Runtime Verification of Timing Constraints

David Urting, Yolande Berbers

Report CW 345, July 2002

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Runtime Verification of Timing Constraints

David Urting, Yolande Berbers

Report CW 345, July 2002

Department of Computer Science, K.U.Leuven

Abstract

The necessity for reusing software components in embedded systems becomes significant due to the ever-increasing software complexity, product diversification and market pressure. In order to facilitate the reuse of components we have developed the CCOM (Component and Contract Oriented Modeling) language for the specification and the composition of embedded components. Non-functional constraints, such as timing and memory constraints, can be specified by means of contracts that are imposed on the CCOM models. This report describes the design of a runtime monitoring system that offers verification mechanisms for the dynamic validation of timing contracts. We first give an overview of existing monitoring mechanisms and discuss important issues related to the monitoring of timing constraints in embedded applications. In the second part of the report a detailed description of the CCOM monitoring system is given.

Keywords : timing constraints, contracts, components, monitoring, embedded systems.

1. Introduction

The necessity for reusing software components in embedded systems becomes significant due to the ever-increasing software complexity, product diversification and market pressure. In order to facilitate the reuse of components we have developed the CCOM (Component and Contract Oriented Modeling) language [1] – and supporting tool – for the specification and the composition of embedded components. Non-functional – like timing and memory – constraints can be specified by means of contracts that are imposed on the CCOM models.

This report describes the design of a runtime monitoring system that offers verification mechanisms for the validation of timing contracts. This monitoring system has been implemented as part of the *SEESCOA*[1] *component system*, which is an execution platform for the CCOM components.

Runtime monitoring consists mainly of two parts: *information gathering* and *information processing*. We apply monitoring techniques to retrieve information about messages exchanged between components. Afterwards, this information is used to validate contracts imposed on the system. This document will mainly focus on *timing* contracts, which impose certain timing and performance constraints on the system.

Section 2 elaborates on the theory of *runtime monitoring*. Runtime monitoring concepts and terminology are quite common, and these will be discussed in detail. In section 3, we will look at the runtime monitoring of timing constraints, with emphasis on the monitoring of RTL (Real-time Logic) constraints. Section 4 gives an overview of the contract monitoring system that has been developed for the SEESCOA project. A conclusion is given in section 5.

2. Monitoring

This section introduces the field of monitoring, and discusses some concepts and issues related to monitoring. The information contained here is important since it defines a common terminology that will be used in the remainder of the document.

2.1. What is Monitoring?

The first question one could ask himself is of course: what is monitoring? Different people have different views on this, but there is at least some common understanding on the topic. If one looks at the definition of *monitoring* given in a dictionary ([1]), following definitions can be found:

---

[1] This report is a result of the STWW-IWT SEESCOA (Software Engineering for Embedded Systems using a Component Oriented Approach) project. For more information, see http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/

Monitoring:

> *To check the quality or content of (an electronic audio or visual signal) by means of a receiver,*
>
> *To check by means of an electronic receiver for significant content, such as military, political, or illegal activity,*
>
> *To keep track of systematically with a view to collecting information,*
>
> *To test or sample, especially on a regular or ongoing basis,*
>
> *To keep close watch over; supervise,*
>
> *To direct, control.*

Two common aspects can be seen in these definitions: monitoring is about **collecting and analyzing** information and monitoring is about **supervision and control**. However, most people think about the first aspect when talking about monitoring.

The definition can also be applied to the *monitoring of software systems* (which is actually the topic we are interested in): monitoring a software system is the activity of *observing* the system's properties and activities, *analyzing* the information and eventually *controlling* (part of) the system based on the collected information.

Several examples of software monitoring can be given. Debugging, for example, is the monitoring of control and data flow to remove bugs or errors from a system. A second example is the monitoring of Quality-of-Service in multimedia systems; observing the properties of a video stream and maintaining these properties are well-known uses of monitoring.

2.2. Monitoring: a classification

There exist different types and uses of software monitoring. Therefore it is a good idea to present a classification based on the particular functionality for which monitoring is used. The classification was described in [10]:

- **Dependability**: fault tolerance and safety monitoring,
  - Examples: server status monitoring, watchdog-like monitoring,
- **Performance enhancement**: dynamic system configuration, dynamic program tuning and on-line steering,
  - Examples: object/component migration if the load exceeds a particular limit, Java HotSpot VM that monitors program execution and compiles heavily used functionality to native code,

- **Correctness checking**: check the consistency of an application with its formal specification to detect runtime errors or to verify if the specification and the implementation match,
    - Examples: pre- and postcondition checking in Eiffel, assertion checking, timing constraints monitoring,
- **Security**: detecting illegal or unwanted actions that could break security rules
    - Examples: login monitoring, Java Applet Security Manager which monitors and controls the activities of an applet,
- **Control**: the monitoring subsystem is an integral part of the entire system,
    - Examples: multimedia systems that analyze their own state and behaviour, and take actions to optimize their own execution (allocation of more cache, modifying internal priorities, …),
- **Debugging and testing**: collecting and analyzing information about data and control flow,
    - Examples: source-level debugger,
- **Performance evaluation**: collecting and analyzing information about the system's performance,
    - Examples: embedded profiling code, Java Virtual Machine Profiling Interface (JVMPI).

In [10] the concept *online monitoring system* was also defined. An online monitoring system is a system in which

- The monitor functions as an **external observer** of the software system. It has no control over it (so it misses the *Control* functionality described above),
- The monitor functions **during** the execution of the software system. As such, the monitoring system has to function in a timely matter to process and respond to events occurring in the system,
- The monitor is a **permanent** part of the system, although it may run at reduced functionality.

Based on the classification and the definition of an online monitoring system, we can ask ourselves what kind of system is a timing constraint monitoring system? Firstly, the timing constraint monitor will mostly be used as an external observer of the application. However, it should be possible to integrate the timing constraint monitor into the application as a component as such that other components can get information about possible timing violations and act accordingly. Secondly, the timing constraint monitor must be able to perform while the monitored system is running, as such that other components can be notified of violations in real time. But it must also be possible to log

timing violations for later analysis. Finally, the timing constraint monitor has to be a permanent part of the system (this will become clear in the remainder of the document).

As such, we can say that the timing constraint monitor is an online monitoring system, but with some characteristics relaxed, namely: the monitor can (but is not obliged to) act as external observer and analysis of the timing behaviour can also occur afterwards.

2.3. Concepts

When talking about monitoring, some concepts are used frequently. In this part, the important concepts are explained. This forms the basis of a terminology that will be used throughout the report. For more information on these concepts, we refer to [6], [10] and [12].

2.3.1. Monitoring system

The *monitoring system* is the set of all modules and activities necessary for monitoring a particular system. The monitoring system is not a monolithic block, but instead a set of related services. In general, we can distinguish between the part that is responsible for *gathering or recording* monitoring data and the part that is *analyzing* the monitored data.

2.3.2. Sensor

A *sensor* is an entity (software or hardware) that observes a small part of the system (it is responsible for gathering monitoring data). In fact, it observes a particular item (condition, action or property) until the item changes (condition is reached, the action occurs or the property is changed). It is then said that the sensor is *triggered*. When a sensor is triggered it generates an *event*.

The triggering of a sensor can occur immediately when the change to the item occurs (synchronous trigger). This is called a *tracing sensor*, since it 'traces' the item for changes.

The triggering of a sensor can also occur by a request of the monitoring system. It is then said that the monitoring system is *sampling* the sensor. Sampling a sensor is an asynchronous activity (the sensor triggers asynchronously).

The kind of needed sensor triggering (tracing or sampling) of course depends on the situation. If it is important to collect every item's change, then tracing is recommended. If it is allowed to miss some item's changes, then sampling is a good candidate. It is of course also possible to use sampling and record every item's change, but then the sampling rate has to be fast enough to catch all changes. In fact, the sampling rate should be at least higher than the fasted expected change rate of the item.

