Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Abhishek Dubey · Steve Nordstrom · Turker Keskinpala · Sandeep Neema · Ted Bapty · Gabor Karsai

Towards a Verifiable Real-Time, Autonomic, Fault Mitigation Framework for Large Scale Real-Time Systems

Received: 15 September 2006 / Accepted: 1 December 2006

Abstract Designing autonomic fault responses is difficult, particularly in large-scale systems, as there is no single 'perfect' fault mitigation response to a given failure. The design of appropriate mitigation actions depend upon the goals and state of the application and environment. Strict time deadlines in real-time systems further exacerbate this problem. Any autonomic behavior in such systems must not only be functionally correct but should also conform to properties of liveness, safety and bounded time responsiveness. This paper details a real-time fault-tolerant framework, which uses a reflex and healing architecture to provide fault mitigation capabilities for large-scale real-time systems. At the heart of this architecture is a real-time reflex engine, which has a state-based failure management logic that can respond to both event- and time-based triggers. We also present a semantic domain for verifying properties of systems, which use this framework of real-time reflex engines. Lastly, a case study, which examines the details of such an approach, is presented.

Keywords Autonomic computing · Fault tolerance · Fault mitigation · Real time systems · Reflex · Healing · Model checking · Timed automaton

1 Introduction and Problem Motivation

The increased influence of smaller, more powerful and more ubiquitous computers in human lives demands a higher degree of reliability from these systems. They are expected to consistently produce correct outputs with very small tolerance for errors. However, as computing technology grows more prominent, the complexity of system design is on the rise. In order to increase reuse and reduce cost, designers are leveraging composition of smaller systems to construct a larger system, which not only increases complexity but also stresses the design process, which makes it difficult to achieve high reliability. Therefore, it is imperative to accept that failures can and will occur, even in meticulously designed systems, and take steps to study the effect of these failures and design proper measures to counter those failures.

Detection and mitigation of faults in large-scale systems with hundreds of components is a crucial and challenging task. Even though the correct operation depends on individual components as well as their interactions, it is necessary that the systems do not suffer a catastrophe every time a subcomponent misbehaves. Most practical systems have to employ some kind of fault detection, the simplest of which can be a threshold sensor and alarm for every critical aspect of the system. In past decades, the problem of fault diagnosis has received considerable attention in literature and a number of schemes based on fault trees [38], quantitative analytical model-based methods [28; 10], expert systems [29; 10], and model-based reasoning systems [17; 21] have been proposed for both continuous systems [12], as well as discrete event systems [20; 33; 1; 22].

While fault diagnosis is certainly a key aspect of the problem, fault mitigation is another crucial problem that deserves added attention. Arguably, one method of fault mitigation is to let humans in the loop to take decision. Alternatively, a component of the computer-based system can itself take fault mitigation decisions. The former technique is useful in systems, which do not necessitate quick reactions. However, in most critical systems, where fault mitigation tends to require time-bounded responses, the latency in decision due to human involvement in the control loop is unacceptable. For example, any human mitigation decision in interplanetary space exploration systems has to travel the vast distances of space between the earth-based station and the spacecraft, which may lead to a situation where it is impossi-

A. Dubey · S. Nordstrom · T. Keskinpala · S. Neema · T. Bapty · G. Karsai
Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37203, USA
E-mail: {dabhishe, steve-o, tkeskinpala, sandeep, bapty, gabor}@isis.vanderbilt.edu

ble to correct for a fault before it becomes threatening to the system [37]. In such and similar critical systems, the only option is to let the computer-based system work autonomously and take the fault mitigation decisions itself.

This drives the need for autonomic computing to address the needs of system reliability. Such self-managing systems should be also self-aware, self-configuring, self-healing, and self-protecting [35; 36; 27]. A motivation for this paradigm comes from biological systems which employ a reflex and healing strategy to mitigate local faults quickly yet still allow a hierarchy of decentralized "managers" to be responsible for healing of widespread faults.

Our previous work in large scale autonomic computing yielded a *Reflex and Healing (RH)* framework presented in [34; 26; 39], which employs a hierarchical network of decentralized fault management entities, called reflex engines. These reflex engines are instantiated as state machines, which change state and initiate reflexive mitigation actions upon occurrence of certain fault events. Other researchers have also proposed to equip systems with decentralized self-management algorithms [31], and an alternative technique has also been developed that consists of a hierarchy of managing entities placed in the system with control decisions being made based on partial knowledge of the system [11].

In order to apply the aforementioned techniques to systems, which are real-time in nature, additional considerations are required. The addition of time deadlines exacerbate the problems associated with self-management. Under such conditions, one has to not only be concerned about the correctness of a given self-management task; one must also ensure that a task finishes before certain deadlines expire [6]. Fault mitigation tasks, which lead to missed deadlines, are, therefore, considered faulty in hard real-time systems.

Even in systems that are not real-time (and hence do not have specific task deadlines), conditions might require that a fault mitigation is completed within a certain boundary from the time of the fault occurrence. Such requirements of time-bounded responses are true for almost all systems that require a fault-tolerant framework.

In such a context, any autonomic computing technique used must respect time constraints. Moreover, in order to ensure correct operation, the autonomic system must conform to the following properties:

1. *Liveness:* If the system was deadlock free, the addition of autonomic behaviors shall not lead to a deadlock in the system.
2. *Safety:* The system shall 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 shall be executed within a specified time bound.

It is necessary to check that the system and its corresponding fault tolerance framework conform to the properties mentioned above. However, an initial investigation revealed that the previous RH framework was unsuitable for verifying these properties. This is because the framework lacks the formalism required for modeling the passage of time. In addition, the previous RH framework also lacks formalism that can allow arbitration between several eligible fault mitigation strategies.

In this paper, we extend the results presented in [9] and detail work toward a verifiable fault tolerant framework for real-time systems. This framework uses a reflex and healing architecture in order to provide fault mitigation capabilities. At the heart of this architecture is a new real-time reflex engine, which implements a state-based failure management logic that can respond to both event- and time-based triggers. Furthermore, this work elaborates our existing model with concepts of schedulers that can arbitrate between several eligible mitigation strategies, which can be shown to respond to failure events within a given time limit. With the additional help of a case study, we show how one can now use the semantics of timed automata [3; 15] to analyze the real-time properties of the framework using UPPAAL [5], a model checking tool for networks of timed automatons.

2 Fault Diagnosis and Mitigation

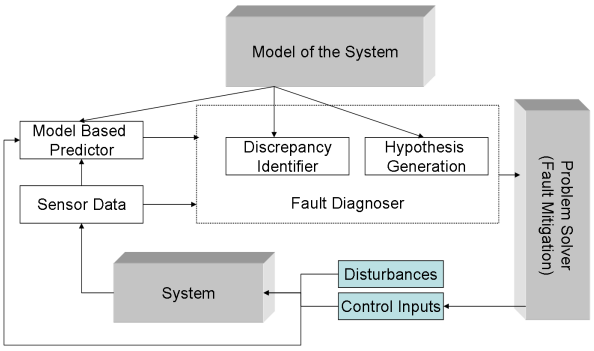

Fig. 1 Usual approach to fault diagnosis and mitigation.

Figure 1 depicts a general approach of model-based fault diagnosis and mitigation. The system model is an abstraction of the plant and is used to estimate the current system state and output using prediction. If the predicted data and measured sensor data differ, the discrepancies are identified to form a symptom set that can be used to generate hypotheses about the faults. The problem solver uses these hypotheses to generate fault mitigation actions, which provide correctional input to the plant.

As we look more in detail at the problem of fault diagnosis and mitigation, a number of questions arise.

What type of model should we use for the system? What are the mechanics associated with the communication between various components? What is the architecture of this problem solver? Answers to these questions are open ended, as there is no "one fits all" solution.

2.1 Large Scale Real Time Systems

Ther real-time embedded systems group (RTES) [2] is a research collaboration of computer scientists, electrical engineers and experimental physicists whose aim is to provide new research in the areas of large-scale, fault-tolerant, high-performance embedded systems. Research efforts of the RTES group are focused on finding new tools and techniques which strengthen the design and development of large scale computation systems used for online phenomena capture, signal processing and data acquisition for high energy physics experiments.

High energy physics (HEP) experiment setups such as the Compact Muon Solenoid (CMS) [13] at the Large Hadron Collider (LHC) at CERN are multi-year, multi-million dollar projects. The estimated particle interactions in these experiments have a periodicity as high as 25 ns, which results in aggregate data rates of several terabytes/second with a petabyte/year in permanent storage needs [13; 19]. It is estimated that an overwhelmingly large percentage of the observed particle interactions will not lead to new science in the areas of high energy physics. Given the size and rate of the data generated during an experiment run, it is infeasible to record complete particle interaction data for a later examination. Therefore, massive real-time embedded systems must be used to execute data-reductive algorithms called "filters", which perform the necessary examination of the collected data to isolate and retain only the observations of interesting phenomena.

Data acquisition and processing systems of high energy physics (HEP) experiments require high throughput and low latency. These systems are typically composed of thousands of data processing nodes, which incorporate various computational resources such as FPGAs, DSPs, commodity PCs, high-bandwidth fiber and copper interconnects. Given the long durations of typical experiments, the failure of these systems are an expected occurrence.

