannot see a ball.

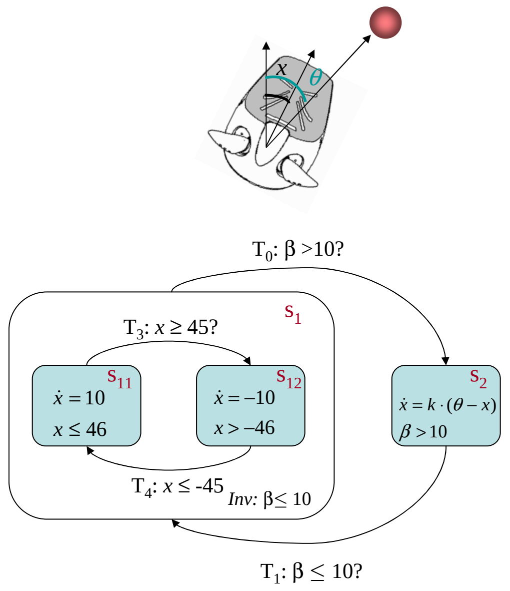


Figure 2: Hierarchical hybrid automaton modeling a SONY AIBO robot tracking an object.

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

where q is a constant, e is a primitive event, $\diamond \in \{+, -, *, /\}$ and $\square \in \{>, <, =\}$ are arithmetical and relational operators, respectively [7].

Table 1: The syntax of MEDL

Modeling Language CHARON**.** CHARON [2] is a modeling language based on hybrid automata. CHARON supports both behavioral and structural hierarchy by allowing the concurrent and hierarchical composition of hybrid automata. CHARON design environment 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 modeling environment in which we implement our framework and demonstrate its benefits.

3. MODEL-BASED RUNTIME VERIFICATION

To detect reconfiguration condition at runtime, we use model-based runtime verification technique introduced in [19]. The technique uses the event-based temporal logics MEDL and mPEDL to encode system requirement. It extends a hybrid automaton model to a self-monitoring model based on MEDL and mPEDL specifications. In our framework, events generated by a self-monitoring model are used to activate reconfiguration mechanism.

3.1 The Temporal Logic MEDL

The syntax of MEDL is given in Table 1, where E and C denote events and conditions. The building blocks of MEDL are *events* - things that occur at some time instance and *conditions* - facts that last for certain duration. MEDL is interpreted over a sequence of primitive events. Primitive events represent the status changes of a monitored system. Their precise definitions are given in mPEDL, which will be introduced shortly after. We use the following notations: we denote $\rho\{\mathcal{I}\}$ and $\rho(t)$ for part of a hybrid automaton trace ρ during an interval $\mathcal{I}$ and at a time t respectively. $\rho\{\mathcal{I}\} \models C$ states that a condition C holds true on a hybrid automaton trace ρ during some interval $\mathcal{I}$, and $\rho(t) \models E$ indicates that an event E occurs at time t on ρ.

MEDL initially was introduced to specify requirements for Java programs [7], and hence originally MEDL was interpreted over discrete execution steps. In [17] we extend its semantics to the continuous time domain. For instance, the semantics of a condition E *when* C is defined as,

$$\rho(t) \models E \;\mathsf{when}\; C \text{ iff } \rho(t) \models E \text{ and } \lim_{\delta \to 0^+} \rho\{[t-\delta, t+\delta)\} \models C$$

That is, the event E when C occurs if E occurs and at the same time C holds. $\mathsf{start}(C)$ and $\mathsf{end}(C)$ defines an event as the start and the end of the condition C. MEDL also uses expressions to help track the history of an execution. For instance, $\mathsf{time}(E)$ specifies the time when an event E occurs. For the complete semantics of MEDL in the continuous time domain, readers may refer to [17].

```
ReqSpec
  buildin event clock;
  import event appear, disappear, track, lose;
  import event approach, leave, almostLose;
  export event loseBall, relax, catchUp;
  condition visible= [appear, disappear);
  event loseTrack = lose when visible;
  alarm loseBall =start (time(loseTrack)-time(appear)>50);
  /* added for self adaptation */
  event catchUp = almostLose when visible;
  condition close = [approach, leave) when visible;
  event relax= start(time(clock) when close \
                 -time(start(close))>3);
End
```

