Verifying Autonomic Fault Mitigation Strategies in Large Scale Real-Time Systems

Abhishek Dubey   Steve Nordstrom   Turker Keskinpala   Sandeep Neema   Ted Bapty

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37203, USA
{dabhishe, steve-o, tkeskinpala, sandeep, bapty}@isis.vanderbilt.edu

Abstract

In large scale real-time systems many problems associated with self-management are exacerbated by the addition of time deadlines. In these systems any autonomic behavior must not only be functionally correct but they must also not violate properties of liveness, safety and bounded time responsiveness. In this paper we present and analyze a real-time Reflex Engine for providing fault mitigation capability to large scale real time systems. We also present a semantic domain for analyzing and verifying the properties of such systems along with the framework of real-time reflex engines.

1. Introduction

Advancements in computing technology coupled with increased integration between information and the physical world has enabled the use of computing devices for controlling modern systems. At the same time, the proliferation in communication bandwidth and ability to construct complex systems from simpler components is abetting the increase in capability and complexity of these computer based systems (CBS). The impact of this technological progress can be felt from small systems such as pacemakers to large systems used in military, high performance computing and avionics.

A side-effect of an increase in system complexity is an increase in chances of system failure. On the other hand, an interesting contradiction is the rise in demand of reliability of complex CBS used in critical areas such as nuclear plants. The demand of system reliability even in the wake of increasing complexity has ensured that fault-tolerance becomes a sine qua non of the design cycle.

Fault tolerance can be implemented by employing humans to make fault mitigation decisions. However, usually the involvement of humans leads to latency in a fault mitigation decision which (a) might not be acceptable in critical systems, and (b) usually results in pecuniary losses. Moreover, in certain situations human decision making might be unavailable due to a lack of plant accessibility such as in high energy physics applications and space systems.

This drives the need for an alternate paradigm for autonomic computing systems that are self aware, self configuring, self healing, and self protecting [1, 2, 3]. Such systems employ fault managers for making the necessary changes in the wake of failures. A motivation for this alternate paradigm comes from the biological world and the ways in which biological systems handle similar challenges of fault tolerance and self management.

Challenges associated with self-management are exacerbated by the addition of time deadlines. In real-time systems, it is not enough to compute correct results, but to compute correct results within the given time deadline [4]. In such systems, a correct but “late” fault mitigation action is an incorrect system behavior. Even in systems which are not real-time by themselves it may be required that the system is repaired within strict time bounds. The autonomic behaviors for such systems must not only be functionally correct but they must also not violate the following properties:

1. *Liveness:* Assuming that the original system was deadlock free, the addition of autonomic behaviors will not lead to a deadlock in the system.
2. *Safety:* The system will meet all of its deadlines and never operate in unsafe modes.
3. *Bounded time responsiveness:* In the case of faults, the designed fault mitigation action will be executed within a specified time bound.

Our previous work in large scale autonomic computing yielded a Reflex and Healing (RH) framework presented in

[5, 6, 7]. This framework employs a hierarchical network of fault management entities, called reflex engines, whose sole purpose is to implement fast reflexive actions when a fault is detected.

An initial investigation toward applying real-time analysis techniques to this framework showed the modeling formalism for RH systems to be inadequate for timed analysis. In this paper, we augment our existing RH framework based on our earlier work to provide timed analysis capability, and present an approach to transform the augmented RH system into a semantically equivalent network of timed automaton models [8, 9]. Once transformed, the timed automaton models can be checked for the properties that ought to be satisfied by autonomic computing systems by using model checking tools such as UPPAAL [10], and KRONOS [11].

The outline for the remainder of this paper is as follows: In the next section we review the background of timed automaton which is essential to our solution approach. We then describe the definition of a timed RH framework. We will then present the transformation of the RH framework into network of timed automatons. The paper will conclude with a case study in which we will check a simple application system with two reflex strategies.

2. Background

For the benefit of our readers, we provide in this section a brief overview of the RH framework as well as a review of the definitions and semantics of a timed automaton and networks of timed automatons. However, readers who are familiar with these concepts may wish to proceed to Section 3.

2.1. Reflex and Healing Framework

A reflex and healing architecture is a tree-like hierarchical deployment of fault managers, also known as reflex engines. The justification for a hierarchical deployment vs. a single layered approach is beyond the scope of this review. However, readers are referred to [5, 6] for a more comprehensive study of the subject.

Fig. 1 shows an example deployment of this framework. Each reflex engine has a management domain which constraints its zone of influence in order to manage the difficulty of implementing a scalable, high performance centralized fault diagnosis and mitigation for a large system.