The high costs involved and the number of computing nodes required makes it infeasible to use traditional redundancy as the primary fault tolerance approach. Furthermore, it is required that the implemented fault tolerance approach should be able to localize and isolate faults and yet maintain some form of acceptable system operation. The system must also possess capabilities to run online recovery procedures, compensate by changing system parameters such as thresholds, and prune out bad computing nodes with minimum impact on system performance. It should be noted that these problems are not unique to the field of HEP but are also present in other fields such as space systems (see [37]).

Like any model-based fault diagnosis and mitigation approach, the first step is to choose a suitable model. The choice of model has profound affect on simplicity of implementation as well as guarantees that can be made about the properties, which the system is required to satisfy.

3 Abstracting Real-Time Systems as Discrete Event Systems

The act of *modeling* has been described in [32] as, "representing a system formally in order to describe and analyze the working of some relevant portions of the concerned system". A widely used abstraction that is valid for most embedded systems is a discrete event system (DES) model.

A DES is a discrete state, event-driven system in which the state evolution depends entirely on occurrence of discrete events over time [7; 30]. The event driven property implies that the state of the system changes at discrete points in time which correspond to *instantaneous* occurrences of events.

In a DES framework, a description of an application's (plant) behavior is given in the form of a finite automaton. Any behavior of the plant can be explained as an execution of this automaton (i.e., a sequence of events). These events can be either *observable* or *unobservable*. The objective is to find a diagnoser that can detect the occurrence of a fault event within a limited number of steps of its occurrence using the observable events. Note that our objective is not to solve this diagnosis problem, rather we are concerned with the mitigation of faults. For the diagnosis problem, readers can refer to [33].

Based on this discrete event model, we presented a hierarchical reflex and healing (RH) framework in our earlier papers [34; 26; 39] for providing fault mitigation capability in large-scale systems. In large-scale reflex and healing systems, software applications use an event based communication scheme. Each application generates a regular heartbeat, which informs an observer that the application is alive. Moreover, each application has a self-monitoring process and generates failure events in case of any problem. Given the failure events, the task of the framework is to contain faults, correlate diagnosis from different regions, and decide on an appropriate mitigation strategy.

3.1 The Reflex and Healing Architecture Using a Discrete Event Model

As shown in Figure 2, the RH framework employs a hierarchical network of fault management entities, called

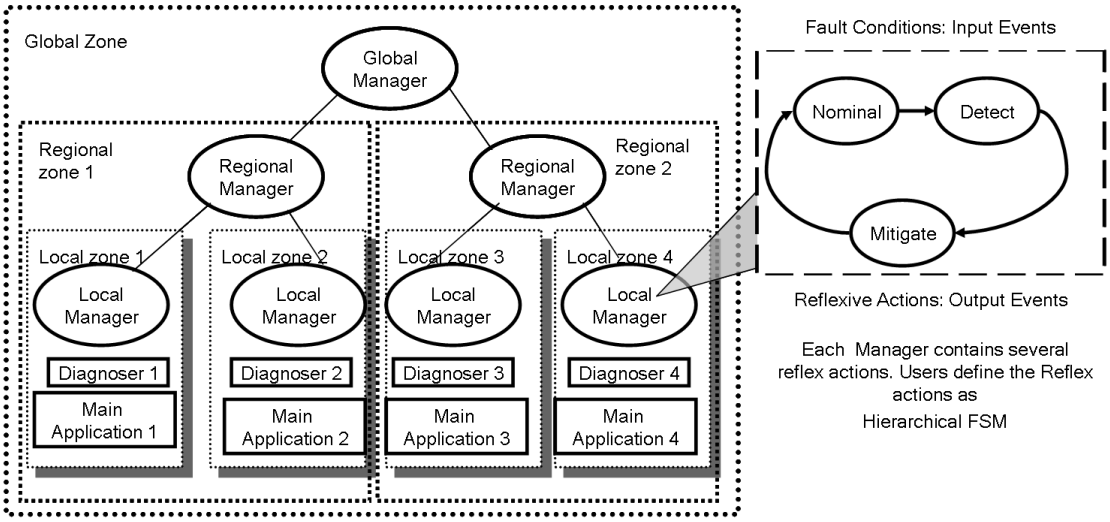

Fig. 2 Hierarchical fault management scheme in reflex and healing architecture. The reflex actions are user defined FSM. Communication between managers themselves and to/from the plant is via predefined events.

reflex engines, whose sole purpose is to implement fast reflexive fault-mitigating actions.

Reflex engines are customized software processes that perform monitoring and execute fault mitigation actions. It is possible to configure a reflex engine using an external source by programming their state machine. As seen in Figure 2, a reflex engine consumes input event (e.g., fault events) and produces output events (e.g., mitigation actions). Upon occurrence of a certain class of fault events, a reflex engine, which is specifically listening for that event, performs fault mitigation by changing its state and executing a set of actions upon the transition.

In order to manage scalability, all reflex engines are divided into logical hierarchical levels called zones. Each of these zones contains a manager whose area of observation, is limited to that particular zone. There are three main categories of managers, which are:

1. *Local Manager:* These managers are nearest to the main applications. They are usually responsible for managing one user process, as shown in Figure 2.
2. *Regional Manager:* These managers supervise and communicate with a number of subordinate managers which are in their area of observation. A regional manager has a wider area of observation and can correlate diagnosis to ascertain if a problem is common to a number of user applications and take coordinated mitigation action. Note that there can be a number of regional levels.
3. *Global Manager:* This manager lies at the top of hierarchy. It coordinates regions and performs any required optimization of the system. This process is usually referred to as *healing*. For more detail on the complicated process of healing readers can refer to [24; 25].

These reflex engines operate concurrently with user applications. The state machine based failure management logic in each reflex engine responds to faults as they are observed. To minimize fault propagation, intra-level communication between reflex engines is forbidden.

There are definite advantages in following a rigid hierarchical structure with strict communication protocols in place. First, fault reaction time improves because the mitigation decisions are made closer to the fault source. Secondly, scalability improves because new fault managers can be added easily to a zone without disrupting other zones. Lastly, chances for the propagation of faults from one user application to another application are greatly reduced.

The development of the framework to date lacks the details necessary to make real-time guarantees. This is because the discrete event-based model used does not capture time. Hence, one cannot analyze any real-time property such as bounded time response. Moreover, this model does not specify how the operating system under the reflex engine affects its execution sequence. In addition, the state machines can be non-deterministic due to the runtime and the state machine definition. Specifically, if a reflex engine has more than one fault strategy enabled, which one is executed? For these scenarios, we need to incorporate the notion of a scheduler, which is necessary when a reflex engine has to choose between various triggered events in order to process them in a consistent fashion.

First, we need to extend the model for real-time systems to capture notion of time. Then we need to aug-

ment the existing RH framework with components that can describe the execution behavior of a reflex engine in detail. Finally, we need to provide exact semantics to this new model using a real-time model of computation, which allows for formal analysis of time based properties outlined in Sect. 1.

4 Abstracting Real-Time Systems As Timed Automatons

A fault-tolerant framework that is capable of providing time guarantees and detecting time anomalies requires a model, which can capture the notion of continuous time. Classically, the timed automaton (TA) model [3; 15] has been used for abstracting time-based behaviors of systems which are influenced by time. This approach has been used to solve scheduling problems [18; 23] by modeling real-time tasks and scheduling algorithms as variants of timed automatons and performing reachability analysis on the equivalent region graph automaton [3].

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. These transitions are assumed 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 the 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.

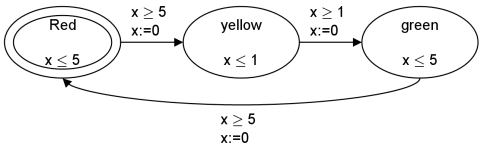

Fig. 3 A timed automaton model of a behavior of traffic light.

It is customary to draw a timed automaton model as a directed graph with nodes, drawn as circles or ellipses, which represent locations and edges, which represent transitions. Initial locations are marked using concentric ellipses or circles. Figure 3 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, 1, and 5 units, respectively.

The state of a timed automaton is a pair of the current evaluation of clock variables and the current location, $Q = (s, v)$, where $s \in S$ and $v : X \rightarrow \mathbb{R}^+$ is the clock value map, assigning each clock a positive real value. At any time, all clocks increase with a uniform unit rate, which, along with events, enable transitions from one state of the timed automaton to another. Since there are an infinite number of possible clock evaluations, the state space of a timed automaton is infinite. The transition graph over this state space, $A =< \Sigma, Q, Q_0, R >$, is used to describe the semantics associated with a timed automaton model. The initial state of A, Q_0 is given by $\{(q, v) | q \in S_0 \wedge \forall x \in X(v(x) = 0)\}$.

The transition relation R is composed of two types of transitions: delay transitions caused by the passage of time, and action transitions, which lead to a change in location of a timed automaton. Before proceeding further with transitions, it is necessary to first define some notation.