2.3.3. Event

Most monitoring systems are event-based. This means that the *unit of observation* is an event that occurs at a particular moment during the execution. Events are generated by sensors. If no sensors are present, then no events will be generated.

As such, a distinction has to be made between *non-monitored* events (these are events that occur in a system, but which are not reported to the monitoring system) and *monitored*[2] events (events generated by sensors and analyzed by the monitoring system). It is clear that every software system consists of millions and millions low-level events, like task switches, function calls, I/O requests and so on. Some monitors are able to monitor on this level, and they are often used for debugging purposes. The problem with these monitors is the huge amount of data that is analyzed or recorded during the execution of an application and the difficulty to extract meaningful information from this data[3].

There exist different types of events, but they can be categorized in three categories:

- **Hardware-level events**: this type of events indicates the occurrence of low-level hardware activities,
    - Examples: page faults, cache misses, interrupts
- **Process-level events**: events of this type report activities external to a process. Process-level events are often reported by sensors built into the operating or runtime system,
    - Examples: inter-process communication, file I/O, memory allocation,
- **Application-level events**: this type of events indicates the occurrence of activities internal to a process. This ranges from basic events (method calling, variable assignment, and so on) to more application-specific events (defined by the application programmer).
    - Examples: method execution, variable changes, object creation.

2.3.4. Action

An *action* is the response of the monitoring system to an event or a set of events. It is in fact a kind of handler within the monitoring system that is triggered when a relevant event occurs. Possible actions are: logging or recording a particular event/set of events, starting a particular process or activity to deal with the event(s), starting user or administrator interactions, and so on.

---

[2] In the rest of the report we will use 'event' to denote 'monitored events'.

[3] This report will elaborate on a monitor that only monitors high-level events (messages exchanged between components), with automatic verification of constraints imposed on the components.

The type of actions that a monitoring system can take is of course dependent on the situation.

2.3.5. Event history

The *event history* is the collection of events that have occurred in the system over a particular period of time. An event history is in fact an image of the history of a system. It is used by the monitoring system to look up events that have occurred. An event history item often stores a particular event identifier, a timestamp for this event and the particular occurrence of this event.

Depending on the type of events and application that is being monitored, the event history will need to be large or small. In some cases all events will need to be stored, while in other cases no events need to be stored. The size and type of event histories is thus dependent on the particular type of monitoring analysis that is performed by the monitoring system.

An event history does not need to be stored in RAM, it can also be stored in a file or database. However, in some cases, event history lookups have to be fast and as such these event histories have to remain in the internal memory.

We will elaborate more on the type of event histories that are needed when monitoring timing constraints.

2.3.6. Intrusion

A side effect of a lot of monitoring systems is of course the influence they can have on the execution of the application. Since monitoring systems often need to insert small pieces of code (sensors) and execute on the same node as the monitored applications, they snoop off resources used by the monitored application.

This type of influence is called intrusion (or also the probe or the Heisenberg effect) and can be unwanted in some situations. If one is monitoring a characteristic of a software system that is influenced by the monitoring system, then the measurements are not really reliable. Therefore, it is important to remove the intrusion effect or at least to take it into consideration.

We also need to make a distinction between the information gathering (retrieving information from sensors) and the processing of this information (during the analysis process). Sometimes, it is possible to remove the intrusiveness of the gathering process, and move the information processing to another node (with its own resources). In that case, the monitored system will not be influenced.

Most systems are however also intrusive during the gathering process (since they make use of software sensors). Systems that use hardware sensors are mostly non-intrusive.

2.3.7. Monitoring approaches

The discussion above about intrusiveness has led to some distinct monitoring approaches. Some monitoring approaches are *hardware* oriented. This means that the

information gathering (and eventually information processing) is done entirely in hardware. Other approaches are entirely in *software*, while others take a *hybrid* approach.

All approaches have their advantages and disadvantages:

- **Hardware approach**: no intrusion, inflexible, huge amount of low-level events (difficult to analyze), costly (to implement),
- **Software approach**: intrusive, flexible, abstract/user defined events, cheap,
- **Hybrid approach**: tries to combine the advantages of both approaches.

In the remainder of the report, we will focus on software monitoring. When discussing software monitoring, different techniques can be applied (mostly a combination of them):

- **operating system** or **runtime system support**,
    - Example: many OS'es have a particular interface for attaching a monitor or debugger to a process,
- **inlined monitoring statements** by the programmer or compiler,
    - Example: assertions added by the programmer,
- **parallel monitoring task**
    - Example: making use of the constructs offered by the OS a parallel task can analyze or log any information coming from a particular process,
- **monitoring node**
    - Example: making use of JVMPI a remote application can connect to a Java VM and retrieve information about the execution of the application running on the VM.

In the last approach (monitoring node), the information processing is done on a node with its own resources. This type of monitoring is the least intrusive[4].

2.3.8. Asynchronous versus synchronous monitoring

A monitoring system is said to be *asynchronous* if it analyzes the events asynchronously with the process that generates the events (the event processing is done in a separate task or node). A monitoring system is *synchronous* if the event analysis is

[4] If the network link to the monitoring node is also used by the monitored system itself, then of course there is intrusion of the monitor on the system!

done when the process generates the events (the event processing is done in the execution thread of the monitored process).

Asynchronous monitoring is less intrusive than synchronous monitoring, and in some cases this kind of monitoring is the only possible one. This is the case in systems where the *non-occurrence* of an event has to be processed. We will come back on this later when discussing timing constraint monitoring.

Note that synchronous monitoring needs synchronous sensor triggering (see section 2.3.2); asynchronous monitoring can be used with synchronous or asynchronous sensor triggering.

3. Monitoring Timing Constraints

This section discusses some aspects that come into play when monitoring timing constraints in embedded systems. First, the issues concerning embedded systems are described. This is followed by a section on timing constraint monitoring. We conclude in section three with some important thoughts one has to keep in mind when developing a timing constraint monitor for embedded systems.

3.1. Monitoring Embedded Systems

Monitoring embedded systems is of course different from monitoring standard desktop applications. Embedded systems have limited resources, which in turn makes it much more difficult to integrate a monitoring system. This section discusses some of the issues and pitfalls when considering monitoring functionality for these systems.

To describe the difficulties that arise when monitoring embedded systems, we have to make a distinction between *resource constrained*, *real-time* and *distributed* embedded systems. Each type of system has its own characteristics, and also its own consequences for supporting monitoring.

3.1.1. Resource constrained systems

A resource constrained system has limited amounts of memory, a relatively slow processor and/or restrictions on its energy consumption. Therefore, it is not easy to integrate full monitoring support in this type of systems. It is a good idea to implement monitoring support on a separate node for such systems. As such, only the sensors take up resources. The actual processing of the events generated by the sensors and the storage of an event history is done remotely on another node. A second problem with this kind of systems is the lack of useful interfaces (a lot of systems only have a serial port to interface with). This is rather unsuited when a lot of events are generated.

3.1.2. Real-time systems

A real-time system has strict deadlines for its tasks, and the impact of a runtime monitoring system has to be taken seriously. Firstly, the sensor (or the probe) takes some time to do its processing. If the probes are removed from the final system, it could imply

that some timings are not adhered to anymore. So, it is recommended to keep the probes within the system[5], even at deployment time.

The problems that arise when removing a probe can be twofold:

- A requested behaviour occurs with the probe in the system, but an unwanted behaviour occurs when it is removed,

- An unwanted behaviour occurs with the probe in the system, but the requested behaviour occurs when the probe is removed.

It is clear that both situations are to be avoided.

3.1.3. Distributed systems

Distributed systems have a lot of issues due to their nature. Distributed monitoring often occurs on a separate node that collects the events coming from various monitored nodes. Two important problems with these systems are the infamous *clock synchronization* and *delay* problems:

In a distributed system, clocks tend to differ from each other. In such cases it is difficult to determine which event occurred before another if both come from different nodes with different clocks. The problem can be partially solved by using some kind of clock synchronization algorithm to keep the differences between the clocks within a specified interval.