User applications, such as those shown in the Fig. 1, are managed by the lowest level reflex engines, termed local managers. A number of local managers themselves are grouped together under the zone of influence of a higher level manager termed as regional managers. At the top lies the global manager which supervises the health of regional managers. There can be other mid-tier managers over the

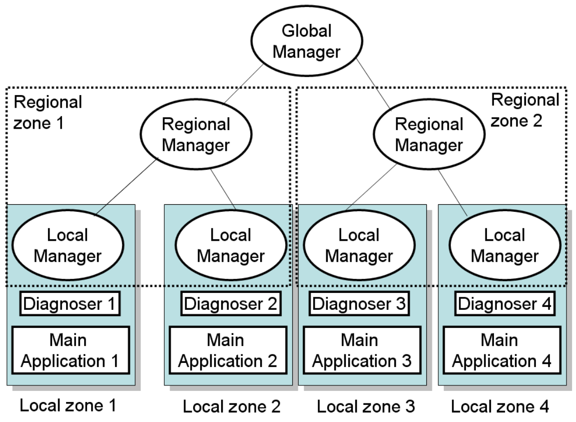

Figure 1. Hierarchical fault management entities in Reflex and Healing architecture.

regional managers. The communication between managers is restricted to communication between parent and children, i.e. direct communication between two managers at the same level is forbidden. This is necessary to prevent leakage of faults between peer zones.

2.2. Timed Automaton

Classically, for systems with continuous timed variables, the timed automaton (TA) model [8, 9] is used for proving the correctness of system designs. This approach has also been applied to solve scheduling problems by modeling real-time tasks and scheduling algorithms as variants of timed automaton and performing reachability analysis on the model [12, 13]. In this paper we have followed a similar approach to analyze an RH framework.

A timed automaton consists of a finite set of states called *locations* and a finite set of real-valued clocks. It is assumed that time passes at a uniform rate for each clock in the automaton. Transitions between locations are triggered by the satisfaction of associated clock constraints known as *guards*. During a transition, a clock is allowed to be reset to zero value. These transitions are assumed to be instantaneous. At any time the value of each clock is equal to the time passed since the last reset of that clock. In order to make the timed automaton urgent, locations are also associated with clock constraints called *invariants* which must be satisfied for a timed automaton to remain inside a location. If there is no enabled transition out of a location whose invariant has been violated, the timed automaton is said to be *blocked*. Formally, a timed automaton can be defined as follows:

Definition 1 (Timed Automaton) *A timed automaton is a 6-tuple TA=$< \Sigma, S, S_0, X, I, T >$ such that*

- *Σ is a finite set of alphabets which TA can accept.*
- *S is a finite set of locations.*
- *$S_0 \subseteq S$ is a set of initial locations.*
- *X is a finite set of clocks.*
- *$I : S \rightarrow \mathscr{C}(X)$ is a mapping called location invariant. $\mathscr{C}(X)$ is the set of clock constraints over X defined in BNF grammar by $\alpha ::= x \prec c | \neg\alpha | \alpha \wedge \alpha$, where $x \in X$ is a clock, $\alpha \in \mathscr{C}(X)$, $\prec \in \{<, \leq\}$, and c is a rational number.*
- *$T \subseteq S \times \Sigma \times \mathscr{C}(X) \times 2^X \times S$ is a set of transitions. The 5-tuple $< s, \sigma, \psi, \lambda, s' >$ corresponds to a transition from location s to s' via an alphabet σ, a clock constraint ψ specifies when the transition will be enabled and $\lambda \subseteq X$ is the set of clocks whose value will be reset to 0.*

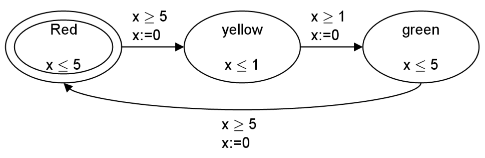

Figure 2. A Timed automaton model of a behavior of traffic light.

It is customary to draw a timed automaton model as a directed graph with its nodes, drawn as circles or ellipses, to represent the locations and the edges to represent the transitions. Initial locations are marked using concentric ellipses or circles. Fig. 2 shows a timed automaton model of a traffic light. This automaton has three locations and one clock variable. The model periodically circulates between red, yellow, and green location at a time gap of 5 units, 1 unit, and 5 units respectively.

The semantics of timed automaton models are described as an infinite state transition graph $A =< \Sigma, Q, Q_0, R >$. Each state in Q is a pair (s, v), where $s \in S$ and $v : X \rightarrow \mathbb{R}^+$ is clock value map, assigning each clock a positive real value. It is assumed that at any time all clocks increase with a uniform unit rate i.e. $\forall x \in X(\dot{x} = 1)$ is true. The initial state of A, Q_0 is given by $\{(q, v) | q \in S_0 \wedge \forall x \in X(v(x) = 0)\}$. Before defining the transition relations we must give some notations. For any $d \in \mathbb{R}^+$, let us define $v + d$ a clock assignment map which increases the value of each clock $x \in X$ to $v(x) + d$. For $\lambda \subseteq X$ introduce $v[\lambda := 0]$ to be the clock assignment that maps each clock $y \in \lambda$ to 0, but keep the value of all clocks $x \in X - \lambda$ same.

The transition relation R is composed of two types of transitions:

- *Delay Transitions* refer to passage of time while staying in the same location. They are written as $(s, v) \xrightarrow{d} (s, v + d)$. The necessary condition is $v \in I(s)$ and $v + d \in I(s)$
- *Action Transitions* refer to occurrence of a transition from the set T. Therefore for any transition $< s, \sigma, \psi, \lambda, s' >$, we can write $(s, v) \xrightarrow{\sigma} (s', v[\lambda := 0])$, given that $v[\lambda := 0] \in I(s')$ and $v \in \psi$.

Usually, a system is composed of several sub-systems, each of which can be modeled as a timed automaton. Therefore, for modeling of the complete system we will have to consider the parallel composition of a network of timed automatons [10, 14, 11].