Let us define $v + d$ to be a clock assignment map, which increases the value of each clock $x \in X$ to $v(x) + d$. For $\lambda \subseteq X$ we introduce $v[\lambda := 0]$ to be the clock assignment that maps each clock $y \in \lambda$ to 0, but keeps the value of all clocks $x \in X - \lambda$ same. Using these notations, we can define delay and action transitions as follows:

* *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 occurrences 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 [5; 8; 40].

A network of timed automatons is a parallel composition of several timed automatons [8]. Each timed automaton can synchronize with any other timed automaton by 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 the synchronizing events, tools like UPPAAL [5] and KRONOS [40] have introduced concepts of integer variables and arrays of integers, which can be shared between different timed automatons. Moreover, UPPAAL has introduced some additional notions in order to make modeling as timed automaton simpler [4]. Some of the notable additions that we will later use are:

* *Committed Locations:* The semantics of a committed location is that if any timed automaton of the network is in a committed location it is assumed that time cannot pass. In such a case, the only possible transition for the whole network is the one that goes out of the committed locations.
* *Urgent Channels:* A usual synchronized action transition can be ignored if any other transition is possible. However, if the synchronized event is urgent, then the transition becomes compulsory.

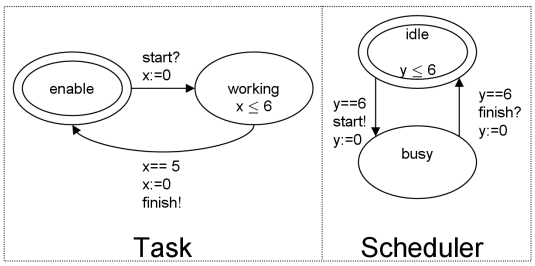

Fig. 4 A network of two timed automatons.

The semantics of network-timed automatons are also given in terms of transition graphs. A state of a network is defined as a pair $(\mathbf{s}, v)$, where $\mathbf{s}$ 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 the same as those for a single timed automaton. However, the action transitions are composed of internal actions and external actions.

An internal action transition is a transition, which happens in any one timed automaton of the network, independent of other timed automatons. On the other hand, an external action transition is a synchronous transition between two timed automatons. For such a transition, one timed automaton produces an output event on its transition leading to a change in its location (denoted as $a!$), while the other timed automaton consumes that event (denoted as $a?$) and takes the transitions leading to a change in its location. An external action transition cannot happen if any of the timed automatons cannot synchronize. Figure 4 shows model of a scheduler executing in parallel to a task. The scheduler enables the task 6 time units after it enters the idle location. The two automatons synchronize using start and finish event/action pair.

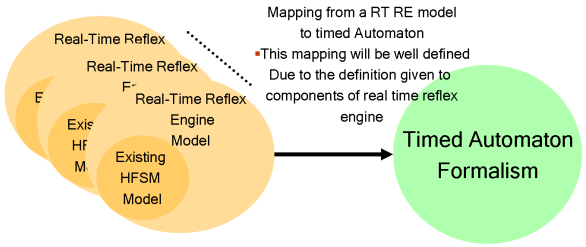

Fig. 5 The previous reflex engine has been augmented to allow analysis within the semantic domain of networks of timed automatons.

The next section details the real-time reflex and healing framework. Fig 5 illustrates the approach at a high level. The first major addition in the new framework was the concept of timers, which generate timeout events after a set period. This timer, when used in parallel with an event-based reflex engine, enables the reflex engine to mitigate time-based faults. Then, by using the network of timed automatons as a semantic domain for the real-time reflex and healing framework, we are able to use timed automaton model checkers for verifying real-time properties.

5 The Real-Time Reflex and Healing Framework

A real-time reflex and healing framework has a hierarchical structure identical to that, which is described in Sect. 3.1. However, each manager model is now more capable than a simple event-based state machine.

In order to provide an overview of the workings of the real-time reflex engine we need to explore the notion of execution threads. For the moment, let us assume that each manager runs on a dedicated node, which has some kind of an operating system. The reflex engine will use the threads leased from the operating system. It is ideal if this operating system is a real-time operating system; this abstraction, however, will work even with non real-time operating systems, though in such cases any real-time guarantee will depend upon the assumption that no RH task will be delayed for an arbitrarily long time. It should be noted that this thread abstraction would still hold true if more than one manager is installed on a node.

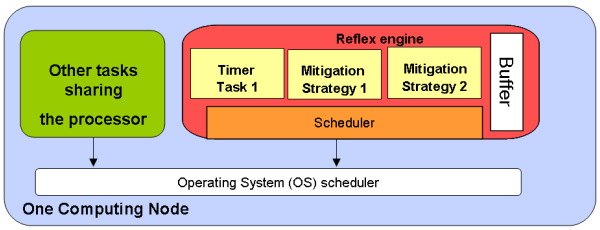

Fig. 6 A computing node in the real time reflex and healing framework.

Fig 6 describes the basic structure inside a managing node. The lowest layer is the operating system scheduler. There can be multiple fault mitigation strategies (state machines) running on that node. In an ideal situation, the operating system is able to provide an execution thread to each of the strategies. If there are only a limited number of threads, then the reflex-engine scheduler is needed. A reflex-engine scheduler has a fixed number of threads that it can lease from the operating system to then schedule the different mitigation strategies. A buffer provides temporary storage of input fault events that the reflex engine scheduler schedules with the corresponding mitigation strategy. Finally, a number of high-priority timers measure the time lapsed for the mitigation strategy.

We can now detail these components and describe their semantics using a model of networked timed automatons.

5.1 Real-time Fault Mitigation Strategies and Timer Tasks

A real-time reflex engine uses fault mitigation strategies as reflexes. Formally, we can define a strategy as:

Definition 2 (Fault mitigation strategy) A fault mitigation strategy used in a real time reflex engine is state machine based failure management logic, $\mathscr{S} =< Q, q, Enable, Z_i, Z_{\tau^+}, \mathscr{T}, R, Z_o >$, where

* Q is the set of all possible states.
* $q \in Q$ 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.

The time-based events associated with any strategy can be divided into two groups, internal and external. Internal time-based events are generated while the strategy is executing on a thread. 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. Thus, fault mitigation strategies use timer tasks to measure time for them and generate an event that wakes them up.

Definition 3 (Timer task) A timer is a task that executes non-preemptively until completion. This task generates a timeout event that is subscribed to by the corresponding fault mitigation strategy. We call the execution time duration of the timer as the lifetime of the timer.

These strategies and timer tasks have a one-to-one mapping with a corresponding timed automaton model. For the purposes of analysis, we restrict this lifetime of the timer to the set of dense but countable positive rational numbers. This is necessary because we wish to use the timed automaton model to govern the semantics of a timer task.

5.1.1 Timer timed automaton

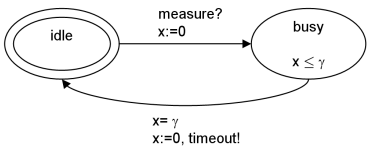

Fig. 7 Timer timed automaton.

A timer task is mapped to a timed automaton, which has two locations, *idle*, *busy* and a single clock variable x. We call this timed automaton a timer timed automaton. 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. Figure 7 shows this model.

5.1.2 Strategy Timed Automaton Model

Consider any strategy of a reflex engine modeled as described in Sect. 5.1. We can formulate an equivalent

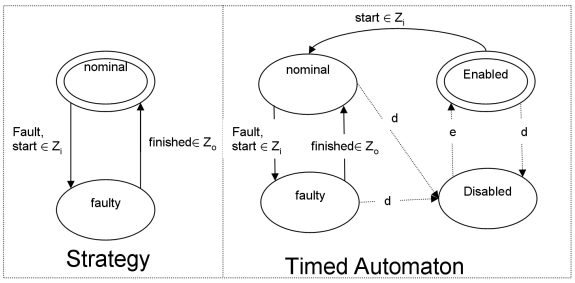

Fig. 8 An example strategy and its corresponding timed automaton.

timed automaton model $TA{=}< \Sigma, S, S_0, X, I, T >$ in the following way:

* States of the strategy Q are mapped to locations of the timed automaton model. Furthermore, two new locations $\{Enabled, Disabled\}$ are added to model the cases when a 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 location of the timed automaton, S_0 is derived from the initial states of the strategy plus the new locations $\{Enabled, Disabled\}$. Thus $S_0 \in q \cup \{Enabled, Disabled\}$.
* 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 in 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.

Figure 8 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.

5.2 Buffer

A buffer is a FIFO, which provides an interface for events that are coming into a reflex engine. Since a reflex engine has a number of strategies that might be more than the number of available threads, the scheduler has to arbitrate as to which strategy should execute. For this purpose, the scheduler can sort the buffer and allow execution of only those strategies that have a subscribed event at the front of the buffer.

There are several issues associated with a buffer that affect the correct operation of the RH framework. If the buffer is not of adequate size and the incoming rate of events is greater than the rate at which they are being processed, the buffer might drop some events. Such mishaps will directly affect the safety property of the real-time framework.