A second problem is the occurrence of delays in the exchange of messages. This problem is in fact quite analogous to the clock synchronization problem, since the monitoring node does not receive all events in the order in which they were sent.

Another issue concerning distributed systems is the network-intrusiveness of the monitoring system. The monitoring functionality is implemented by sending events to the monitoring node over a network that is also used by the monitored nodes.

3.2. Timing Constraints Specification and Monitoring

A lot of work has been done on the specification and static analysis of timing constraints in real-time systems. More information on this can be found in [8]. This work has led to a lot of different formalisms that can be categorized into four main categories: temporal logic, operational semantics, process algebras and assertion methods. Another categorization can be done on the basic concept used in the formalism: events or transitions. Most formalisms have some kind of event notion, while Petri Net based formalisms make use of transitions.

One of the 'temporal logic' event based formalisms is the Real-Time Logic language or RTL. In the following sections, we will give a short overview of the language, and

---

[5] In fact, it is a good idea to keep the probes in any system. Even if the system is not real-time, removing the probes could result in different behaviour (eg. race conditions).

discuss the application of the language to the monitoring of timing constraints in embedded systems.

3.2.1. Real-Time Logic (RTL)

Formalisms that belong to the temporal logics category are relatively easy to understand and make use of a solid logical base. Traditional temporal logic is however not powerful enough for expressing timing constraints.

RTL is a discrete, event based, *real-time* temporal logic, which is more powerful than traditional temporal logic:

- **Discrete**: time is represented by discrete numbers,
- **Event based**: events are the basic constructs of the formalism and are 'timestamped',
- **Real-time logic**: a temporal logic with an explicit clock representation. Traditional temporal logic only allows the specification of ordering between events, while real-time logic also introduces the notion of (absolute and relative) time.

RTL was first introduced in 1986 to specify and statically analyze timing specifications of real-time systems. To do this, one had to specify formally the (timing) behaviour of the system by means of RTL formulas. Afterwards, an RTL specification of the constraint had to be made. By means of a formal proof, one was then able to prove or disprove the constraint. Alas, it has been shown that this problem was NP complete. See [4] for more information on this.

3.2.2. RTL and runtime monitoring

As was said in the previous section, RTL was initially intended for static analysis of real-time systems. But in later work ([6], [7] and [12]), it has been shown that RTL could also be used for dynamic analysis of real-time systems.

The central idea to this approach is to evaluate the RTL constraints at runtime: events are recorded and timestamped by the underlying runtime system and these events are then used to validate the RTL constraints. The subsystem responsible for validating these constraints is called a *satisfiability checker*.

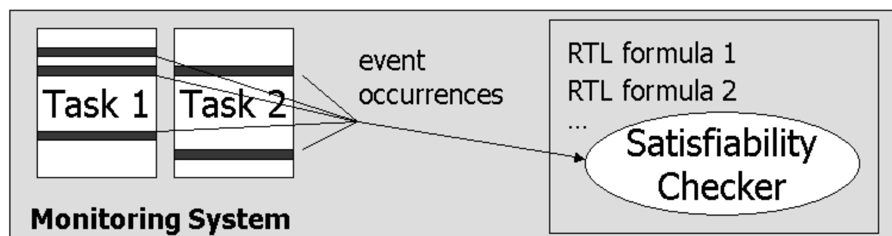


Figure 1: RTL monitoring system

To summarize, this type of monitoring system consists of three important parts:

- **Annotation formalism**: an RTL-like formalism for annotating timing constraints,
- **Probes**: the probes are responsible for intercepting events and timestamping them (these can be seen as a kind of sensors),
- **Satisfiability checker**: checks the RTL constraints making use of the events generated by the probes. The algorithm that is used by the checker is graph-based (see [12] for more information on this).

3.2.3. Constraint Specification

The initial RTL syntax was based on the occurrence relation `R(e, i, t)`. This relation indicates that the `i`th occurrence of event `e` occurs at time `t`. RTL formulas can be quite complex, making use of quantors, mathematical relations and algebraic expressions. This complexity makes it hard to formally proof RTL constraints.

To monitor RTL constraints at runtime, it was necessary to constrain the possible set of RTL formulas, since complex formulas tend to be difficult to monitor. Also, the occurrence relation `R` has been replaced by an occurrence function `@(e,i)` that returns[6] the occurrence time of the `i`th occurrence of event `e`. For example `@(start_button_pressed, 5)` returns the time at which the start button was pressed for the fifth time. To retrieve the occurrence time of the last `i`th occurrence of an event, `@(e,-i)` can be used. As an example, @(update, -2) returns the occurrence time of the 2nd last occurrence of the event "update" at a certain moment

Based on the occurrence function, one is now able to construct basic constraints by means of the ≤, + and - operators. A monitorable RTL formula looks like:

```
@(e1, i) <= @(e2, j) +/- C
```

Making use of these formulas, it becomes possible to formulate deadline and delay constraints:

- `@(ping, -2) ≤ @(ping, -1) – 10` is a delay constraint:
    - It indicates that there must be at least a period of 10 time units between successive occurrences of the 'ping' event,
- `@(alive, i) ≤ @(ping, i) + 20` is a deadline constraint:
    - It indicates that if the ith occurrence of the 'ping' event occurs, then the ith occurrence of the 'alive' event has to occur no later than 20 time units after the 'ping' event.

---

[6] In fact, the @-function accesses the event history to retrieve the time at which this event has occurred.

These constraints could for instance be used in a system where one component has to ping another component to determine if it is alive. Between 'ping' events there should be at least 10 time units, and when the *pinged* component receives a 'ping' event, it has to respond within 20 time units with an 'alive' event.

It is clear that the set of constraints that can be defined is rather small. In [6] the authors have expanded this set to more complex formulas, but the resulting expressiveness is still not very advanced.

3.2.4. Embedded versus Monitored Constraints

One of the questions that arises when monitoring RTL constraints (and time constraints in general) is where to put the constraint evaluation logic?

The first approach is to use *embedded constraints*. This means that the programmer can add RTL constraints to his source code (like assertions) and these assertions are evaluated when the code reaches that particular assertion. This kind of monitoring is also called synchronous monitoring as discussed in section 2.3.8. A disadvantage of synchronous monitoring is that the assertion has to be executed in order to be evaluated, and at that moment it could be too late. Also, depending on a program's logic, it could be that the assertion will never be executed (because the thread of execution never reaches the assertion), and as such, the constraint violation would go unnoticed!

The other approach is called *monitored constraints*. The monitoring is done in a task running in parallel with the monitored application. The monitored application only generates events and puts them in a queue. This queue is then emptied by the monitoring task, which puts the events in its event history. Afterwards, it processes the requests stored in the event history. This kind of monitoring is asynchronous, and enables the detection of constraint violations before the actual overdue events have occurred.

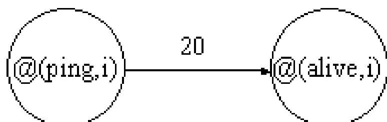
3.2.5. Checking Constraints at Runtime

We will not discuss in detail how the actual constraint evaluation is done, since this would take us too far. The main technique behind the runtime monitoring of these RTL constraints is graph based. A graph is constructed by means of the initial constraints, where each @-function is represented by a node. The ≤ relation is represented by edges between the nodes. Every edge is also weighted by means of the C constant.

As such, the constraint:

$$@(\text{alive}, i) \leq @(\text{ping}, i) + 20$$

would result in the following graph:



Evaluating this constraint is based on filling the nodes with the occurrence times of the events. Afterwards the graph is transformed and then the graph is analyzed for specific paths. The actual analysis process is described in [6] and [12] and is based on a lemma also presented in the paper.

Now there remains one issue: when should this graph filling, transformation and analysis take place? Normally, this will take place when a particular event of interest occurs (this is an event that is part of the monitored constraint). Another aid in checking constraints at run-time is to generate additional events by means of the triggering of a timer. This timer could have been set by the checker to denote that a particular event had to occur before the triggering of the timer. This can be used to detect violations at the moment a deadline elapses rather than when the late event occurs.