2.3. Networks of Timed Automatons

We consider a network of timed automatons as the synchronous parallel composition of $TA_1||TA_2||.....||TA_n$ of n timed automatons. Each timed automaton can synchronize with any other timed automaton by hand shake synchronization using input events and output actions. For this purpose we assume the alphabet set Σ to consist of symbols for input events denoted by $\sigma?$ and output actions $\sigma!$ and internal events τ. Apart from these synchronizing events, we also use the concept of a shared variable of integer type and arrays of integers. This concept is motivated from the concepts in timed automaton based tools such as UPPAAL and KRONOS. Effectively, guard conditions can also be set on the values of shared variables. During a reset, the values of these shared variables can be changed to some integral quantity.

For example, Fig. 3 shows model of a scheduler executing in parallel to a task. The scheduler enables the task 6 time units after it finishes. The two automatons synchronize using start and finish event/action pair. For sake of simplicity we assume that there are no shared clocks in the system.

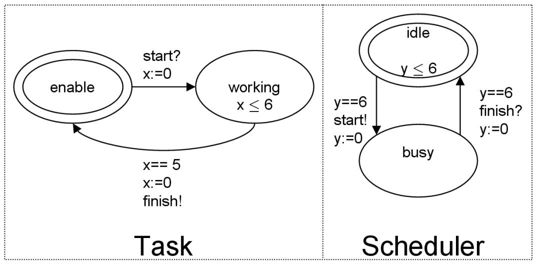

Figure 3. A network of two timed automatons.

The semantics of network timed automatons are also given in terms of transition graphs. A state of a network is defined as a pair $(\vec{s}, v)$, where $\vec{s}$ denotes a vector of all the current locations of the network, one for each timed automaton, and v is the clock assignment map for all the clocks of the network. The rules for delay transitions are same as those for a single timed automaton. However, the action transitions are composed of internal actions and synchronizing actions. These actions can be described as follows:

1. *Delay transitions* are described as $(\vec{s}, v) \xrightarrow{d} (\vec{s}, v + d)$, if $v, v + d \in \cap_{i=1}^{i=n} I(s_i)$, where $\forall i \; s_i \in \vec{s}$ and n is the total number of timed automaton in the network.

2. *Internal action transitions* are described as $(\vec{s}, v) \xrightarrow{\tau} (\vec{s}[s'_i|s_i], v')$ if an action transition $s_i \xrightarrow{\tau} s'_i$ is possible for i^{th} timed automaton in the network . All the timed automatons in the network are allowed to have internal action transition at the same time.

3. *External action transitions* are associated with change in state of transitions of at least two timed automaton of the network in parallel. Suppose i^{th} timed automaton produces an output action $a!$ which is consumed by j^{th} timed automaton as $a?$. Then the external action transition is written as $(\vec{s}, v) \xrightarrow{a} (\vec{s}[s'_i \wedge s'_j|s_i \wedge s_j], v')$. This transition is composed of two actions transitions of i, j single timed automaton given as $s_i \xrightarrow{a!} s'_i$ and $s_j \xrightarrow{a?} s'_j$. The final clock value $v' = v[\lambda_i := 0][\lambda_j := 0]$.

2.4. Timed Automaton Based Verification

In real-time systems, Timed Computation Tree Logic (TCTL) [15, 14, 16] is usually used in order to specify the system properties that need to be verified. A number of model checking tools are then used to check the veracity of these properties against the system model. These tools use well developed algorithms described in [8, 9, 10, 11] and other similar works. The TCTL properties that can be checked in these tools can be summarized as:

- *Reachability:* These set of properties deal with the possible satisfaction of a given state-based predicate logic formula in a possible future state of the system. For example, the TCTL formula $E\Diamond\phi$ is true if the predicate logic formula ϕ is eventually satisfied on any execution path of the system.

- *Invariance:* These set of properties are also termed as safety properties. As the name suggests, invariance properties are supposed to be either true or false throughout the execution lifetime of the system. For example, the TCTL formula $A\Box\phi$ is true if the system always satisfies the predicate logic formula ϕ. A restrictive form of invariance property is sometimes used to check if some logical formula is always true on some execution path of the system. An example of such a TCTL property is $E\Box\phi$.

- *Liveness:* Liveness of a system means that it will never deadlock, i.e. in all the states of the system either there will be an enabled action transition and/or time will be allowed to pass without violating any location invariants. Liveness is also related to the system responsiveness. For example, the TCTL formula $A\Box(\psi \rightarrow A\Diamond\phi)$ is true if a state of the system satisfying ψ always eventually leads to a state satisfying ϕ.

3. Remodeling the Reflex and Healing Framework

Our earlier definition of the Reflex and Healing (RH) framework provided in [5, 6] was inadequate to formally define the timed behavior of a Reflex Engine (RE). This was because the modeling formalism didn't provide for faults that can be signaled not only by the presence of an event, but also by the absence of a certain event in a given time interval. Moreover, in the earlier definition the notion of schedulers on a Reflex Engine was lacking. A scheduler is important in scenarios where a Reflex Engine has to choose between various triggered events and process them in a timely fashion. What follows in the next section is the new definition for a Reflex Engine which adds to our earlier design of RH framework in order to bridge the gaps described earlier. This new definition will allow us to model the RH framework as an equivalent network of timed automaton for analysis.