5.2.1 Buffer Timed Automaton Model

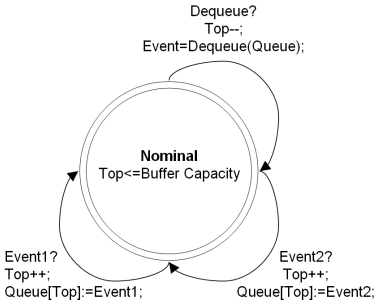

Fig. 9 Buffer timed automaton listening to two different events. More events can be added to the model by creating new self-transitions. The event, *dequeue*, is used to pop the event from the front of the queue.

Buffer timed automatons have an associated shared array of integers, called a *queue*. In order to identify events in the model, we map them to the domain of positive integers. The Buffer TA (see Figure 9) 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*. A maximum queue size is set to drop events if the queue becomes full, however, such a situation is undesirable.

5.3 Scheduler

Scheduler is an important component of a real-time reflex engine. Based on the number of threads available

from the main operating system, the scheduler governs and arbitrates the execution of timer tasks and strategies. It picks up the input events from the buffer and then executes a number of strategies, which in turn will perform the required mitigation actions 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. It can be defined as follows:

Definition 4 (Scheduler) A scheduler, S, maps the input events in the buffer to the corresponding strategies. It uses an arbitration scheme to sort the buffer based on the priority of the notification events. It then executes a maximum m (number of threads leased from the operating system) strategies based on m number of events from the head of the buffer. A number of priority schemes can be used with the notification event. The simples being equal priorities. In such a case, the scheduler will never sort the buffer and always pick m events from the head of the buffer.

5.3.1 Scheduler timed automaton model

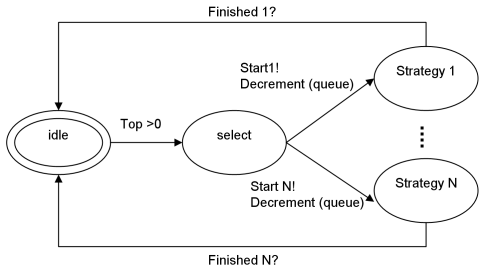

Fig. 10 A scheduler timed automaton, which has only one thread. The select state is the one in which the scheduling decision is made.

The semantics of a real-time reflex engine scheduler is a network of m timed automatons, m is the number of available threads. Each of these individual timed automatons is identical to the one illustrated in Figure 10. Working in parallel, one of these timed automatons keeps control of execution on one of the threads available to the scheduler. The number of threads that are available for the strategies is equal to the number of threads leased from the operating system minus the number of threads for the timers (timers are non-preemptive and must have a thread for their executions).

These timed automatons start from the idle location. When the buffer is non-empty, the location changes to select, which implements the scheduling policy by sorting the queue with a given priority associated with the incoming events. It then pops the event from the top of queue, transitions to the corresponding strategy location and passes the thread of execution to the strategy by using the start event. When the strategy finishes execution, it generates a finished event that returns the timed automaton to the idle state.

With all its subcomponents defined, we can now review the operation of a real-time reflex engine.

5.4 Real-Time Reflex Engine

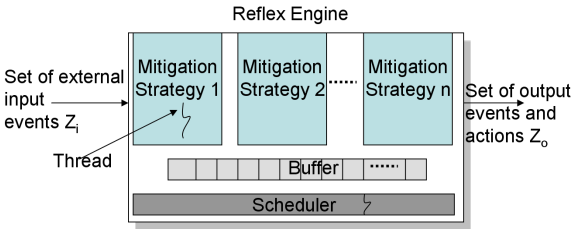

Fig. 11 An overview of a real-time reflex engine

A Real-time Reflex Engine (RTRE), as shown in Figure 11, is comprised of multiple fault-mitigation strategies that take action in the presence of certain input events, which signify a fault condition. New strategies can be added 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. Formally, a RTRE can be defined as:

Definition 5 (Real-time reflex engine) A real time reflex engine is a tuple $E_r = < S, B, \mathscr{S}, \mathscr{S}', \mathscr{Z}_i, \mathscr{Z}_o >$, where S is the scheduler , B is a buffer, $\mathscr{S}$ is a parallel composition of all the enabled strategies. $\mathscr{S}'$ 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 RTRE 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 [26], some events are reserved for communication between reflex engines that have a manager/subordinate relationship.

Since a reflex engine is a parallel composition of its subcomponents, its timed automaton is also a parallel composition of the component's timed automatons. Hence, we can consider a real-time reflex engine to be a network of timed automatons. In the same spirit, we can consider a number of reflex engines in a framework to be a larger network of timed automatons.

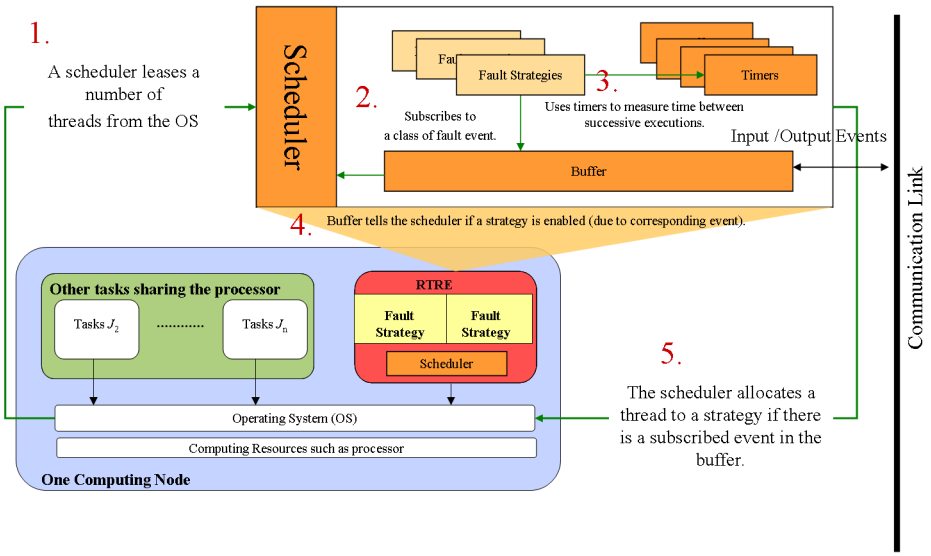

Fig. 12 Overview of the chain of operations inside a computing node, which has a real time reflex engine.

5.5 Sequence of Operations inside a Real-Time Reflex Engine

Figure 12 illustrates the operation inside a computing node, which houses a RTRE. Once it is operational, the scheduler leases a number of threads from the operating system. The fault strategies then inform the scheduler about their subscribed events through the publish subscribe mechanism.

When a new event arrives in a buffer, it signals the scheduler. If the buffer is full, then the event is dropped. This is an undesirable condition, which necessitates prior verification to check that such a situation will never arise. Once there is an event in the buffer, the scheduler waits for a thread to be available and then triggers the corresponding strategy and transfers the thread to that strategy. A strategy then takes the corresponding event and transfers the output event onto the communication link. If the strategy needs to measure time, it starts a timer task, which is non-preemptive and runs until completion or its termination, whichever happens first.

6 Analysis of Real-Time Reflex and Healing Framework

In the last section, we detailed the mapping between a real-time reflex and healing framework and a corresponding network of timed automatons. Since we wish to analyze the network's real-time properties, we can use timed computation tree logic (TCTL) [16; 8; 4] to specify the system properties. Then we can use model-checking tools to check the veracity of these properties against the system model. In general, these model-checking tools check the following properties:

* *Reachability:* These sets 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 sets 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.

It is possible to use these general classes of TCTL properties to map the real-time properties of a RH framework to corresponding TCTL properties for the network of timed automatons.

* *Liveness* of the RH architecture amounts to checking if the system has any deadlocks. In UPPAAL $A\Box$*not deadlock* is the specification for this property.
* *Safety* of the RH architecture requires a check for any violation that might lead to not finishing a task

on time. Safety violations also include checking for queue overflow conditions. One way for checking a safety property is to introduce an error location in all time automatons and force a transition to this error location if the queue overflows or if a time deadline expires. 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.

In the next section, we will present a case study that will help in clarifying the ideas introduced in this paper so far. In this case study, we will use UPPAAL for analysis purposes.

7 Case Study

Typical HEP data processing systems experience a variety of both temporal and persistent faults. Persistent faults are those in which software or hardware components undergo terminal failure and need to be replaced. Temporal faults, however, are sporadic in nature and are not necessarily indicative of direct failure of the software or hardware. One example of temporal faults is data stream corruption.

Our approach toward designing tools and techniques for advancing the capabilities of HEP data processing systems has included the means for non-experts (physicists and plant engineers) to easily design and deploy fault mitigation behaviors, which are specific to their domain of experience. Systems that provide this capability have been shown to be beneficial for HEP computing [14]. However, with this great benefit comes the risk that non-expert users might create behaviors, which may, at best, cause unexpected behavior or, at worst, be harmful to the system.

The following case study presents a scenario in which an existing system [2] is augmented with user-defined behaviors for recovering from temporal faults due to data stream corruption. These behaviors are analyzed using UPPAAL [4], a tool for verifying behavioral models expressed as timed automata.