It is clear that this analysis process takes some time if the events of interest occur frequently. The different steps in the actual process (graph filling, transformation and analysis) take some time, and the time needed grows with the size of the graph ($O(n^3)$ with n the number of nodes).

3.3. Conclusions

The previous sections have shown that monitoring timing constraints in embedded systems is not trivial. One of the approaches, RTL monitoring, is interesting in the sense that it is one of the few existing formalized[7] approaches to monitor timing in an application.

3.3.1. Intrusion

Since intrusion has to be avoided, or at least minimized, it is important to determine correctly the amount of intrusion introduced in the system. There are two facets to intrusion: intrusion caused by the probes and intrusion caused by the monitoring activity.

To keep the probe intrusion low and predictable, it is important that the structures used for passing events between the probes and the monitoring system, are efficient and non-blocking. Also, the amount of processing done in the probe has to remain low (a probe should only record information) and predictable (only use deterministic operations and data structures). The probe has to remain in the system at deployment time otherwise unwanted probe effects can occur in the deployed system.

To keep the monitoring activity low and predictable, the actual processing can be moved to another node on the network. If the monitoring has to occur on the monitored node, the monitoring task has to be taken into consideration when developing the system and assigning priorities to the tasks. In the case of rate monotonic scheduled systems, the monitoring task can run during idle periods. As such, the removal of the monitoring task in the final system will go unnoticed. Unfortunately, our component system and run-time

[7] There exist of course ad hoc techniques for performing timing monitoring, often synchronous (embedded) approaches.

monitor is implemented as a layer on top of the JAVA VM, and therefore we cannot have real control over the actual scheduling of the threads.

3.3.2. Checking Timing Constraints

Before being able to measure a system's property one has to be able to quantify the property that is measured. In the RTL approach, this is possible by writing RTL formulas that define the timing constraints. These formulas are then evaluated by a kind of RTL evaluation engine (satisfiability checker). This is interesting since there is only one engine that works for all timing constraints (deadline or delay constraints).

Another approach is using dedicated checking engines; each type of constraint has its own evaluation engine. The advantage of this is that the engine can be fine-tuned to that particular constraint. A generic approach like the RTL satisfiability checker is slower than a dedicated constraint-type checking engine.

3.3.3. Event History

Another issue when monitoring embedded systems is the size of the event history. If the monitoring activity has to be done on the monitored device, the device needs a necessary amount of available memory.

Even on systems with lots of memory, it is not possible to keep all events in the history. To know which event to keep and which not, the constraint has to be analyzed and information from this analysis can predict the maximum size of the event history. Next to determining the maximum size[8] of the history, this history has also to be managed at runtime; events have to be removed when they are not needed anymore. We will present a particular event history management algorithm in section 4.

3.3.4. Distributed Monitoring

As was made clear in section 3.1, the fact that a monitored system is distributed adds to the complexity of the monitoring system. A solution for the clock synchronization problem (see [7]) consists of taking the maximum clock deviation into account when performing the graph analysis. The work done in [7] also investigates how the number of messages exchanged between the monitored and monitoring nodes can be minimized. It is shown that the minimization problem is NP complete.

It is clear that monitoring distributed systems has some problems that are quite hard to solve. Some of these problems have partially been solved, but the solutions often dictate assumptions that are difficult to meet.

---

[8] A trivial technique is to allow event histories of any size, by raising the size dynamically when necessary.

4. SEESCOA Monitoring System

In this chapter, a technical description of the design and implementation of the monitoring subsystem is presented. First, a description of the main parts and the interactions between them is given. Then each part is described in more detail. A conclusion is given at the end, in which we describe some advantages/problems of the system and where some directions for the future are described.

4.1. Overview of the Monitoring System

The system has been designed as such that the *event processing* part of the monitor can be put on any node in a network, while the *event gathering* part has to remain in the component system. This is shown in figure 2. However, if timing violations have to be reported back to components that are part of the monitored application, then the monitoring system has to reside on the same system. In that case, there is no monitor proxy: all communication between the probe system (event gathering) and monitoring system (event processing) occurs directly. Currently, this last option is the supported one. We are working on the addition of remote monitoring functionality.

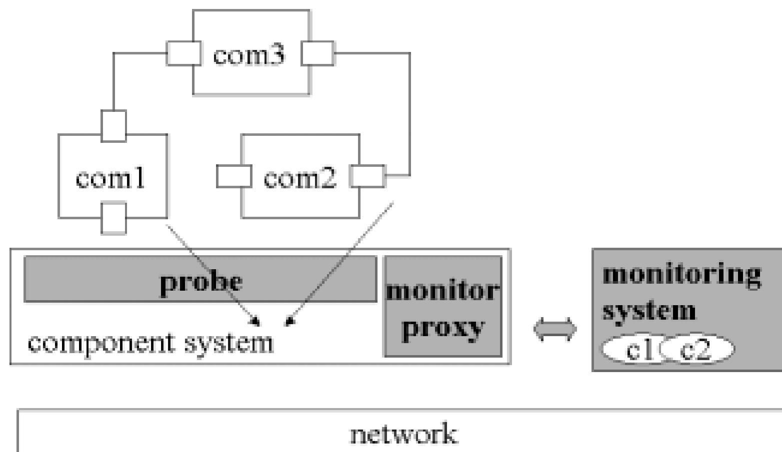


Figure 2: overview of the monitoring system

The *probes* part of the monitoring subsystem is responsible for intercepting events occurring in the running application. The types of events that can be intercepted are events related to the **sending**, **receiving** and **end of processing** of messages exchanged between components. In fact, the *probes* part consists of multiple probes attached to the ports[9] of the components. If one of the monitored ports receives or sends an event, the probe intercepts this and generates an event that is passed to the monitoring system.

[9] The port of a component is the place where messages are received and sent out.

The *contract checking system* is responsible for accepting and analyzing the events coming from the probes. The type of analysis that is done on these messages is not predefined: one is able to add *contract checkers* to the monitoring system. These contract checkers then get the chance to process the events.

The principle of contract checkers enables the monitoring of several characteristics related to the running application. For instance, there could be a *synchronization contract checker* (which monitors the sequence of the messages) and a *deadline contract checker* (which monitors deadlines for particular messages). Both contract checkers can then be added[10] to the contract checking system.

In the next sections we will discuss the monitoring system in more detail.

4.2. The Monitoring Subsystem

As was made clear in section 4.1, the monitoring subsystem consists of two important parts: the *probes* and the *contract checking system*. This subdivision is visible in the package structure of the code. Following packages are of importance:

- **highresclock**: contains a high resolution clock and timer,
- **probe**: contains functionality that is injected in the component system (necessary for event gathering),
- **shared**: contains structures that are shared between the monitoring packages or between the monitoring packages and the component system or tool,
- **runtime_system**: contains the classes that implement the actual event processing functionality,
- **configuration**: this package contains classes responsible for starting the monitoring system, on the monitored and monitoring node.

4.2.1. Package highresclock

This package contains a high resolution clock and timer necessary for the monitoring system. The monitoring system needs a notion of time when timestamping events and when analyzing these events. Since the standard Java time retrieval mechanism is not powerful enough we had to provide support for this ourselves.

Problem Statement

One of the problems with Java is the lack of a high resolution timing mechanism. If a Java program needs to retrieve the time on the VM it is running, the static method `System.currentTimeMillis()` has to be called. This method returns the time in

[10] However, be aware of the fact that contract checkers do need resources to perform their analysis. If the contract checking system resides on the monitored node, then the level of intrusion increases by adding more contract checkers.

milliseconds that have elapsed since January 1st 1970, 00.00. There are two problems with this. Firstly, standard java machines do not return the time in steps of 1 millisecond, but rather in steps of 10 milliseconds. This means that the resolution of the clock is rather high (10 milliseconds). Secondly, even if the resolution would be 1 ms, this is still to coarse grained for certain applications.