3.1. Real-Time Reflex Engines

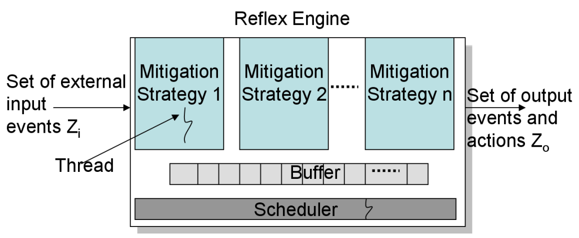

Figure 4. An overview of a real-time Reflex Engine

A real-time Reflex Engine, as shown in Fig. 4, comprises of multiple fault-mitigation strategies that takes action in the presence of certain input events which signify a fault condition. New strategies can be added online to a real time reflex engine by an upper level external reflex engine or by direct human intervention. Moreover, a supervisory higher level reflex engine can enable or disable a strategy.

The difference from our earlier design for RE is the presence of a scheduler and ability of fault-mitigation strategy to measure time progress. A formalized notion of a scheduler is required because availability of multiple events in the buffer could enable multiple strategies. A scheduler arbitrates the dispatch of strategies onto the available threads based on the priority of the notification event that triggered the strategy. Formally, a real-time Reflex Engine can be defined as:

Definition 2 (Real-Time Reflex Engine) *A real time reflex engine E_r is a tuple $< S, B, \mathscr{S}, \mathscr{S}', \mathscr{Z}_i, \mathscr{Z}_o >$, where S is the scheduler , B is a buffer, $\mathscr{S}$ is a parallel composition of all the enabled strategies and $\mathscr{S}'$ is the set of disabled strategies, $\mathscr{Z}_i$ is the set of all the possible inputs to a reflex engine and $\mathscr{Z}_o$ is the set of all the possible outputs generated by the reflex engine.*

All possible communication in and out of a real-time Reflex Engine[1] is carried out through event sets $\mathscr{Z}_i, \mathscr{Z}_o$. These event sets are a union of corresponding input and output event sets of all strategies implemented by that engine. In a multi-level hierarchy as the one motivated in [6], some events are reserved for communication between reflex engines that have a parent and child relationship.

We will now present the definition of the components of a real-time Reflex Engine.

Definition 3 (Fault Mitigation Strategy) *A fault mitigation strategy used in a real time reflex engine is state machine based failure management logic. It can be defined as a tuple $\mathscr{S} =< Q, q, Enable, Z_i, Z_{\tau^+}, \mathscr{T}, R, Z_o >$, where*

- *Q is the set of all possible states.*
- *q is the initial state.*
- *$Enable \in \{True, False\}$ is a Boolean flag used to enable or disable a strategy.*
- *Z_i is the set of all possible external events (input) to which the strategy is subscribed. Every strategy has two special events $start \in Z_i$ and $finish \in Z_o$ which are used to communicate with the scheduler.*
- *Z_{τ^+} is the set of all events generated due to passage of time.*
- *$\mathscr{T} \in Q \times (Z_i \cup Z_{\tau^+}) \times Q$ is the set of all possible transitions that can change the state of the strategy due to passage of time or arrival of an input event.*
- *Z_o is the set of all possible output events and mitigation actions generated by the strategy. For sake of brevity one can abstract the mitigation action as an event.*
- *$R : \mathscr{T} \rightarrow Z_o$ is the reflex action map which generates an output event every time a transition is taken.*

[1] In the rest of this paper we will use the term reflex engine and real-time Reflex Engine interchangeably without ambiguity.

In order to make the communication between the reflex engines and the plant unambiguous it is required that there is no overlap between subscribed events of two different strategies.

The time-based events associated with any strategy can be divided into two groups, internal time-based events and external time-based events. Internal time-based events are generated and used to measure time while the strategy is executing on a thread. However, if the strategy needs to measure time across two instances of execution, it can start a timer task, which shall generate the time-based event after the required passage of time. This timer is a high priority task and it should be allowed to execute non-preemptively until completion. We call the execution time duration of the timer as the lifetime of the timer. For the purposes of analysis we restrict the lifetime of the timer to the set of dense but countable positive rational numbers.

The execution of timers and strategies is governed by the scheduler. This scheduler picks up the input events from the buffer and then executes a number of strategies which in turn will perform the required mitigation action and generate some output events. We say that a strategy is triggered when the event to which it is subscribed is present in the buffer. We can describe the scheduler as follows:

Definition 4 (Scheduler) *A scheduler S, maps the input events in the buffer to the corresponding strategies. It then uses an arbitration scheme to select and execute a maximum m triggered strategies based on the priority of the notification event that triggered the strategy. Here m signifies the number of available threads.*

A primitive scheduler may employ First in First out (FIFO) scheme to dispatch triggered strategies on the available thread.

3.2. Problem Formalization

Consider a large scale real time system supported by a framework of hierarchical reflex engines for fault mitigation purposes. In order to guarantee the correctness of the system we need to be able to verify that the system possesses the properties of *liveliness, safety, and bounded time responsiveness*.