7.1 Experiment Setup

The data rate at which physics events are generated in an actual HEP experiment are on the order of several Tera Byte per second [19]. Since only a few of these physics events are of value to physicists, a number of *filters* are used to examine the events to determine which should be kept. Due to the nature of the environment, a number of these physics events may become corrupted as they flow through the system. This type of data corruption is classified as a temporal fault; the system should detect these corrupted physics events and ensure that the number of subsequently corrupted events is reduced. This is done by reducing the intensity (also referred to as the *luminosity*) at which the accelerator operates, thereby reducing the number of physics events produced by the experiment.

The architecture used in this case study is illustrated in Figure 13. The sub-components of the reflex and healing architecture have been arranged in a hierarchical fashion. The communication between the local managers, filters, data source and the regional manager is strictly event-based. For the sake of brevity, we have summarized all these events in Table 1. We will now detail the models of each of these components.

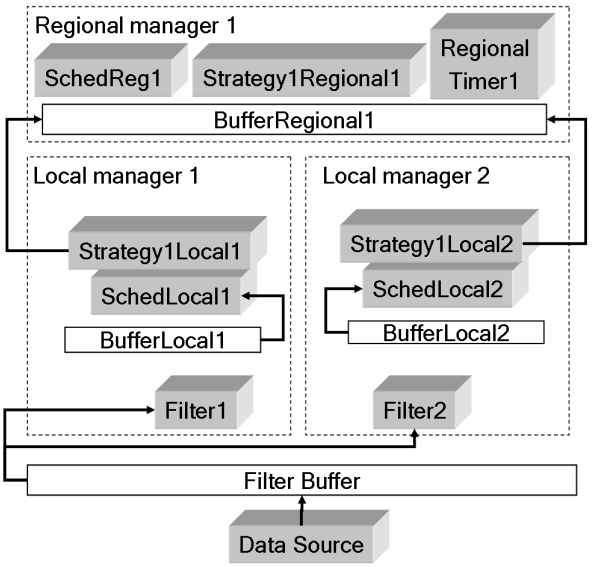

Fig. 13 The architectural setup for the case study. There are two local managers, one data source, two filter applications and one regional manager.

7.1.1 DataSource

A data source is used to mimic the physical data production mechanism of a high-energy physics experiment by generating simulated data for all the particle collisions that would normally happen in the particle accelerator. These data are referred to as *physics events*. The data source is true to the behavior of the actual system in that sets of physics events are generated at a fixed periodic rate, with a variable number of events per set. A control

Table 1 The detailed list of all the events being used in the case study.

Event Name	Relevance
GoodData	The Data source produced a good physics event.
BadData	The Data source produced a bad/ corrupt physics event.
GetData	One of the filters is dequeuing the event from filter buffer.
F1BadData	Filter1 has detected a bad physics event.
F2BadData	Filter2 has detected a bad physics event.
BufSchedLoc1	Scheduler of local manager 1 is dequeuing the local buffer.
StaStra1Loc1	Instruction from local scheduler 1 to start the first local strategy.
StopStra1Loc1	The local strategy 1 is signaling its completion to its scheduler.
L1R1BadData	Signal from local manager 1 to its regional about a detection of bad data by its Filter.
BufSchedLoc2	Scheduler of local manager 2 is dequeuing the local buffer
StaStra1Loc2	Instruction from local scheduler 2 to start the first local strategy.
StopStra1Loc2	The local strategy 2 is signaling its completion to its scheduler.
L2R1BadData	Signal from local manager 2 to its regional about a detection of bad data by its Filter
BufSchedReg1	Regional Scheduler is dequeuing the regional buffer.
StaStra1Reg1	Instruction from regional scheduler to start the first regional strategy
StopStra1Reg1	Signal from regional strategy to the scheduler that it is relinquishing the thread.
StaReg1Timer1	Signal from regional strategy to the timer to start measuring time.
StopReg1Timer1	Signal to stop the timer, issued either by the timer itself after the set time has elapsed or by the regional strategy if it not longer needs to measure time.
Reg1DataSource	A mitigation instruction, in response of the fault condition, from Regional manager to the data source to change its behavior.

parameter of the data source is exposed which allows the number of physics events generated per unit time to be varied.

Figure 14 is the timed automaton model constructed in UPPAAL for the data source. Its initial state is *Init*. In its normal model of operation, after every four time units - measured by using the clock x - a variable number of physics events are generated. There are two categories of these physics events, good data and bad data (see Table 1).

In case a temporal fault is detected such that the number of bad data events generated has to be reduced, the data source can be instructed by using the event *Reg*1*DataSource* to change its luminosity and move to the *SecondInit* location.

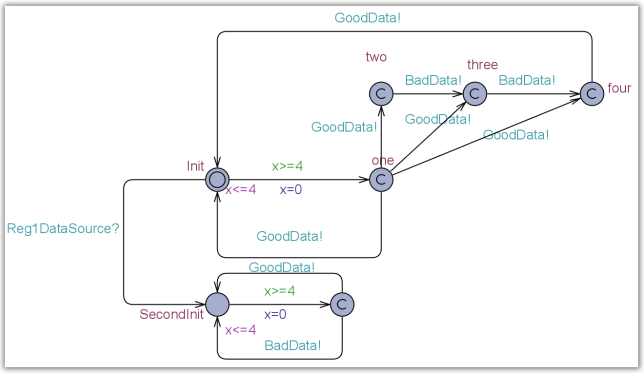

Fig. 14 The data source generates data every four time periods. The number of data events generated varies over periods.

7.1.2 FilterBuffer

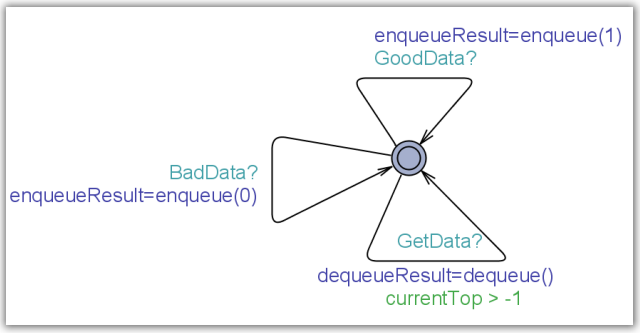

Fig. 15 The filter buffer stores the generated data source events.

A buffer is required to store the generated physics event before they can be processed by a filter application. As shown in Figure 15 the buffer uses two methods called enqueue and dequeue to push and pop physics events into the queue. The maximum size of the queue is modeled by a fix integer constant (not shown in the figure). The filter applications uses the get data event to access the top of the queue.

It is evident that the overflowing of the queue is undesirable as it might lead to a loss of important physics events. Therefore, one of the key properties that we will analyze in this case study is to check if the buffer queue overflows. This will be done by checking if the boolean variable *enqueueResult* becomes false.

*7.1.3 Filter Applications: Filter*1

The filter applications, filter1 and filter2, work concurrently to process the physics events present in the filter buffer. Figure 16 illustrates the timed automaton model for the first filter. After obtaining the data from the filter buffer, a filter stores the physics event into a temporary

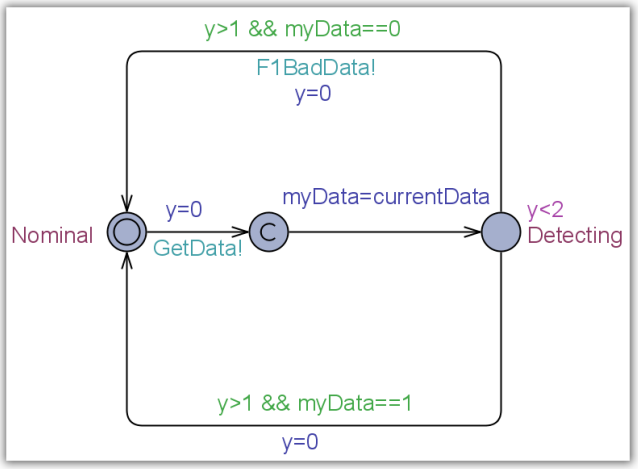

Fig. 16 There are two filters in this setup. Both filters consume the data from the filter buffer and detect whether it is a good physics event or a bad physics event.

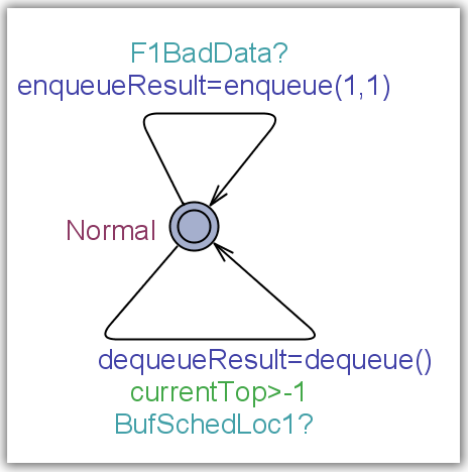

Fig. 17 The buffer of one of the local managers.