Overview of the package

The package consists of a couple of classes needed for implementing a high resolution clock (in subpackage timer).

The Timer interface defines timer-related methods, like starting, stopping and retrieving relative times (relative to start and stop). The timer interface has also a getTimeStamp() method for retrieving the absolute time. The return value is a long representing the absolute time in microseconds.

The `Timer` interface is implemented by `NativeTimerHighRes` and `JavaTimer`. The first implementation is a native implementation of the timer. This enables retrieval of time at a much higher resolution (see below) than using standard Java mechanisms. The second implementation (`JavaTimer`) makes use of the standard Java mechanisms and is thus less usable in certain circumstances.

4.2.2. Package shared

This package contains structures that are shared between the different packages of the monitoring system, the component system and the composer tool. The package consists of four subpackages: *contracts*, *events*, *rtfifos* and *violations*.

4.2.2.1. Package shared.contracts

The `contracts` package contains all classes that are used to represent runtime contracts. The `RuntimeContract` class is the root of all runtime contracts. Every `RuntimeContract` has a unique identifier, a name and a list of participants. Allowed participants are component instances, port instances and connectors. A `RuntimeTimingContract` is the root class of all timing contracts.

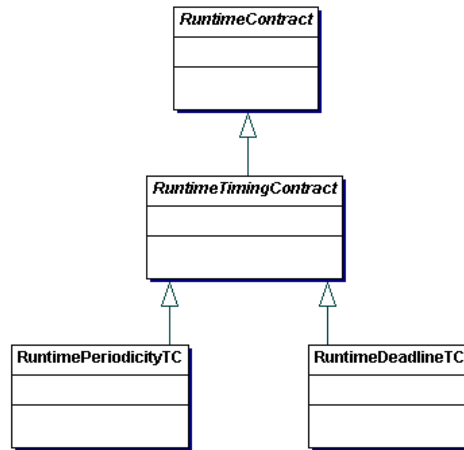


Figure 3: runtime contract hierarchy

The figure contains also two concrete subclasses: `RuntimeDeadlineTC` and `RuntimePeriodicityTC`. The first class represents deadline constraints imposed on the messages exchanged between components. A `RuntimeDeadlineTC` is represented by following properties:

- **start event**: when this event occurs the stop event has to occur before the deadline. A start event is represented by its port identifier, its message identifier, the type of the message (sent, received, processed) and the occurrence of the event (an integer or unlimited[11]).

- **stop event**: this event has to occur before the deadline has elapsed. The stop event is also represented by a port identifier, a message identifier, the message type and the occurrence of the event. If the occurrence of the start event is unlimited, then the occurrence of the stop event should also be unlimited.

- **deadline**: an int value indicating the maximum time (in milliseconds) that may elapse between the occurrence times of the start and stop events.

The second class represents periodicity contracts. A periodicity contract is represented by following properties:

- **start event**: when this event occurs the periodic events have to occur within their periods. The timeline starts at the occurrence time of the start event. A start event is represented by its port identifier, its message identifier, the type of the message (sent, received, processed) and the occurrence of the event (must be an integer).

[11] This means that the constraint should apply for each occurrence of the start event. This is denoted with an asterisk (*).

- **stop event**: this event denotes the end of the timeline. The stop event is also represented by a port identifier, a message identifier, the message type and the occurrence of the event (must also be an integer).

- **periodic event**: this event has to occur periodically. Every periodic event has to occur within its associated period (not earlier, not later). As with the start and stop events, a periodic event is represented by its port identifier, its message identifier and its message type. A periodic event has no occurrence value.

- **period**: an int value indicating the period length (in milliseconds).

We will discuss in section 4.2.4 how these contracts can be monitored.

4.2.2.2. Package shared.events

This package contains classes that are shared between the probes and the contract checking system. The communication between probes and the contract checking system occurs by means of events. An event is the indication that something has occurred in the running application.

The most important class here is `MessageEvent`. The probes that are present in the component system generate these message events when messages are sent, received or processed. In future versions of the monitoring system, we could add more event types. As such it would be possible to check other types of actions that occur in the component system (like object creation, I/O, …).

A `MessageEvent` has following attributes:

- **sender**: the identifier of the port that sends the message,

- **received**: the identifier of the port that receives the message,

- **message**: the identifier of the message that is exchanged between the sender and the receiver,

- **type**: the type of the message (sent, received or processed)

- **occurrence**: the occurrence of this event (an integer)

- **timestamp**: the absolute time at which this event has occurred.

The probes that intercept message-related actions are thus responsible for filling in these attributes.

4.2.2.3. Package shared.rtfifos

Events are the objects that are exchanged between probes and the contract checking system. The mechanism for exchanging these events is a FIFO abstract data type.

As one can recall, the intrusion effect of the probe has to remain *low* and *deterministic*. This means that the processing a probe performs may not be blocked or

suspended by an unbound period of time. Therefore we have added the RTFifo class to this package. An RTFifo is a FIFO ADT in which putting an event (by means of the probe) takes constant time. The retrieving of an event takes quasi-constant time.

The implementation of the queue is done by means of a circular buffer that is filled by the probe and emptied by the contract checking system. The buffer is not synchronized (since this would introduce non-deterministic blocking periods). A consequence of this is that a buffer can overflow and erase events that were not retrieved yet.

To eliminate overflow problems, it is possible to attach an RTFifoObserver to the RTFifo. This observer uses a thread to regularly[12] check the load of one or more buffers. If the load exceeds the maxLoadFactor for that buffer then a LoadFactorObserver is notified of this. This is an interface that has to be implemented[13] by the programmer.

The class hierarchy for this package is shown in figure 4. The RTFifo class is subclassed in MessageRTFifo. This is the FIFO that is used between probes generating MessageEvents and the contract checking system.

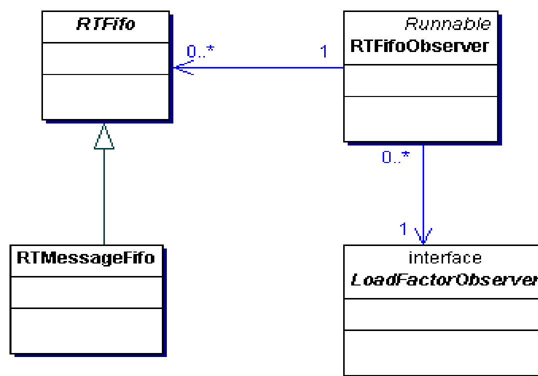


Figure 4: RTFifo package hierarchy and dependencies

To avoid the necessity to create MessageEvent objects every time a message-related action occurs, the RTMessageFifo queue is initialized with empty MessageEvent objects. As such, the probes do not create new MessageEvent objects; instead they fill in the required fields. This ensures that the probe intrusion remains low and predictable (since object allocation on the heap takes an unknown amount of time).

[12] The rate at which the thread checks the FIFO queues can be initialized at creation time.

[13] The appropriate thing to do when the load factor is exceeded is of course to empty the buffer.

4.2.2.4. Package shared.violations

This last shared package is used by any class that is interested in receiving contract violations coming from the contract checking system. The contract checking system creates violation objects whenever a constraint is violated.

This package contains the root class `Violation` that is subclassed in `TimingViolation`. This type of violation indicates that a particular timing constraint has not been adhered to. `TimingViolation` is then further subclassed in `DeadlineTimingViolation` and `PeriodicityTimingViolation`:

The first type of violation is used to indicate that a particular deadline has been missed. An object of this class contains information about the particular `RuntimeDeadlineTC` that was violated, the absolute deadline and the actual occurrence time. Notice that a deadline violation is only reported after the occurrence of the appropriate event, not at the moment that the deadline is exceeded.

The second type of violation indicates that a periodic event occurred too early or too late. This violation also contains information about the `RuntimePeriodicityTC` that was violated, the type (too early or too late), the period and the actual occurrence time. Notice also that a periodicity violation is only reported after the occurrence of the appropriate event, not at the moment that a cycle period is exceeded.