Property 1 (Safety) *The system shall always be schedulable i.e. it will not miss any deadline. Moreover, buffers of reflex engines shall never overflow.*

Property 2 (Liveness) *The system shall always be void of deadlocks.*

Property 3 (Bounded Time Responsiveness) *The mitigation response from a strategy shall be produced within certain time period off the generation of fault event.*

4. Analyzing the RH Framework

In this paper, we show that the real-time reflex and healing architecture can be treated as a network of timed automaton. With this semantic representation, we can pose questions of safety, liveness and bounded time response as corresponding model checking questions on the timed automaton models. Though, we have done this transformation manually, one can use the transformation rules to automatically synthesize the analysis model from a given specification of an RH framework.

4.1. Reflex Engines as a Network of Timed Automatons

In this section we will map all the components of a reflex engine (refer to section 3.1) namely its strategies, buffer, and the scheduler to timed automaton models.

4.1.1 Timer timed automaton

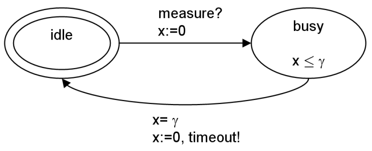

Figure 5. Timer timed automaton.

A timer timed automaton has two locations, $idle, busy$ and a single clock variable x. The transition from idle to busy is guarded by a synchronized event $measure?$ which allows the timer to start working. The invariant associated with $busy$ is given by $x \leq \gamma$, where $\gamma \in \mathbb{Q}^+$ is the time interval which has to be measured for the requesting strategy. The transition from $busy$ to $idle$ is guarded by constraint $x = \gamma$. Upon this transition from $busy$ to $idle$, the timer generates an output action $timeout!$ which is an input event for the requesting strategy. Fig. 5 shows this model.

4.1.2 Strategy Timed Automaton Model

Consider any strategy of a reflex engine modeled as described in Section 3. We can formulate an equivalent timed automaton model TA=$< \Sigma, S, S_0, X, I, T >$ in the following way:

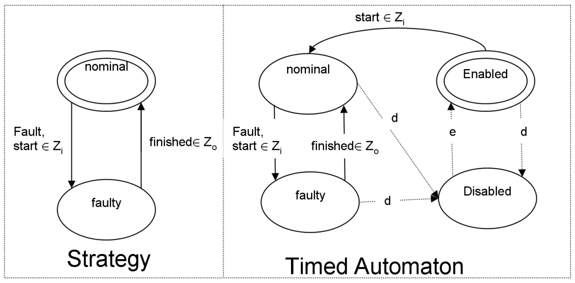

Figure 6. An example strategy and its corresponding timed automaton.

- States of the strategy Q are transformed as locations of the timed automaton model. Furthermore, two new locations $\{Enabled, Disabled\}$ are added to model the cases when strategy is enabled or disabled. Only when the strategy is enabled can it be triggered by the presence of events in the buffer and dispatched to a thread by the scheduler. Thus $S = Q \cup \{Enabled, Disabled\}$.

- Initial locations of the timed automaton, S_0 are derived from the initial states of the strategy plus the new locations $\{Enabled, Disabled\}$. Thus $S_0 = q \cup \{Enabled, Disabled\}$.

- In order to measure the time spent by the strategy in each of its locations we use a clock variable $x \in X$. This clock is used to generate internal time-based events while the strategy is executing.

- All invariants and guards to TA are specified by a time specification used for the generation of internal time-based events that can trigger a transition.

- Output actions of the timed automaton are the same as the set Z_o and are all written with a suffix of '!'. The input events are same as the set $Z_i \cup Z_{\tau^+}$ and are written with a suffix of '?'. Therefore, $\Sigma = Z_i \cup Z_{\tau^+} \cup Z_o$.

- Transitions T of the timed automaton are a union of transitions of the strategy i.e. $\mathscr{T}$ and transitions due to $Disabled$ and $Enabled$ locations. To model the ability of disabling the strategy from any of its states, we specify transitions from all locations of the timed automaton to the $Disabled$ location. Moreover, one transition from $Disabled$ to $Enabled$ location is specified to model the enabling of the strategy. Lastly, a transition is added from $Enabled$ location to the initial state q of the strategy.

- The reflex action map R of the strategy is created by mapping the output events Z_o to the corresponding transition of the timed automaton as output actions.

Fig. 6 shows an example of a strategy which has two states *nominal* and *faulty* and its corresponding timed automaton model. Note the extra states and transitions added in the timed automaton model. The events d, e are used by the supervisory reflex engine to disable or enable the strategy.

4.1.3 Buffer Timed Automaton Model

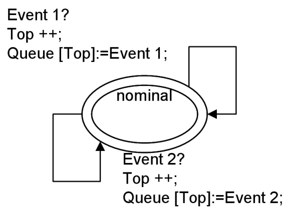

Figure 7. Buffer timed automaton listening to two different events. More events can be added to the model by creating new self transitions.

Buffer timed automatons have an associated shared array of integers, called a *queue*. In order to uniquely identify events in the model, we map them to the domain of positive integers. The Buffer TA (see Fig. 7) has only one location and no clocks. It uses an internal integer variable, Top, to keep track of the queue size. An event is added to the queue by using a self transition that also increments Top. If more than one event is available for addition to the queue at the same time, only one gets added while the others might be dropped. Moreover, a maximum queue size is set to drop events if the queue becomes full.