Figure 3: The MEDL script $\mathbb{M}_a$ for the AIBO example

MEDL as a script language has a richer syntax. In a MEDL script, primitive events are referred as *imported events*. There is also a special type of events called *alarms*. An alarm indicates the violation of safety properties. Therefore, a hybrid automaton trace, marked by a sequence of primitive events, is considered to satisfy a MEDL script if no alarm is raised. A MEDL script also defines *exported events* which are visible to the outside. These events may be used to trigger reconfiguration.

Figure 3 gives a MEDL script which specifies the requirements for the AIBO example in Figure 2. The script requires the robot to track a ball when it is visible. An alarm *loseBall* is raised if the robot fails its mission.

3.2 Model-Based Primitive Event Definition Language mPEDL

In our framework primitive events are defined in the Model-Based Primitive Event Definition Language mPEDL, which is extended from PEDL [9] in context of hybrid automata. A primitive event can either be a transition or a change of some predicate's value. Figure 4 shows an mPEDL script

```
MonScr
  export event appear, disappear, lose;
  export event approach, leave, almostLose;
  monobj int dog.beta,dog.theta,dog.x;
  /* Event definition */
  event  appear= start (dog.beta>10);
  event  disappear= end (dog.beta>10);
  event  lose = start(abs(dog.theta-dog.x)>20);
  event  approach = start(abs(dog.theta-dog.x)<5);
  event  leave = end(abs(dog.theta-dog.x)<5);
  event  almostLose = start(abs(dog.theta-dog.x)>15);
End
```

Figure 4: The mPEDL script $\mathbb{P}_a$ for the AIBO example

for the AIBO example. It has two sections.

- *Monitored objects* declare shared variables and transitions being monitored.

- Event section defines primitive events as transitions or changes of predicates' values. For instance, the events *appear* and *disappear* indicate the start and the end of the predicate *dog.beta* > 10.

mPEDL bridges monitored systems and MEDL. It defines interesting changes of system status as primitive events. The mPEDL script in Figure 4 defines six primitive events. These primitive events are taken by the MEDL script in Figure 3 to compute conditions and *non-primitive* events which identifies more complicate temporal patterns of a trace.

3.3 Build Self-Monitoring Model

In model-based runtime verification framework [19, 17] we automate the process of building a self-monitoring model by first instrumenting the original system model and then extending it with a model-based monitor synthesized from a MEDL script. The embedded model-based monitor emits events via share variables, which can be used to activate reconfiguration mechanism in Section 4.

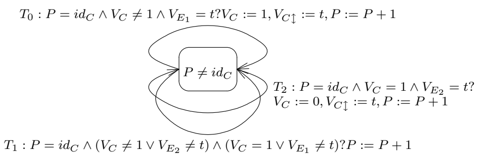
3.3.1 Model Instrumentation

To facilitate runtime verification, a system model must be instrumented to emit events defined in an mPEDL script. Primitive events are emitted via shared variables: for each primitive event we introduce a variable that records last occurrence of the event. We need to consider two cases when instrumenting a model,

1. To emit primitive events defined as changes of a predicate's value, the system model is composed with an observer. An observer is the parallel composition of a collection of two-mode automata. Each mode stands for either true or false of a predicate. The transitions between these modes emit primitive events by updating the related event variables. Figure 6 shows the model in Figure 2 after instrumentation.

2. To emit events defined on transitions, the assignment actions of these transitions are extended to update the related event variables.