4.2.3. Package probe

Classes in the `probe` package are responsible for intercepting events occurring in the component system. Currently, there is one generic class: `PortProbe`. An object of this class can be attached to a port and a RTFifo queue. Events occurring at the port (sending, receiving and processing of a message) are intercepted and put in the queue.

A subclass of `PortProbe` – `MessageTimingProbe` – also intercepts messages, and adds a timestamp (indicating the occurrence times) to these message events. These timestamped message events are then put in a `MessageRTFifo`. Actually, not the entire message is put in the FIFO, since it is not necessary to know the arguments of the message for performing time monitoring. As such, only the sender, receiver, message identifier, occurrence number and the timestamp are put in the queue.

Probe objects are attached to the ports of a component when this component is created. The actual mechanism for doing so is described in section 4.2.5.

4.2.4. Package runtime_checker

The package `runtime_checker` contains all the classes that implement the contract checking[14] functionality:

[14] Note that contract checking is a kind of event processing activity.

- **ContractMonitoringSystem**: the root class of the contract checking functionality. It consists of a ContractChain object and ViolationProcessor.

- **ContractChain**: is a collection of ContractChecker (package runtime_checker.contract_checker) objects. These objects implement a particular checking algorithm. The ContractChain object retrieves Event objects from EventCollector object(s).

- **ContractChecker**: a ContractChecker is an object that monitors a particular constraint imposed on the events. The currently implemented checker is a TimingContractChecker which examines deadline and periodicity issues.

- **EventCollector**: an object of this class is responsible for intercepting events generated by the probes. Mostly, an EventCollector will be attached to the RTFifos; in that case, the collector is responsible for emptying the RTFifo queues.

- **ViolationProcessor**: if one of the ContractChecker objects reports a violation of a contract, this is done through the ViolationProcessor. This object is responsible for notifying eventual ViolationObservers.

- **ViolationObserver**: this interface has to be implemented by objects that want to receive notifications about contract violations.

- **ViolationBuffer**: this buffer is used between the ContractCheckers and the ViolationProcessor to indicate that violations have occurred.

Figure 5 represents the class hierarchy.

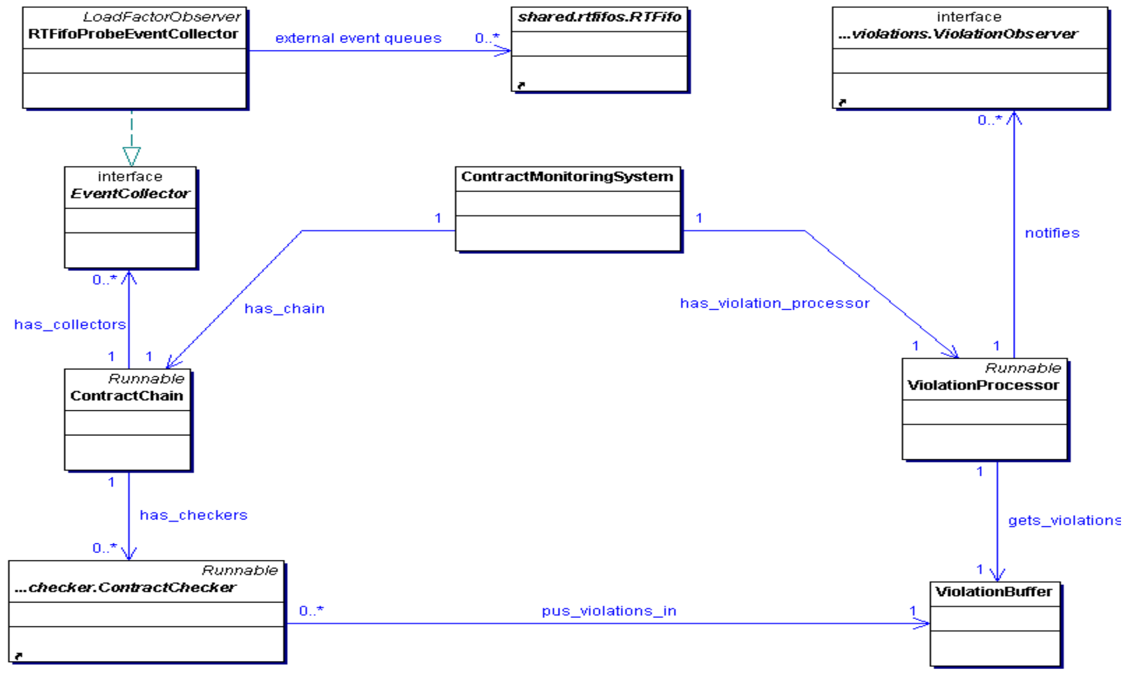


Figure 5: runtime checker class hierarchy

4.2.4.1. Package runtime_checker.contract_checker

This package contains classes that one might find helpful when developing your own monitoring functionality. Following classes are of importance:

- **ContractChecker**: this class has to be subclassed by a class implementing the concrete contract checking functionality.

- **EventHistory**: every `ContractChecker` object has an `EventHistory` object. This object is a collection of event occurrences that are of importance to the checker. The implementation of the history is done by means of `TreeMap`[15] objects. The history stores its events based on the *event type* and the *event occurrence*. The event types are stored in a `TreeMap`, and for each event type there is a `TreeMap` to store the `EventHistoryItems` (which represent event occurrences).

- **EventHistoryItem**: objects of this class represent event occurrences. They are specified by a `type` attribute (indicating the event type) and an `occurrence` attribute. To attach more information to event occurrences, one has to subclass this class.

---

[15] TreeMaps are red-black trees: the insertion, retrieval and removal takes O(log(n)) time, with n the number of items in the tree.

- **ContractActivationPool**: a `ContractChecker` object has also a `ContractActivationPool` object. This object is responsible for storing `ContractActivation` objects. A contract activation is a concept that is used to represent *something that has to be checked in the future*. This is often needed when checking contracts, since contracts are often partially validated; this means that some events have not occurred yet, and as such it is not possible to fully validate the contract. We will come back on this later.

- **ContractActivation**: objects of this class represent contract activations, or better: 'partially' validated contracts. The `ContractChecker` is responsible for adding, retrieving and removing these activations from the pool. Contract activations can also refer to one or more event history items. If no more activations refer to a particular event history item, then this item is removed[16].

Figure 6 shows a representation of this package. Important notice: every `ContractChecker` has its own thread of control. This ensures that the actual event processing activity is done separately from the event gathering activity (which is done by the contract chain). It also ensures that a particular contract checker does not take all the available processing time.

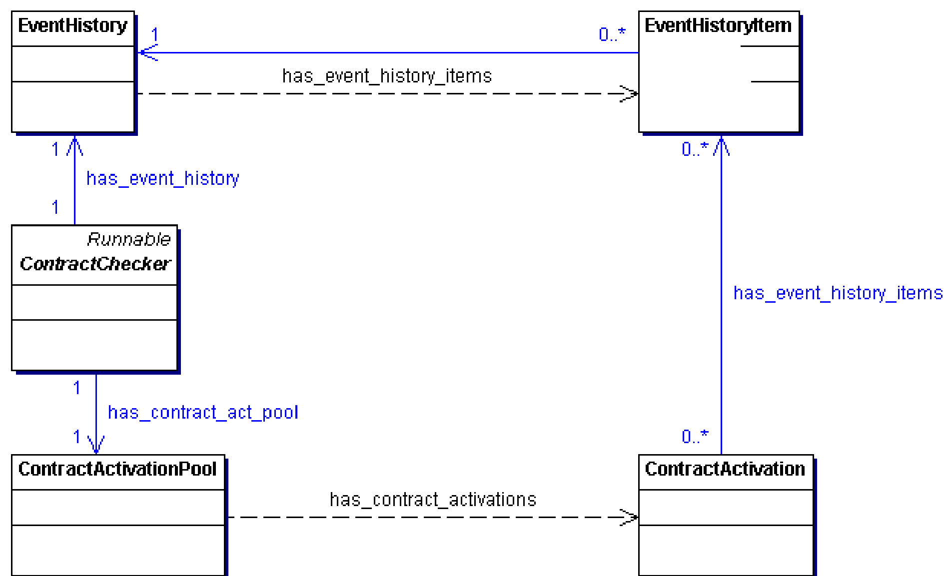


Figure 6: class hierarchy of the runtime_checker.contract_checker package

4.2.4.2. Package runtime_checker.contract_checker.timing_contract_checker

This last package contains all classes necessary for performing monitoring of timing contracts (see figure 7). There exist different types of timing contracts (deadline,

[16] In fact, the ContractChecker is not responsible for removing EventHistoryItems from the EventHistory; this is done indirectly via the ContractActivationPool. This avoids the potential problem of memory leaks.

periodicity, freshness, and so on), and these have to be defined and implemented separately. We will come back on this in section 4.2.6.