The possible dropping of events in a buffer is a safety concern for the RH framework. Therefore, a property that must be checked is to ensuring that no events are dropped.

4.1.4 Scheduler Timed Automaton Model

To express the mapping implemented by the scheduler of a reflex engine, we introduce the scheduler timed automaton construct as shown in Fig.8. This automaton shares the variable queue with the buffer timed automaton model. The possible locations for a scheduler model are $\{idle, select, Strategy\ 1, \cdots, Strategy\ n\}$, where there are n strategies in the reflex engine.

There are no clocks in the scheduler automaton. All transitions are synchronized via the $start$ and $finished$ events of the strategies. The $select$ state implements the scheduling policy by sorting the queue with a given priority associated with the incoming events. In the simplest case, all events have equal priority and the select algorithm picks up the event from the front of the queue without sorting the queue and starts the strategy which is subscribed to this input event. Upon processing the event from the top of the queue, the scheduler decrements the queue by deleting its front element and updating all the indices. A number of scheduler TA models can be executed as concurrent processes when more than one thread is available to the reflex engine.

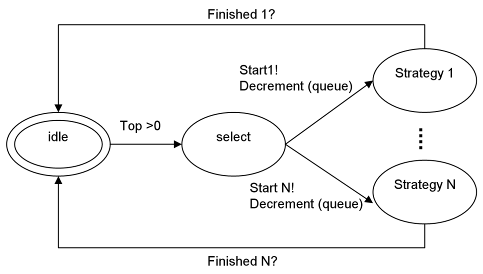

Figure 8. A scheduler timed automaton. The select state is the one in which the scheduling decision is made.

We have not delved into the modeling of main application and diagnosers as timed automatons in this paper. However, one can argue that the concerned model can be deduced from the abstraction of main application's behavior used for designing the fault-mitigation strategy.

5. Timed Automaton Based Verification

Once we transform the reflex engine architecture into a network of timed automatons we can utilize the model checking algorithms built in UPPAAL and KRONOS. However, first we have to translate the safety, liveness, and bounded time responsiveness questions for the reflex and healing architecture as a TCTL formula for the network of timed automatons:

- *Liveness* of the reflex and healing architecture will amount to checking if the system has any deadlocks. In UPPAAL this property is preprogrammed as a macro $A\Box$ not deadlock.
- *Safety* of the reflex and healing architecture can be checked by introducing an error location in all time automatons and forcing a transition to error if any deadline is missed. Then the checking of the safety property will amount to checking the reachability property $not\ E\Diamond\ error$.

- *Bounded Response Properties* can be formalized using the reachability property and liveness property. Suppose we have to check if $state = state1$ happens then $state = state2$ will happen within 5 time units. In order to formulate this property, we augment the time automaton with an additional clock called a formula clock, say z, whereby we reset its value to 0 on all the transitions leading to $state1$ and check if the liveness property $A\Box(state = state1 \rightarrow A\Diamond state = state1 \wedge z <= 5)$ is true or not.

6. Case Study

In this case study we consider a real-time experimental physics data processing system [17, 18], which employ a software task commonly referred to as a *filter* application. This application is responsible for filtering and marking the data generated from high-energy particle interactions inside a particle accelerator. Interactions are classified as scientifically significant or not, causing only interesting data to be stored.

In order to specify the real-time properties of the filter, we assume that it has a deadline of 25 milliseconds for processing one data element. It is known that the worst case execution time of the filter lies between 20 to 25 milliseconds. Fig. 9 shows the correct working filter modeled as a timed automaton. For this study we consider two types of faults:

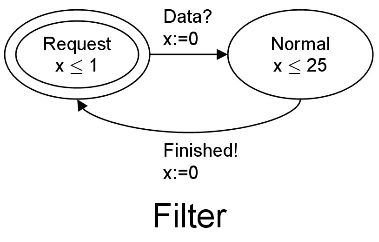

Figure 9. In the request mode the filter collects the next data within 1 millisecond. It must complete processing within 25 milliseconds.

- *CRC faults* (Fault 0) are caused if a number of data errors occur consecutively without giving enough time for the filter application to apply correction algorithms. For this study, we assume a fault if the filter witnesses consecutive occurrences of 5 CRC faults, with the time of arrival between all consecutive pairs of faults being less than 10 milliseconds. The mitigation action reconfigures the error detection and correction scheme to a more robust code such as Convolutional code [19].

- *Timeout faults* (Fault 1) are caused if a filter violates the 25 millisecond deadline for processing any data element assigned to it. The mitigation action (mitigation 1) for this fault is to reconfigure the filter, changing its precision, and to forward the data element to the next-level filters.

6.1. Description of Designed Strategies

For this system, we used one reflex engine, which has a single available thread executing two possible behaviors:

- *Strategy 0* uses a timer for measuring the time between two consecutive occurrences of a CRC error detection. This behavior produces a mitigation action if the inter-arrival time was less than 10 milliseconds for 5 consecutive pairs. Fig. 10 shows the state-based logic of this behavior along with its timer.