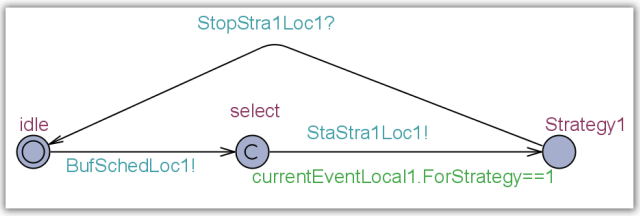

Fig. 18 The local scheduler dequeues the local buffer and starts the appropriate strategy. This scheduler only has one thread. Therefore, only one strategy can be executed at a time.

local variable and then executes the detection algorithm. It has been assumed in the model that the filter takes between $[1, 2]$ time units in order to detect if the physics event is good or bad. If the physics event is bad, the filter generates an event, $F1BadData$, to the buffer of its local manager, and resumes work with the next physics event.

Filter2 is identical in its model to filter1 except that it generates ,$F2BadData$, and sends it to its local manager, which is different from the first filter's local manager. Together, in the worst case, the filters can process 2 data events every 4 time units.

7.1.4 Local Managers: Local Manager 1

Recall that a local manager has a scheduler, a buffer, and one or more strategies. This case study considers two local managers. In this section, we will describe the first local manager's components. The second local manager is identical to the first with the exception that it has its own corresponding events as described in the Table 1.

*7.1.5 Local Manager's Buffer: BufferLocal*1

The local manager's buffer is used to store the events, which are subscribed to by the manager's strategy. The model of a local manager's buffer, illustrated in Figure 17, is similar to that of the filter buffer, with an enqueue function call for each event that is being added. The arguments passed to this function are the ID of the event and number of strategy, which has subscribed to the event. This buffer's communicates with the scheduler, which pops the queue and executes the corresponding strategy.

*7.1.6 Local Scheduler: SchedLocal*1

In this case study, the scheduler only has one thread on which it can execute a strategy. Figure 18 illustrates the scheduler model. It uses the $BufSchedLoc1$ event to obtain the current event from the queue and then starts the strategy - in this case there is only one possible strategy. It returns to its idle state in which the strategy generates the $StopStra1Loc1$ event, signaling that the strategy has finished its execution and has relinquished the thread.

*7.1.7 Local Strategy: Strategy*1*Local*1

Temporal data corruption faults can arguably be mitigated at the local level as well as the regional level. However, we will show in the analysis that if one attempts to detect and mitigate a temporal fault at a local level there is a chance of missing a subsequent temporal fault. It is due to this reason that we chose to implement the mitigation strategy at the regional level. Therefore, the local strategy has been reduced to simply notify its regional manager whenever its filter reports a bad physics event. Fig 19 illustrates this local strategy for the first local manager. The model captures the possibility of a worst-case delay of one time unit at the local strategy

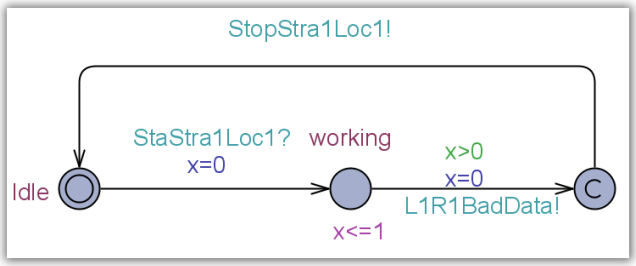

Fig. 19 The local strategy. The local scheduler starts this strategy.

level before it can generate the $L1R1BadData$ event to notify the regional manager.

Additional local strategies can be added by including an additional start strategy state to the local scheduler.

7.1.8 Regional Manager: Regional Manager 1

The components of a regional manager are identical to a local manger, the only difference being its hierarchical position. We will now describe these components.

*7.1.9 Regional Buffer: BufferRegional*1

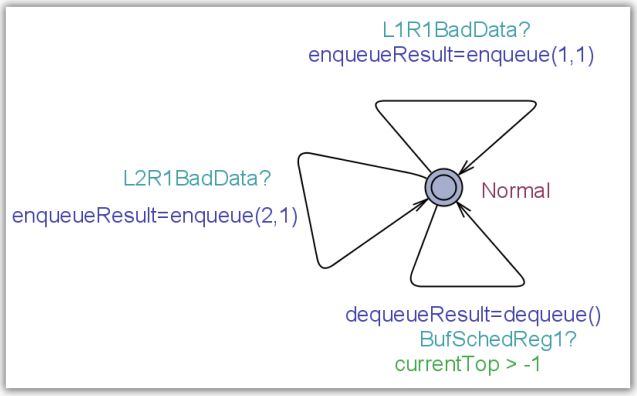

Fig. 20 The regional buffer stores the events communicated by its local managers.

Figure 20 illustrates the model for the regional buffer. It stores the events communicated to it by the local managers. In this case these events are $L1R1BadData$ and $L2R2BadData$. The former event signifies the detection of a bad data by the first local manager, while the latter is the signal from the second local manager.

*7.1.10 Regional Scheduler: SchedRegional*1

The regional scheduler, shown in Figure21, is identical in its operation to the local schedulers. It has one thread at its disposal, which it uses to execute a strategy depending upon the event popped from the regional buffer.

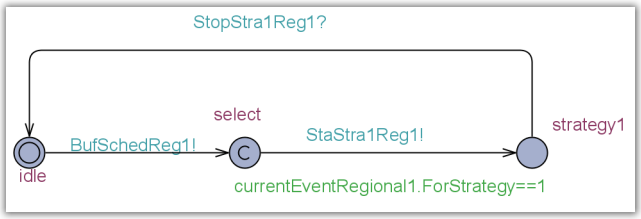

Fig. 21 The regional scheduler also works like the local scheduler. It consumes events from the regional buffer and starts the appropriate regional strategy.

*7.1.11 Regional Strategy :Strategy*1*Regional*1

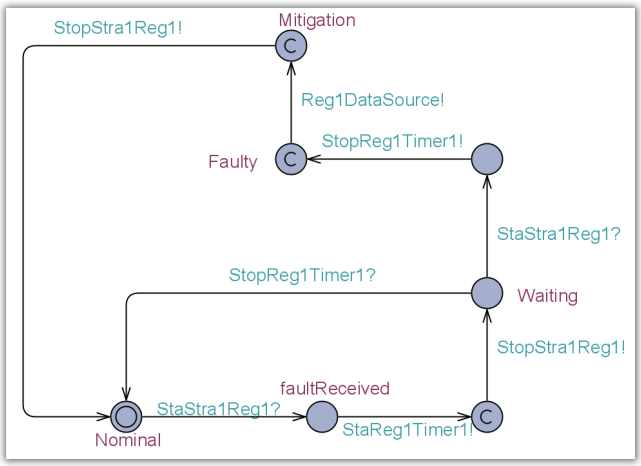

Fig. 22 This regional strategy issues a mitigating action if contiguous bad data events arrive before the timer event arrives. This mitigating action changes the parameters of the data source effectively reducing the number of physics events generated per cycle.

The regional strategy, shown in Figure 22, detects a region wide fault and then issues the mitigating action as an event, $Reg1DataSource$. The fault is assumed to have occurred if the regional manager receives two bad data events in any particular order from its local managers within a period of two time units.

As shown in the fig 22, this strategy starts its operation in the nominal state. When the scheduler wakes the strategy, it moves to the fault received state and starts a timer to measure two time units. After starting the timer, the strategy moves to a waiting state and relinquishes its thread back to the scheduler. It the timer sends a $StopReg1Timer1$ event before the next start event from the scheduler, the strategy resets and moves back to the nominal mode of operation. However, if a second start event from scheduler is received before the timer's event, the strategy moves to a faulty state. This is because two bad physics events have been detected within two time units of each other. Then the strategy issues a mitigation action using the $Reg1DataSource$ event to tell the data

source to reduce its luminosity and hence reducing the number of bad data being generated.

*7.1.12 Regional Timer: RegionalTimer*1

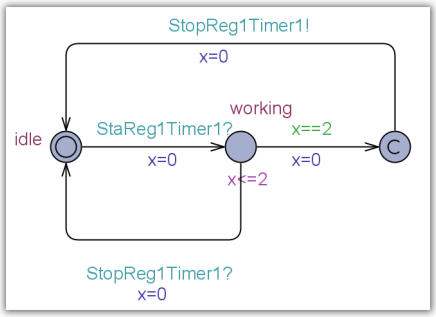

Fig. 23 This timer is used by the regional strategy as a stopwatch set for a period of 2s to measure time elapsed between two bad data. It is necessary to use the timer because the regional strategy relinquishes the thread between the two contiguous arrivals of bad data.

Fig 23 describes the model of the non-preemptive timer used in this case study. Once started by the regional strategy, this timer generates a $StopReg1Timer1$ after two time units, unless it is stopped.

7.2 Simulation

Figure 24 details a simulation trace as obtained from UPPAAL for the temporal data stream corruption fault and the corresponding mitigation. In the figure, the time line flows from the top to bottom.

The trace starts with the production of four physics events in the sequence, *GoodData*, *BadData*, *BadData*, and *GoodData*. These events are then enqueued into the filter buffer. Then filter1 synchronizes with the filter buffer by using the *GetData* event - in this case *GoodData*- and finally transitions to the *Detecting* location. After a brief delay, filter2 also pops an event from the filter buffer - *BadData* and moves to the *Detecting* location. Once it completes the detection process, filter2 informs its local manager about the bad physics event by using an $F2BadData$ event and then it gets another physics event. Since this next physics event is also a corrupt, it again detects it and raises another $F2BadData$ event. Simultaneously, filter1 detects that its physics event was good and transitions to $Nominal$ location and gets another physics event.