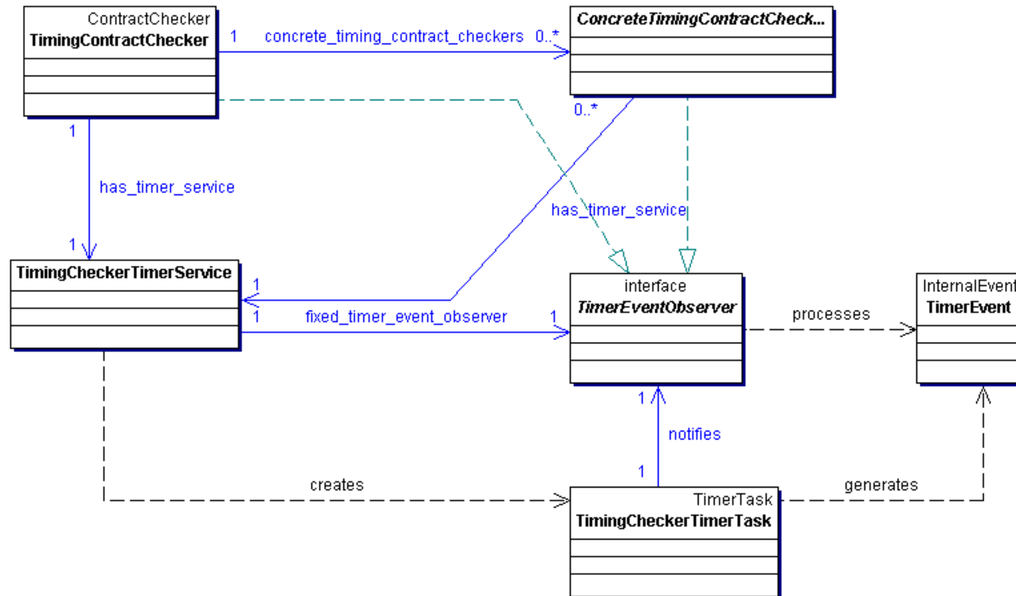


Figure 7: timing contract checker class hierarchy

An object of the `TimingContractChecker` class can hold many `ConcreteTimingContractChecker` objects. These objects are objects that perform the actual event processing. For instance, there will be multiple `DeadlineChecker` objects (for each deadline contract) and multiple `PeriodicityChecker` objects (for each periodicity contract). All these concrete checker objects share the same thread (the thread that is attached to the `TimingContractChecker` object). As such, there is only one thread performing all the timing monitoring in the system.

Since most concrete checkers will need some kind of timer service, three concepts have been added to this package:

- **TimingCheckerTimerService**: every `TimingContractChecker` has a reference to an object of this class. This object enables concrete checkers to schedule an event that has to occur in the future. Therefore, they have to provide an absolute time at which the event should occur.

- **TimingCheckerTimerTask**: objects of this class are attached to a particular event that was handed over to the timer service. These task objects are then scheduled by a `java.util.Timer`[17]. The timer is responsible for scheduling the `TimingCheckerTimerTask` at an absolute time. In this case, the execution of the task will result in the triggering of the event associated to it.

---

[17] For more information see the JDK API information (java.util.Timer and java.util.TimerTask)

- **TimerEventObserver**: this interface is implemented by classes that want to receive notifications of events that were scheduled earlier. Normally, this interface is implemented by the one who has scheduled the event.

The timer service described above is important since a lot of concrete checkers will need to use timers. For instance, a `DeadlineChecker` object will need to be notified if a particular deadline has passed. The same applies for a `PeriodicityChecker` that needs to be notified when a particular period has elapsed.

4.2.5. Package configuration

This package consists of two subpackages: `monitored_node` and `monitoring_node`. The first one contains classes that have to be deployed on the monitored node, while the second one will contain classes to be deployed on the monitoring node.

4.2.5.1. Package configuration.monitored_node

The `ProbedComponentImpl` and `ProbedComponentImplFactory` are responsible for attaching `MessageTimingProbes` to the ports of a component. The factory class creates a new `ProbedComponentImpl` object each time a component is created in the component system. This object is then attached to the component.

A `ProbedComponentImpl` overrides the two methods offered by `ComponentImpl`: `filterSendMessage()` and `filterReceiveMessage()`. The first message is invoked each time a message arrives on one of the ports of a component, and the second message is invoked when a message is sent out over one of the ports. As such, all the communication in which the component is involved can be intercepted here.

To install a `ProbedComponentImplFactory` in the component system, the command line option

```
impl: configuration.monitored_node.ProbedComponentImplFactory
```

has to be passed to the component system. The component system will then use this factory to created `ProbedComponentImpl` objects for each component that is created.

At component creation time, the `ProbedComponentImpl` object will query a `ProbeManager` instance to know if the component has some ports that need to be probed[18]. The `ProbeManager` returns all `PortProbe` objects for this component to the `ProbedComponentImpl` object. At that moment, the component is fully probed: every message that enters or leaves a probed port of the component will be intercepted.

[18] The `ProbeManager` provides a mapping between component identifiers and `PortProbe` objects.

Another important class is the `MonitoredNodeManager`. This is the root class of the entire monitored node monitoring system. It is responsible for creating a `ProbeManager` and a `ViolationReporter`. The `ViolationReporter` is an object that can report violations on screen, to file or to other components. Currently, only the first possibility is supported. The `MonitoredNodeManager` is also responsible for reading contract and probe configuration information. Both the contract and probe information can be stored in a file. The `MonitoredNodeManager` also creates a runtime checker subsystem to perform local monitoring.

The `MonitoredNodeManager` is in fact a component that can receive following messages: `StartMonitoring()`, `StopMonitoring()` and `Configure()`. The first two messages have no arguments, and are used to respectively start or stop the monitoring activity. The `Configure()` message expects some (String) arguments, necessary to initialize the monitored node monitoring system:

- **monType:<LOCAL_MONITORING|REMOTE_MONITORING>**:
  indicating if the runtime checking should occur locally or remote. Currently only the first option is supported.

- **monViolationReporting:<TO_SCREEN|TO_COMPONENTS|TO_FILE>**:
  tells the monitored node system where timing violations should be reported. TO_SCREEN is the option currently supported.

- **monProbeFile:<filename>**: filename (and path) of the probe information file,

- **monContractFile:<filename>**: filename (and path) of the contract information file.

After having configured the `MonitoredNodeManager`, the `Done()` message is sent back. At that moment, one is allowed to send `StartMonitoring()`. The monitored node system will then start intercepting and analyzing events.

4.2.5.2. Package configuration.monitoring_node

This package contains functionality responsible for the remote monitoring of systems.

4.2.6. Monitoring Timing Constraints

Till now, the general architecture of the monitoring system has been discussed. Here, a description is given how the deadline monitoring has been implemented.

4.2.6.1. Overview

The deadline monitoring functionality has been implemented in the `DeadlineChecker` class. This class subclasses from `ConcreteTimingContractChecker` (introduced in section 4.2.4). A deadline checker object is attached to one `RuntimeDeadlineTC` object. This object contains the contract parameters. As such, there is one deadline checker object for each deadline timing contract.