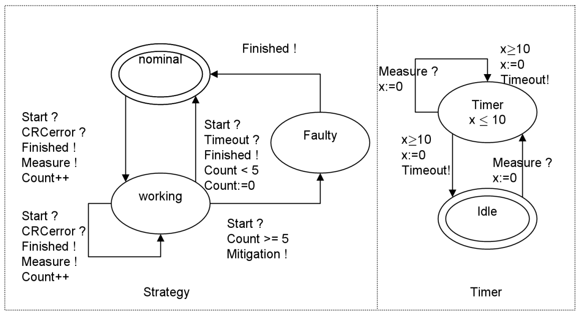

Figure 10. Logic of the first strategy.

- *Strategy 1* simply tracks the incoming event, Timeout fault. Its corresponding mitigation action (mitigation 1) causes a change of state to a reconfiguration state, shown in Fig. 9. It has two possible locations, idle and busy. Fig. 11 shows the logic of this state machine based behavior.

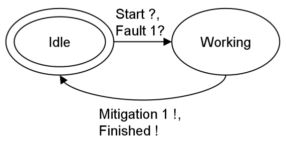

Figure 11. Logic of second the strategy. Start and finished events are used to communicate with the scheduler.

6.2. Analysis

We used the timed automaton model checking tool UPPAAL for this case study. Using the mapping procedure

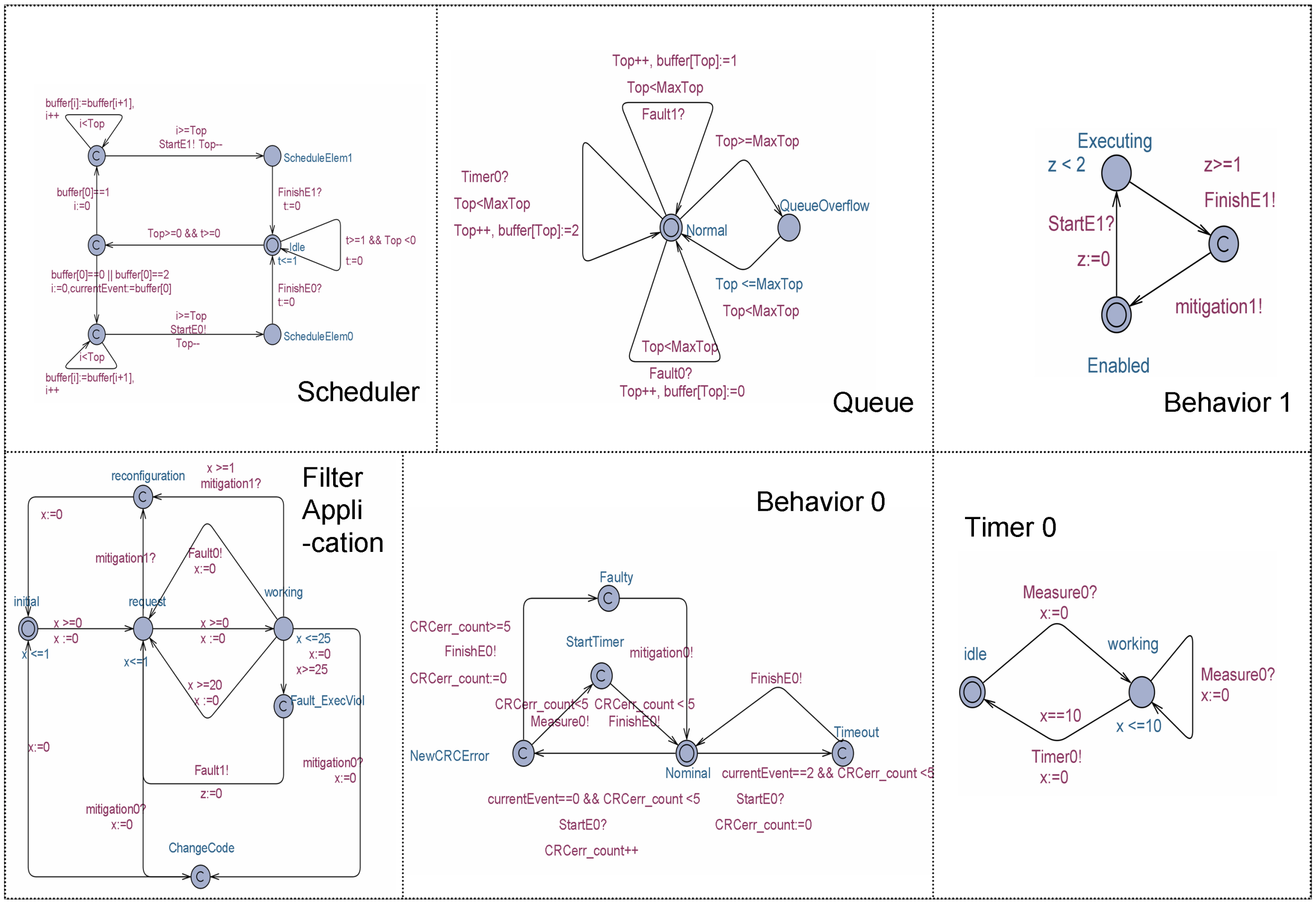

Figure 12. Network of timed automatons for the case study

outlined in Section 4.1, we transformed the reflex engine into a network of timed automatons. We used the integers 0,1, and 2 to represent fault 0, CRC error event, and timeout event of the timer in the buffer TA. The complete network of timed automatons for this configuration is shown in Fig. 12. Overall, the network has 10 synchronized input/output event/action pairs and 5 clocks. One clock is for the scheduler, one measures the execution time of behavior 1, one measures the filter time, one is for the timer, the last clock measures the bounded response time for fault 0.