Once the second local buffer becomes nonempty, its scheduler wakes up the local strategy, which raises the $L2R1BadData$ event and enqueues it into the regional buffer. At this time, the regional scheduler dequeues its buffer. It then starts the regional strategy. The regional strategy moves to the $faultReceived$ location and starts the regional timer and proceeds to the *waiting* location. When the second $F2BadData$ event reaches the local buffer, the local manager again raises an $L2R1BadData$ event, which causes the regional scheduler to start the regional strategy again. At this time the regional strategy is in the $Waiting$ location and the regional timer has not sent the $StopReg1Timer1$ event, yet. Thus, the regional strategy moves to the $Faulty$ location. Then the strategy moves to the $Mitigation$ location, issues a $Reg1DataSource$ event and causes the data source to change its luminosity and hence reduces the number of contiguous bad data events.

7.3 Analysis and Verification

In the subsequent subsections, we will present the properties, which we verified for this case study. For each property, we will provide the following details:

* *Specification:* The formal specification of the property in TCTL.
* *Description:* A textual explanation of the property.
* *Result:* The result of the verification process as obtained with UPPAAL on a $3GHz$ computer with $1024MB$ RAM running Linux.
* *Computation Time:* Time required to compute the results.
* *Analysis:* A textual explanation if the property is not satisfied.
* *Correction of Design:* If applicable, we provide a corrected design that will satisfy the concerned property.

7.3.1 Feasibility Analysis: Why not a Local Temporal Fault Strategy?

One might argue why we need to implement a regional strategy to mitigate the data stream corruption. Why not use two local strategies, one for each local manager instead?

As an experiment, we demoted the current regional strategy and replaced the two local strategies with this regional strategy. We then checked the setup against the following property:

Specification:
A□(DataSource.two→A◇(Strategy1Local1.Faulty || Strategy1Local2.Faulty))

Description:
The contiguous bad data fault occurs two *BadData* events are pushed next to each other in the filter buffer. This only happens when the Data Source is in its location “two” (see Sect. 7.1.1). If we consider that the current local strategies are replaced by the current regional strategy (see Sects. 7.1.7 and 7.1.11), then we can say that the fault will be detected when either of the local manager is in its fault state. The formal specification mentioned above checks for this fact.

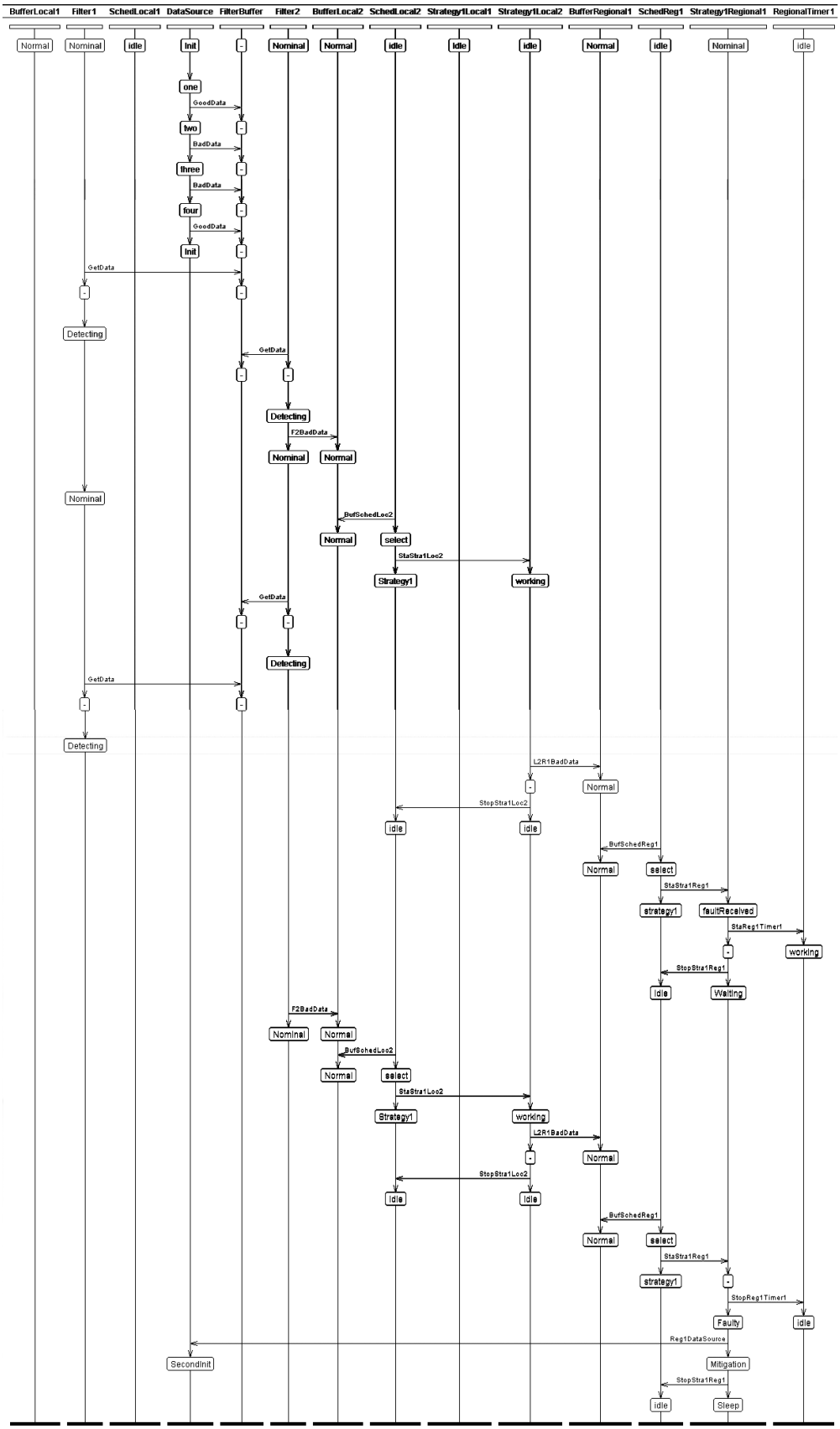

Fig. 24 A simulation of a temporal fault and its mitigation is produced by UPPAAL.

Result:
The result of verification stated that that this property is not universally true.
Computation Time: 2 seconds.
Analysis:
The reason for this is because there can be a case when two contiguous faults from the filter buffer can be picked by different filters and hence can remain undetected by their local manager.

Therefore, we decided to promote this strategy as a regional strategy.

7.3.2 Regional Strategy Will Always Catch Data Fault

This analysis furthers the property check mentioned in the previous section. The architectural setup detailed previously describes the regional strategy. In order to check if this strategy will always work, we checked the following property:

Specification:
A□DataSource.two→A◇ Strategy1Regional1.Faulty
Description:
This specification is similar to the one mentioned in Sect. 7.3.1. It should be noted that the verification of this property would not have been possible without a timed automaton model. This is because the definition of the fault requires a measure of time.
Result:
This property was universally true.
Computation Time: 2 seconds.
Analysis:
All contiguous bad physics events will be mitigated.

7.3.3 Liveness: Deadlock check

It is important to check that the designed system will not end in deadlock. For this purpose, we verified the model against the following property.

Specification:
A□ not deadlock
Description:
The system will never deadlock.
Result:
The result of this check was false.
Computation Time: $3-4$ seconds.
Analysis:
The UPPAAL verification engine generates a counter example in the case where a property is deemed false. Upon reviewing the counter example, we found the problem. Currently, the data source can generate two pairs of contiguous bad physics events before the first one is detected by the regional strategy, which then issues a $Reg1DataSource$ event to lower the luminosity of the data source. The deadlock happens when the regional manager detects the second pair of contiguous bad data events and it again tries to synchronize with the data source by using the event $Reg1DataSource$. However, the data source, see Sect. 7.1.1, at this time is in the $SeondInit$ location and cannot synchronize the event from regional manager.
Correction of Design:
To correct the deadlock, we modified the regional strategy to ignore the second bad data. The new strategy is shown in Figure 25. Although this is modification is not complex, it still underscores the benefit that the verification provides in correcting bad mitigation strategies.

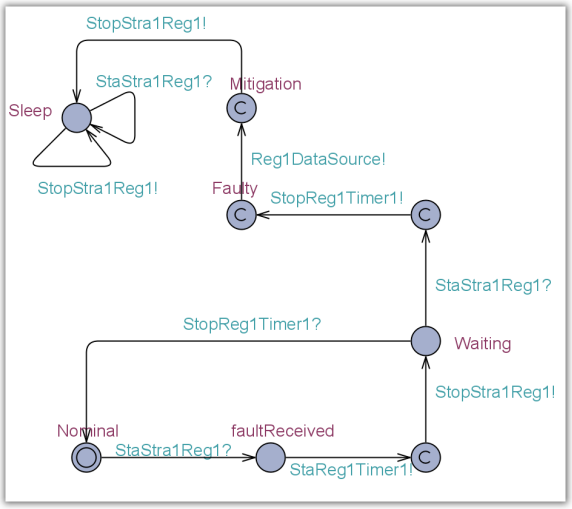

Fig. 25 The regional strategy ignores the second false alarm by transitioning to sleep state.