3.3.2 Synthesize Model-Based Monitors



$T_0 : P = id_C \wedge V_C \neq 1 \wedge V_{E_1} = t ? V_C := 1, V_{C\updownarrow} := t, P := P + 1$
$P \neq id_C$
$T_2 : P = id_C \wedge V_C = 1 \wedge V_{E_2} = t ?$
$V_C := 0, V_{C\updownarrow} := t, P := P + 1$
$T_1 : P = id_C \wedge (V_C \neq 1 \vee V_{E_2} \neq t) \wedge (V_C = 1 \vee V_{E_1} \neq t) ? P := P + 1$

Figure 5: Translation rule for $C = [E_1, E_2)$

Synthesizing a model-based monitor from a MEDL script is carried out in a compositional manner. Each term (condition, expression, or non-primitive event) is translated to a *term* automaton. Figure 5 shows the translation rule for the case $C = [E_1, E_2)$. Each event E is related to a variable V_E that records the last occurrence of E. Each expression Q is also related to a variable V_Q that stores its current value. Each condition C is associated with two variables: V_C records C's value and $V_{C\updownarrow}$ records the last time C changes its value. A term automaton updates the related term variables. The monitor also implements a token-passing process: the automata for a term T is activated only after automata for terms that T depends on have been activated. A model-based monitor is the parallel composition of term automata and an *engine* automaton. The engine automaton checks incoming primitive event and passes the *token* to the first term automaton on the ring to start event processing.

An instrumented model is composed with a model-based monitor to form a self-monitoring model. During simulation, a self-monitoring model performs runtime monitoring in addition to its original behavior. The occurrences of an event E are indicated by changes of the event variable V_E's value. Theorem 1.a shows that the size of additional monitoring mechanism is linear to the size of MEDL and mPEDL requirements, and Theorem 1.b states that the computation complexity of a synthesized monitoring automaton is linear to the size of the MEDL script.

THEOREM 1. *Let M be the monitoring automaton synthesized from the MEDL script $\mathbb{M}$, and $\mathcal{A}'$ be an instrumented version of a hybrid automaton $\mathcal{A}$ for the mPEDL script $\mathbb{P}$,*

a. $|M||\mathcal{A}'| = O(|\mathbb{M}| + |\mathbb{P}|)$

b. The update of variables in M can be done in $O(|\mathbb{M}|)$.

Figure 7 shows a simulation trace of a self-monitoring model for the AIBO example with the MEDL requirement $\mathbb{M}_a$ and the mPEDL script $\mathbb{P}_a$. We use a model-based tester [18] to model a moving ball whose position is fed to the self-monitoring model via input variables. Figure 7.(a) shows the ball's movement. It swings slowly before the robot during initial 10 seconds, then moves faster in next 10 second, and slows down again. Figure 7.(b) shows the relative angle between the ball and the robot's head. The simulation reveals a failure: the change of the event variable $V_{loseBall}$'s value at the 10th second indicates that the alarm *loseBall* occurs. It is because the robot doesn't move fast enough when the ball starts to accelerate. Note that the latency factor k in the robot model decides how aggressively fast the robot tracks the ball. A proper fix to the problem requires to let the robot on-the-fly reconfigure k at runtime.

4. REDL: A RECONFIGURATION DEFINITION LANGUAGE

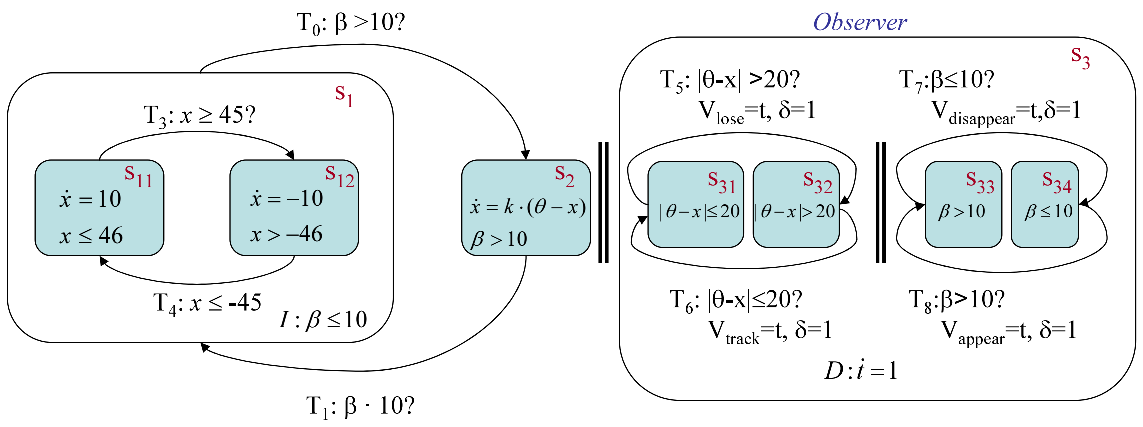


Figure 6: Instrument the AIBO robot model.