In order to formalize a bounded response for faults (communicated through events) using state-based predicates we had to use special locations called Committed locations [16]. By definition, time is not allowed to pass in a Committed location. Therefore, one can identify the generation of a fault or mitigation event as an entry/exit to/from these locations. For example, we used the Committed locations $Filter1.Fault1\_ExecViol$ and $Filter1.reconfiguration$ to represent the occurrence of fault 1 and the corresponding mitigation action. Finally, the TCTL formula $A\Box(Filter1.Fault1\_ExecViol \rightarrow A \Diamond Filter1.reconfiguration \wedge Filter1.z <= 3)$ was used to check if mitigation 1 causes filter 1 to reconfigure within 3 milliseconds of any occurrence of fault 1.

Table 1 shows the various properties that were verified for this case study. The computer used to perform the UPPAAL analysis was a Pentium IV 3.0 GHz dual core machine with 512 MB memory. Of the 4 properties we checked, we discovered that the mitigation action for fault 1 (execution time violation fault) cannot happen within 2 milliseconds. Therefore, we can safely conclude that for this model we will need at least 2 milliseconds to recover from fault 1.

7. Conclusion and Future Work

In this paper, we presented a method for representing and analyzing real-time autonomic systems based on our existing RH framework. The real-time behavior of the system can be analyzed by mapping the engine components to a semantic domain of networked timed automatons.

We demonstrated our approach with a case study of a

Table 1. Results

Property	*TCTL formula*	*Results*
Liveness	$A\Box not$ deadlock	True
Safety: Queue will never overflow	$A\Box not$ Queue.QueueOverflow	True
Bounded Response: Mitigation 1 will happen within 3 milliseconds of fault 1	$A\Box(Filter1.Fault1\_ExecViol \rightarrow A\Diamond Filter1.reconfiguration \wedge Filter1.z <= 3)$	True
Bounded Response: Mitigation 1 will happen within 1 milliseconds of fault 1	$A\Box(Filter1.Fault1\_ExecViol \rightarrow A\Diamond Filter1.reconfiguration \wedge Filter1.z <= 1)$	False

simple filter application with two kinds of faults. Even though the timing data used for this case study was simulated, we have constructed similar autonomic systems with real embedded hardware processing real data. These tools and methods can be used to analyze the time-correctness of autonomous fault tolerant strategies before they are deployed in a real system.

It is noteworthy to point out that computer memory required for timed automaton model checking algorithms increases exponentially with the number of clocks. Therefore, as the size of system increases, the verification algorithm will require an infeasible amount of memory/time in order to execute to completion. Thus, it is not a straightforward task to scale this approach of model checking large systems as a network of timed automatons.

In order to address the scalability issues we will either have to reduce the total number of clocks in the system model or will have to investigate ways of finding symmetrical patterns in the hierarchical fault architectures so that we only need to verify the basic patterns of operation, instead of the complete system model. We are currently exploring both of these possibilities in order to make our approach effective.

8. Acknowledgments

This work is supported by NSF under the ITR grant ACI-0121658. Authors are also grateful for the help of their colleagues in the Real Time Embedded Systems (RTES) group.

References

[1] R. Sterritt and M. G. Hinchey. Autonomic computing - panacea or poppycock? In *ECBS*, pages 535–539, 2005.

[2] R. Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, April 2005.

[3] M. Parashar and S. Hariri. Autonomic computing: An overview. In *UPP*, pages 257–269, 2004.

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2005.

[5] S. Shetty, S. Nordstrom, S. Ahuja, D. Yao, T. Bapty, and S. Neema. Systems integration of large scale autonomic systems using multiple domain specific modeling languages. In *ECBS*, pages 481–489, 2005.

[6] S. Nordstrom, S. Shetty, S. K. Neema, and T. A. Bapty. Modeling reflex-healing autonomy for large scale embedded systems. *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Autonomic Computing*, to be published in 2006.

[7] D. Yao, S. Neema, S. Nordstrom, S. Shetty, S. Ahuja, and T. Bapty. Specification and implementation of autonomic large-scale system behaviors using domain specific modeling language tools. In *Proceding of International Conference on Software Engineering and Practice*, June 2005.

[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[9] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[10] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaala tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[11] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 126:110–122, 1997.

[12] P. Krcál and W. Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *TACAS*, pages 236–250, 2004.

[13] G. Madl, S. Abdelwahed, and G. Karsai. Automatic verification of component-based real-time corba applications. In *RTSS*, pages 231–240, 2004.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge MA, USA, 2000.

[15] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University press, 2 edition, 2000.

[16] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004.

[17] J. G. et. al. Clustered data acquisition for the cms experiment. In *International Conference on Computing In High Energy and Nuclear Physics*, September 2001.

[18] S. Kwan. The btev pixel detector and trigger system. In *FERMILAB-Conf-02/313*, December 2002.

[19] A. J. Viterbi. Convolutional codes and their performance in communication systems. *IEEE Transactions on Communications*, 19(5):751–772, October 1971.