7.3.4 Safety: Buffer Queue Will Not Overflow

In Sect. 7.1.2 we had mentioned that it is imperative that the filter buffer never overflows. This is necessary to ensure that generated physics events are processed and not lost. To ensure that this never happens we have to verify the following property:

Specification:
A□ $FilterBuffer.enqueueResult$
Description:
In the filter buffer (Sect.7.1.2), the enqueue method call is used to push the data event into the queue. This method returns a boolean value stored in the variable $enqueueResult$, which is set to true if the action was successful. In the event of a buffer overflow, the return value is set to false. Hence, it is sufficient in the formal specification to ensure that $enqueueResult$ is always true.
Result:
This property is satisfied.

Computation Time: 4 seconds.
Analysis:
The buffer will never overflow.

7.4 Evaluation

In this case study, we verified a real-time reflex and healing setup with two local managers, and one regional manager. Overall, there were 14 timed automaton models in the network (see Figure 13), with five clock variables, one for the data source, two for the two filters, two for the two local strategies and one for the regional timer. With this model, the property verification took 3s, on an average.

It has been known that the model-checking tools for timed automaton suffer from the state-explosion problem. Specifically, it has been experienced that the performance of the verification algorithm degrades as the size of the timed automaton network increases. With the increasing number of clocks, the memory required to be able to explore all possible regions of the region automaton becomes extremely large. Therefore, a concern is how will we be able to scale our verification technique to systems with hundreds or even thousands of processors.

It is evident that we cannot verify the whole system model with several processors together. However, due to our architecture, which forbids any communication between peer managers and only allows communication between a parent and child node (refer to Sect. 3.1), we can abstract the system at any particular node by only using its parent node and children node. It is our contention that by using a bottom to top approach and performing iterative verification checks for each level of hierarchy we can guarantee that the full system model will also satisfy the property. This is the basis for our future work.

8 Conclusion and Future Works

In this paper, we have shown that the real-time reflex and healing framework is an autonomic, fault mitigation framework, which with the semantics of networked timed automation can verify real-time properties of the system. In particular, we have shown this to be useful in determining the liveliness and safety of the system, as well as the time-boundedness of the system's mitigation responses. One can further conclude that in the area of large scale computing systems, model-based analyses such as these are necessary to ensure that non-expert users do not introduce into system behaviors, which violate these properties.

As future work, we are investigating the possibility of using a discrete time model, which will help in reducing the computational complexity of verifying increasingly larger systems. This approach may lead to automated synthesis of new mitigation behaviors by considering the fault mitigation as a discreet timed supervisory control problem. This would further serve to provide self-managing systems, which are more robust, yet more easily augmented with new behavior.

Acknowledgements 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. Aghasaryan A, Fabre E, Benveniste A, Boubour R, Jard C (1998) Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets. Discrete Event Dynamic Systems, Volume 8, Issue 2 pp 203–231
2. Ahuja S, Bapty T, Cheung H, Haney M, Kalbarczyk Z, Khanna A, Kowalkowski J, Messie D, Mosse D, Neema S, Nordstrom S, Oh J, Sheldon P, Shetty S, Wang L, Yao D (2005) RTES demo system2004. SIGBED Rev 2(3):1–6
3. Alur R, Dill DL (1994) A theory of timed automata. Theoretical Computer Science 126(2):183–235
4. Behrmann G, David A, Larsen KG (2004) A tutorial on uppaal. In: SFM, pp 200–236
5. Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W (1996) Uppaala tool suite for automatic verification of real-time systems. In: Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control, Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp 232–243
6. Buttazzo GC (2005) Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Norwell, MA, USA
7. Cassandras CG, Lafortune S (1999) Introduction to Discrete Event Systems. Kluwer Academic Publishers, Norwell, MA, USA
8. Clarke EM, Grumberg O, Peled DA (2000) Model Checking. MIT Press, Cambridge MA, USA
9. Dubey A, Nordstrom S, Keskinpala T, Neema S, Bapty T (2006) Verifying autonomic fault mitigation strategies in large scale real-time systems. ease 0:129–140
10. Frank PM (1990) Fault diagnosis in dynamic systems using analytical and knowledge-based redundancya survey and some new results. Automatica 26(3):459–474
11. Garlan D, Cheng SW, Schmerl B (2003) Increasing system dependability through architecture-based self-repair. Architecting Dependable Systems
12. Gertler J (1998) Fault Detection and Diagnosis in Engineering Systems. Marcel Dekker
13. Gutleber J, et al (2001) Clustered data acquisition for the CMS experiment. In: International Conference on Computing In High Energy and Nuclear Physics
14. Haney M, Ahuja S, Bapty T, Cheung H, Kalbarczyk Z, Khanna A, Kowalkowski J, Messie D, Mosse D, Neema S, Nordstrom S, Oh J, Sheldon P, Shetty S, Volper D, Wang L, Yao D (2005) The RTES project - btev, and beyond. Real Time Conference, 2005 14th IEEE-NPSS pp 143–146
15. Henzinger T, Nicollin X, Sifakis J, Yovine S (1994) Symbolic model checking for real time systems. Information and Computation 111(2):193–244
16. Huth M, Ryan M (2000) Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University press
17. de Kleer J, Williams BC (1987) Diagnosing multiple faults. Artificial Intelligence 32(1):97–130

18. Krcál P, Yi W (2004) Decidable and undecidable problems in schedulability analysis using timed automata. In: TACAS, pp 236–250
19. Kwan S (2002) The btev pixel detector and trigger system. In: FERMILAB-Conf-02/313
20. Lamperti G, Zanella M (2002) Diagnosis of discrete event systems from uncertain temporal observations. Artificial Intelligence 137(1-2):91–163
21. Lerner U, Parr R, Koller D, Biswas G (2000) Bayesian fault detection and diagnosis in dynamic systems. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, The MIT Press, pp 531–537
22. Lunze J (2000) Diagnosis of quantized systems based on a timed discrete-event model. IEEE Transactions on Systems, Man, and Cybernetics, Part A 30(3):322–335
23. Madl G, Abdelwahed S, Karsai G (2004) Automatic verification of component-based real-time corba applications. In: RTSS, pp 231–240
24. Nordstrom S, Bapty T, Neema S, Dubey A, Keskinpala T (2006) A guided explorative approach for autonomic healing of model based systems. In: Second IEEE conference on Space Mission Challenges for Information Technology (SMC-IT)
25. Nordstrom S, Dubey A, Keskinpala T, Bapty T, Neema S (2006) Ghost: Guided healing and optimization search technique for healing large-scale embedded systems. In: Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06)
26. Nordstrom S, Shetty S, Neema SK, Bapty TA (2006) Modeling reflex-healing autonomy for large scale embedded systems. Systems, Man and Cybernetics, Part C, IEEE Transactions on 36(3):292–303
27. Parashar M, Hariri S (2004) Autonomic computing: An overview. In: UPP, pp 257–269
28. Patton RJ, Frank PM, Clarke RN (eds) (1989) Fault diagnosis in dynamic systems: theory and application. Prentice-Hall, Inc., Upper Saddle River, NJ, USA
29. Rafea AA, Desouki AE, El-Moniem S (1990) Combined model expert system for electronics fault diagnosis. In: IEA/AIE '90: Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems, ACM Press, New York, NY, USA, pp 23–31, DOI http://doi.acm.org/10.1145/98784.98793
30. Ramadge P, Wonham W (1987) Supervisory control of a class of discrete event processes. Siam J Control and Optimization 25(1)
31. Resnick M (1999) Decentralized modeling and decentralized thinking. Modeling and Simulation in Science and Mathematics Education pp 114–137
32. Rothenberg J (1989) The nature of modeling. Artificial intelligence, simulation & modeling pp 75–92
33. Sampath M, Sengupta R, Lafortune S, Sinnamohideen K, Teneketzis D (1996) Failure diagnosis using discrete-event models. IEEE Transactions On Control System Technology 4(2):105–124
34. Shetty S, Nordstrom S, Ahuja S, Yao D, Bapty T, Neema S (2005) Systems integration of large scale autonomic systems using multiple domain specific modeling languages. In: ECBS, pp 481–489
35. Sterritt R (2005) Autonomic computing. Innovations in Systems and Software Engineering 1(1):79–88
36. Sterritt R, Hinchey MG (2005) Autonomic computing - panacea or poppycock? In: ECBS, pp 535–539
37. Truszkowski WF, Hinchey MG, Rash JL, Rouff CA (2006) Autonomous and autonomic systems: a paradigm for future space exploration missions. Systems, Man and Cybernetics, Part C, IEEE Transactions on 36(3):279–291
38. Vries RD (1990) An automated methodology for generating a fault tree. IEEE Transactions On Reliability 39:76–86
39. Yao D, Neema S, Nordstrom S, Shetty S, Ahuja S, Bapty T (2005) 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
40. Yovine S (1997) Kronos: A verification tool for real-time systems. International Journal on Software Tools for Technology Transfer 126:110–122