To formally specify *reconfiguration* - such as when it shall take place and what it shall achieve - we introduce the Reconfiguration Definition Language REDL. In conjunct with MEDL and mPEDL, REDL specifies reconfiguration requirements for embedded systems. Having a formal reconfiguration specification for embedded systems has several importances. A formal reconfiguration specification adds one more dimension to a formal system specification. It defines what needs to be done, shall a system be failing, to save the system and improve its performance. A formal reconfiguration specification also provides the basis for mechanically extending a system design with reconfiguration capability and the possibility of applying formal verification techniques to reconfigurable embedded systems.

$$
\begin{array}{lcl}
\langle R \rangle & ::= & \langle T \rangle ; \langle R \rangle \mid \langle R \rangle \\
\langle T \rangle & ::= & \textbf{trigger } e : \langle B \rangle \\
\langle B \rangle & ::= & \langle A \rangle ; \langle B \rangle \mid \langle A \rangle \\
\langle A \rangle & ::= & \textbf{if } \langle C \rangle \textbf{ then } \langle D \rangle \\
\langle C \rangle & ::= & \langle C \rangle \&\& \langle C \rangle \mid \langle C \rangle || \langle C \rangle \mid \langle S \rangle \mid p \\
\langle S \rangle & ::= & \{ \langle S_l \rangle \} \mid \{\} \\
\langle S_l \rangle & ::= & s, \langle S_l \rangle \mid s \\
\langle D \rangle & ::= & \langle F \rangle ; \langle D \rangle \mid \langle F \rangle \\
\langle F \rangle & ::= & s \ \langle P \rangle \\
\langle P \rangle & ::= & \langle C_t \rangle \langle C_p \rangle \\
\langle C_t \rangle & ::= & \textbf{within } t \mid \epsilon \\
\langle C_p \rangle & ::= & \textbf{under } p \mid \epsilon
\end{array}
$$

where s is a state, and p is a predicate over a set of variable V, and e is an event.

Table 2: The syntax of REDL

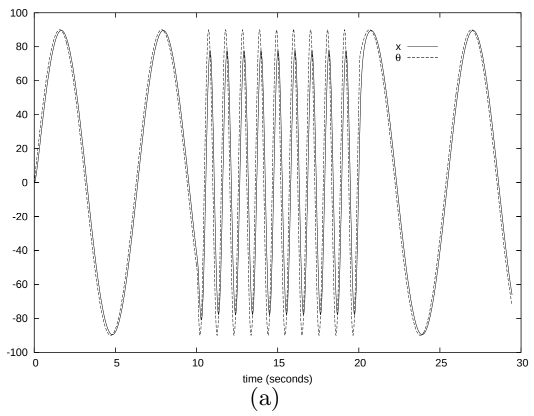
4.1 Syntax

A REDL script defines the requirement for reconfiguration. Table 2 gives the syntax of REDL. A REDL script is a list of reconfiguration scenarios. A scenario $\langle T \rangle$ specifies the event e triggering a reconfiguration and a list of actions $\langle B \rangle$. Each action $\langle A \rangle$ defines its precondition $\langle C \rangle$ and postcondition $\langle D \rangle$. A proposition in the precondition $\langle C \rangle$ can be either a set of states $\langle S \rangle$, which holds when a state in $\langle S \rangle$ becomes active, or a predicate p over a set of variables V defined in a targeted system model. Besides the triggering event, the precondition must hold before a reconfiguration action can take place. The postcondition $\langle D \rangle$ specifies which state the reconfiguration ends up with, an optional worse-case execution time, and an optional predicate which must be satisfied after the reconfiguration.