4.2.6.2. An example

We present here an example specification of a deadline timing contract. Suppose that every time message msg1 is received at port comp1.port1, msg2 has to be sent from port comp1.port2 within 100 milliseconds. The specification of this timing contract is as follows:

- **start hook**: comp1.port1.msg1()$_{receive}$[*]
- **end hook**: comp1.port2.msg2()$_{send}$[*]
- **duration**: 100

See [3] for more information on this deadline timing contract specification notation.

4.2.6.3. Event History and Activation Pool

The deadline checker makes heavily use of the event history and the activation pool. Every time an interesting event occurs, the deadline checker puts it in the event history. The event history used by the deadline checker can be shared with other concrete timing contract checkers. This is more efficient since different checkers often need to store the same events.

The deadline checker also makes use of a contract activation pool. This pool is not shared with other concrete checkers. In this pool, the deadline checker puts 'things to remember'. For instance, once a start event has occurred, the deadline checker starts a timer and needs to store a start activation in the pool. This start activation contains a reference to the start event stored in the history. Once the timer triggers, the contract checker knows for which start event occurrence this timer has triggered.

The deadline checker puts also stop activations in the pool. This is necessary to remember that a particular stop event occurrence has occurred before the corresponding start event occurrence. Otherwise, when the start event occurs, a violation will be reported after the specified duration a violation will be reported since the stop event has already occurred and will not occur again.

4.2.6.4. Deadline Checking Algorithm

Once the system is started, the deadline checker object receives message events that are intercepted at the probes. Events resulting from `comp1.port1.msg1` and `comp1.port2.msg2` are analyzed, other events are not analyzed.

While the deadline checker is running, it performs the following activities:

- Whenever a start event occurs (`comp1.port1.msg1`), one of the following actions is taken:
    - o The deadline checker looks in its activation pool if there is a corresponding stop occurrence. If this is the case, the stop activation is removed from the activation pool. No deadlines are violated, since the stop event occurred before the start event.

    - Otherwise, the deadline checker creates an activation for the start event occurrence and puts it in the activation pool. It also creates a timer event that has to be triggered at an absolute time (= occurrence time of the start event + deadline). The `TimingCheckerTimerService` is then told to schedule the timer event at the absolute time. The timer event also gets a reference to the start event occurrence.

- Whenever a stop event occurs (`comp1.port2.msg2`), one of the following actions is taken:

    - The deadline checker looks in its activation pool if the corresponding start event has already occurred. If this is the case:

        - It checks if the start event activation has already been processed by a timer event (which means that this stop event has occurred too late). If this is the case, nothing happens, since a violation was already reported when the timer event triggered.

        - If the start event activation has not been processed yet, the deadline checker retrieves the occurrence time of the start event and controls if the occurrence time of the stop event <= occurrence time of start event + deadline. If the stop event occurred too late a violation is reported. Also, the start activation is removed from the activation pool and it's marked as 'processed' (since there could still be a timer event – with a reference to this activation – that could trigger in the future).

    - Otherwise, the deadline checker creates an activation to indicate that the stop event has occurred and adds it to the activation pool.

- Whenever a timer event occurs:

    - The deadline checker controls if the start activation associated with this timer event has already been processed.

        - If this is the case, nothing happens, since the stop event has already been dealt with,

        - Otherwise, a violation is reported. The start activation is then marked as 'processed' and is left in the activation pool. (Note that the triggering of a timer event has no influence on the activation pool.)

4.2.6.5. Conclusion

The internal working of the deadline checker may seem complex, but this is a consequence of the complexity of timing monitoring. The functionality of the periodicity checker is even more complicated and will not be discussed in this document.

The event history used by a deadline checker can be shared among other concrete timing checkers (deadline or periodicity checkers). As such an event occurrence is never stored twice.

To remember that a particular event has occurred but needs to be processed somewhere in the future, the deadline checker makes use of an activation pool. Adding and removing activations from the pool is done by the deadline checker. Removing event occurrences from the history is done automatically: every activation refers to one or more event occurrences. Whenever an activation is removed; the event history is notified of this and lowers the reference count for that particular event occurrence. When there are no more activations referring to an event occurrence, this event occurrence is removed. This makes event history management automatic and avoids the problem of memory leaks.

One problem with our approach is that event histories and activation pools can grow significantly. This is especially the case if start events happen at a high rate: then many activations and timer events will be created. This problem is however not really related to the implementation of the deadline checker, but rather to the problem domain itself.

5. Conclusion

This report gave an overview of existing and well-known concepts and techniques related to runtime monitoring. In the second part of the report we discussed some issues related to the monitoring of timing constraints in embedded systems. One particular approach to runtime monitoring (RTL) has been described in detail.

The third part has shown our approach for monitoring of timing constraints. A separation was made between the activity of collecting system information (by means of probes) and the processing of this information (by means of contract checkers). Some problems and solutions related to runtime monitoring of timing constraints have been illustrated. Mostly, these problems had to do with the efficiency, determinism and memory usage of the monitoring activities.

The monitoring system is however not finished yet. At first, support for remote monitoring of multiple nodes can be added. Secondly, additional contract checking algorithms should be defined and implemented. Finally, since the intrusion and efficiency of the monitoring system is of utmost importance, there is a need for detailed profiling and efficient algorithms.

References

[1] D. Urting, Y. Berbers, S. Van Baelen, T. Holvoet, Y. Vandewoude, P.Rigole, A Tool for Component Based Design of Embedded Software, Proceedings of the Fortieth International Conference on Technology of Object-Oriented Languages and Systems, Sydney, 2002, pp. 159-168

[2] Lexico LLC, http://www.dictionary.com

[3] URTING D., VAN BAELEN S., HOLVOET T., BERBERS Y. (2001): Embedded Software Development: Components and Contracts*, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems,* Anaheim, USA: 685-690, ACTA Press.

[4] URTING D., BERBERS Y., VAN BAELEN S., HOLVOET T., VANDEWOUDE Y., RIGOLE P. (2002): A Tool for Component Based Design of Embedded Software, *Proceedings of the Fortieth International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS Pacific 2002, Sydney, Australia: 159-168, Australian Computer Society Inc.

[5] JAHANIAN, F., MOK, A.K., STUART, D.A. (1988): Formal Specification of Real-Time Systems. Technical Report UTCS-TR-88-25, Department of Computer Sciences, the University of Texas at Austin, USA.

[6] LOYALL, J.P., SCHANTZ, R.E., ZINKY, J.A., BAKKEN, D.E. (1998): Specifying and Measuring Quality of Service in Distributed Object Systems. IEEE Proceedings of ISORC'98, Japan.

[7] MOK, A.K., LIU, G. (1997): Efficient Run-Time Monitoring of Timing Constraints. Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97), Montreal, Canada.

[8] RAJU, S.C.V., RAJKUMAR, R., JAHANIAN, F. (1992): Monitoring Timing Constraints in Distributed Real-Time Systems. IEEE Real-Time Systems Symposium, Arizona, USA: 57-67.

[9] SINGHAL, A. (1997): Real Time Systems: A Survey. Computer Science Department, University of Rochester, New York, USA.

[10] THANE, H. (2000): Monitoring, Testing and Debugging of Distributed Real-Time Systems. Ph.D. Thesis, Mechatronics Laboratory, Department of Machine Design, Royal Institute of Technology, Sweden.

[11] SCHROEDER, B (1995): On-Line Monitoring: A Tutorial. IEEE Computer, Volume 28, Number 6, June 1995, pp. 72-78.

[12] BLOM, J. (2001): Monitoring of Embedded Distributed Real-Time Systems, Department of Computer Engineering, Mälardalen University, Västeras, Sweden, February 26, 2001.

[13] CHODROW, S.E., JAHANIAN, F., DONNER, M. (1991) Run-Time Monitoring of Real-Time Systems. Real-Time Systems Symposium, Dec. 1991, pp. 74-83