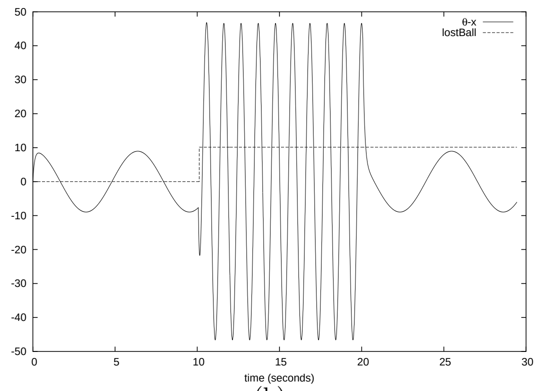
Figure 8 shows the reconfiguration script for the AIBO robot model. k in the model is a reconfigurable parameter controlling how aggressively the robot tracks a ball. A higher k means a lower risk of losing the ball, but also costs more energy to operate the robot. A higher k is also a cause of “jittering”, in which the robot constantly overshoots the target ball. The purpose of reconfiguration is to adjust k based on runtime situation so the robot can maintain the optimal operation. We start with a formal definition of suboptimal situations which reconfiguration shall respond to. In Figure 3 we define the events *relax* and *catchUp* in the MEDL script, and supporting primitive events *approach*, *leave*, and *almostLose* in the mPEDL script in Figure 4. Casually speaking, the event *catchUp* indicates that the robot lags significantly behind the ball and hence a more aggressive tracking is necessary to avoid losing the ball. The event *relax* indicates that the robot follows a ball too close and hence it can slow down to conserve energy and avoid jittering. Note that neither *catchUp* nor *relax* is a simple measurement of angle between the ball and the robot. They use temporal operators supplied by MEDL to identify more subtle temporal patterns. For instance, the event *catchUp* also tests whether a ball is visible, and the event *relax* waits three seconds after the robot enters within the 5 degree range of the ball. This three-second delay is crucial to the system's stability because a premature *relax* event makes the robot slow down when the ball already starts to pick up its speed, which makes the robot bounce between low and high speeds.

4.2 A Formal Semantics for Reconfiguration

A REDL script specifies the requirement for reconfiguration. Reconfiguration alters the behavior and structure of an embedded system. Semantically reconfiguration changes the courses of original system traces. Structurally reconfiguration extends the original system with a collection of new transitions, new modes, and other changes to facilitate runtime self adaption. A trace of a self-adaptive system has two different kinds of subsequences: subsequences exhibiting the original system behavior when reconfiguration is not activated; and subsequences in which a reconfiguration occurs.



(a)



(b)

Figure 7: Simulate the self-monitoring model with $k = 10$

The latter sequence must satisfy pre- and post- conditions of reconfiguration requirement. This causal definition is formalized in Definitions 1 and 2.

Definition 1. *Let $\mathbb{R}$ be a REDL script and $\mathcal{B}$ be a self-adaptive extension of a hybrid automaton, a trace ρ of $\mathcal{B}$ satisfies $\mathbb{R}$ if ρ can be partitioned to a set of subsequences $Q \cup O$ as follows,*

1. *For each subsequence $\rho\{[t_i, t_{i+1})\} \in Q$ with $h = \langle s, \mathbf{v}\rangle$ as its start state and $h' = \langle s', \mathbf{v}'\rangle$ as its final state, there is a reconfiguration action in $\mathbb{R}$,*

   trigger e: if S and P then enter s' under P' within t

   such that $\rho(t_i) \models e$, $s \in S$, $P(\mathbf{v}) = \mathbf{true}$, $P'(\mathbf{v}') = \mathbf{true}$, and $t_{i+1} - t_i \leq t$.

2. *For each segment $\rho\{[t_i, t_{i+1})\} \in O$ with s as its start state and $\mathbf{v_t}$ as its initial valuation of the variables V,*

   (a) There doesn't exist a time $t \in [t_i, t_{i+1})$ and an reconfiguration action in $\mathbb{R}$

   trigger e: if S and P then enter s' under P' within t

   such that $P(\mathbf{v_t}) = \mathbf{true}$, $\rho(t) \models e$, and $s \in S$.

   (b) $\rho\{[t_i, t_{i+1})\}$ is a prefix of a trace of the automaton $\mathcal{A}_t = \{S, V, T, G, W, D, I, \langle s, \mathbf{v_t}\rangle\}$. $\mathcal{A}_t$ is a variance of A with a new starting state $\langle s, \mathbf{v_t}\rangle$.

```
ReqExp
 import event catchUp, relax;
 Trigger catchUp:
        if {s2} and k<50 then
            enter s2 under k=50 within 0;
 Trigger relax:
        if {s2} and k>10 then
            enter s2 under k=10 within 0;
End
```

Figure 8: The reconfiguration script $\mathbb{R}_a$ for the AIBO model

Definition 1 partitions a trace of a self-adaptive model to two different types of subsequences: Q, the subsequences where reconfiguration actions take place; and O, the subsequences which preserve the original system behavior. Definition 1.1 states that a subsequence $\rho\{[t_i, t_{i+1})\}$ in Q completes a reconfiguration action: a triggering event occurs at t_i and the valuation at t_i satisfies the precondition. All the postcondition and the time constraints shall be satisfied after the reconfiguration. Definition 1.2 requires that a reconfigured system shall perform exactly same as the original system if no reconfiguration action is triggered: 1.2.(a) states that a subsequence $\rho\{[t_i, t_{i+1})\} \in O$ doesn't satisfy the precondition of any reconfiguration actions, and hence it does not engage in any reconfiguration activities; and 1.2.(b) requires that such subsequence shall respect the semantics of the original system, i.e., it behaves as if the original system starts with the same initial state as $\rho\{[t_i, t_{i+1})\}$.

Definition 2. *Let $\mathcal{B}$ be a self-adaptive extension of $\mathcal{A}$, $\mathcal{B}$ satisfies a REDL script $\mathbb{R}$ if every trace of $\mathcal{B}$ satisfies $\mathbb{R}$.*

5. BUILD SELF-ADAPTIVE MODELS

A complete set of reconfiguration requirements in our framework is a $\langle \mathbb{R}, \mathbb{M}, \mathbb{P}\rangle$, which consists of a REDL script $\mathbb{R}$, an MEDL script $\mathbb{M}$ defining triggering events, and an mPEDL script $\mathbb{P}$ specifying supporting primitive events. A REDL script $\mathbb{R}$ only specifies the requirements for reconfiguration actions, but doesn't mandate the actual implementation. A designer can choose his own implementation as long as it satisfies Definitions 1 and 2. To take advantage of model-based runtime verification techniques we introduced, we extend the original system model to a self-monitoring model based on $\mathbb{M}$ and $\mathbb{P}$. We then use the following procedure to extend it into a self-adaptive model: for each reconfiguration action,

trigger e: if S and p then enter s' under p' within t

and each mode $s \in S$,

(1) Add a transition t leaving s with $G(\mathsf{t}) = (V_e > 0 \wedge p)$ as its guard and $W(\mathsf{t}) = \{V_e := 0\}$ as its assignment action.

(2) Add $\neg p \vee V_e \leq 0$ to s's invariant set $I(s)$.

(3) Add a transition t' entering s with $W(\mathsf{t}') = p'$ as its assignment action.

(4) Add customized reconfiguration steps connecting t to t', and verify the time constraints.

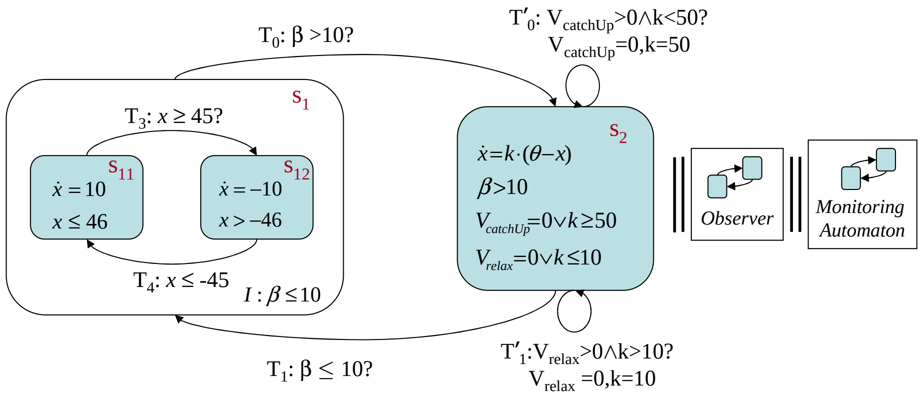


Figure 9: The self-adaptive model for SONY AIBO example

Note that $V_e > 0$ indicates that the event e occurs. Step (1) adds a transition t which checks if a triggering event presents and the precondition of a reconfiguration action is satisfied. Step (2) will make the transition t *urgent*: the invariant added to s is the negation of t's guard. This forces the model to take the transition t if t is enabled. Step (3) enforces the post-condition after the reconfiguration. Finally, Step (4) allows a user to add customized reconfiguration steps. The customized steps should be checked against the worst-case execution time constraint. Verifying time constraints is essentially a reachability problem consisting of only those new transitions and modes from s to s'. Such problem may be solved using, for example, model-checking tools [4, 8] for timed systems.

Figure 9 gives an example of a self-adaptive model, which is extended from the model in Figure 2 under the requirement $\langle \mathbb{R}_a, \mathbb{P}_a, \mathbb{M}_a \rangle$. It satisfies the REDL script in Figure 8. At the step (1) we add two new transitions T_0' and T_1' to handle events *relax* and *catchUp*, respectively. For instance, the guard of T_0' checks the event *catchUp* and precondition $k < 50$. The guards of both T_0' and T_1' are negated and added to the invariant of S_2 to make T_0' and T_1' urgent. At the step (3) we add assignments to T_0' and T_1' to force the post-condition. Since either T_0' or T_1' is taken instantly, the worst-case execution time constraint is clearly satisfied.

One advantage of our proposed model-based self-adaptive framework is its scalability. First, the size of additional transitions and modes introduced by the above procedure, except optional customized reconfiguration steps, is linear to the size of the REDL script; second, the framework also benefits from the scalability of underlying model-based runtime verification technique. Theorem 1.a shows that the size of model-based runtime verification mechanism is linear to its requirements. Additional mechanism to support self adaption is linear to the size of the reconfiguration requirement set $\langle \mathbb{R}, \mathbb{M}, \mathbb{P} \rangle$. By Theorem 1.b, the computational cost of runtime verification is linear to the size of requirement script, and the delay introduced by reconfiguration is bounded by worst-case execution time constraints in $\mathbb{P}$.

6. FROM MODELS TO SELF-ADAPTIVE EMBEDDED PROGRAMS

We established a model-based framework for self-adaptive embedded system design by first building self-monitoring models and then extending it with reconfiguration mechanism. Models at each construction step are written in the exactly same modeling language as the original system model, and hence existing model-based design tools can be used to simulate these models and produce embedded codes with some additional benefits: a self-monitoring model also performs runtime verification[17] during simulation; and an self-adaptive model can be used to generate self-adaptive embedded program using an existing model-based code generator. We follow our case study on SONY AIBO robot to discuss these benefits.

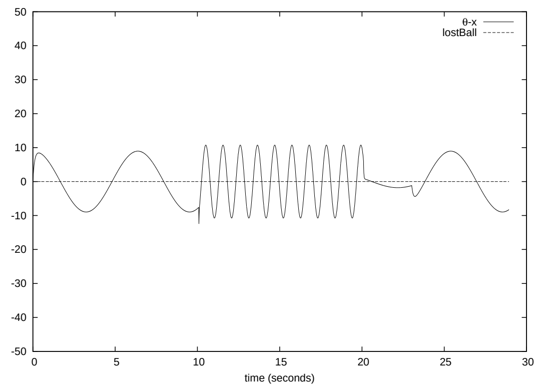


Figure 10: Simulate the self-adaptive model in Figure 9

About the case study. The target hardware platform in our case study is the SONY AIBO robotic dog [15]. AIBO 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 192 MHz, and equipped with 32 MB internal memory. The operating system is SONY's propriety object-oriented real-time operating system known as Aperios.

In Section 3.3.2 the simulation of a self-monitoring model already reveals a problem with the fixed latency factor k. As a solution, we develop a self-adaptive model in Figure 9. Figure 10 shows the simulation trace for the self-adaptive model. The self-adaptive system model evaluates its own execution and adapt k to optimize performance. The self-adaptive system effectively avoids raise of the alarm *loseBall*.

	Original	Self-monitoring	Self-adaptive
CHARON	221	541	552
Simulink	1613	3017	3361
C++ code	551	836	860
Binary	466,555	469,461	470,063

Table 3: The size of the self-adaptive embedded program for the AIBO example

In [18] we model AIBO's control program in CHARON and extend it with runtime verification capability. We also use a model-based generator developed in [3] to generate C++ code. We take a slight different route in this case study, we extend the control system model to a self-adaptive model also in CHARON; we then translate CHARON models to Simulink/Stateflow [14] and use the real-time workshop to generate embedded codes, since the real-time workshop has a better hardware target support. Table 3 shows the code size for the different extensions of original system model. All the sizes are measured in lines, except for binary code, which is in bytes. The self-monitoring model is generated automatically using the M^{2}IST toolkit [11].

7. CONCLUSIONS

A self-adaptive system evaluates its own execution and reconfigures itself to avoid failure. The concept of self adaption is especially important to embedded system applications that demand dependability and survivability. We proposed a model-based framework for building self-adaptive embedded systems. Our approach has several benefits: first, monitoring and reconfiguration mechanisms are built in as part of design. We work with models and hence our approach can be readily incorporated into existing design environment. For instance, an existing model-based code generator can be used to generate embedded codes directly from a self-adaptive model; second, requirements are formally specified. We introduce a formal language REDL to specify reconfiguration plans rigidly and give a formal semantics for reconfiguration in context of hybrid automaton-based modeling; finally, the liberal use of formal techniques at different stages of design and development allows us to rigidly specify system requirement and reconfiguration plans, and helps automate the process of building self-adaptive embedded programs.

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. *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*, 91:11– 28, 2003.

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

[4] T. A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[5] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Engineering*, 12(9), 1986.

[6] Gabor Karsai, Ákos Lédeczi, Janos Sztipanovits, Gábor Péceli, Gyula Simon, and Tamás Kovácsházy. An approach to self-adaptive software based on supervisory control. In *IWSAS'01*, pages 24–38, 2001.

[7] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: a run-time assurance tool for Java. In *1st International Workshop on Run-time Verification*, Electronic Notes in Theoretical Computer Science 55 No. 2, 2001.

[8] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshel. *Software Tools for Technology Transfer*, 1:134–152, 1997.

[9] 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.

[10] 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.

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

[12] K. Narendra and A. Annaswamy. *Stable Adaptive Systems.* Prentice-Hall, 1988.

[13] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[14] Stateflow Simulink and Real time workshop. The MathWorks, Inc. In *http://www.mathworks.com.*

[15] SONY AIBO Robot. Sony corporation. http://www.aibo.com.

[16] E. Strunk and J. Knight. Dependability through assured reconfiguration in embedded system software. *IEEE Transactions on Dependable and Secure Computing*, To appear, 2006.

[17] L. Tan. Model-based self-monitoring embedded programs with temporal logic specification. In *Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, 2005.

[18] L. Tan, J. Kim, and I. Lee. Testing and monitoring model-based generated program. In *Proceeding of Runtime Verification Workshop*, volume 89 of *Electronic Notes in Theoretic Computer Science*. Elsevier Science, 2003.

[19] L. Tan, J. Kim, I. Lee, and O. Sokolsky. Model-based testing and monitoring for hybrid embedded systems. In *Proceedings of IEEE International Conference on Information Reuse and Integration*, 2004.