

Asynchronous Event Handling in the Real-Time Specification for Java

MinSeong Kim

Department of Computer Science
University of York, UK
djmin@cs.york.ac.uk

Andy Wellings

Department of Computer Science
University of York, UK
andy@cs.york.ac.uk

ABSTRACT

The Real-Time Specification for Java (RTSJ) is becoming mature. It has been implemented, formed the basis for research and used in serious applications. Some strengths and weaknesses are emerging. One of the areas that require further elaboration is asynchronous event handling. The primary goal for `AsyncEventHandlers` is to have a light concurrency mechanism. Some implementation will, however, simply map an `AsyncEventHandler` to a server thread and this results in undermining the original motivations. In this paper we closely look at some known RTSJ implementations in terms of their asynchronous event handling techniques. We also present our own model of AEH, a monitor model. We then define formal models of RTSJ AEH implementations using the automata formalism provided in the UPPAAL tool. Using the models their properties are explored and verified. The results of the verifications have shown that the current AEH systems used in some RTSJ implementations can be optimised so that the original motivations for AEH could be better achieved.

Categories and Subject Descriptors

C.5 [COMPUTER SYSTEM IMPLEMENTATION]:
General

Keywords

Asynchronous Event Handling, jRate, Java RTS, RI, RTSJ, Leader&Followers, Monitor design

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [2] has made several modifications to the Java specification in order to make true real-time processing possible. The primary goal of the RTSJ is to provide a platform, a Java execution environment and application program interface (API), that lets programmers correctly reason about the temporal behaviour

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

JTRES '07 September 26-28, 2007 Vienna, Austria
Copyright 2007 ACM 978-59593-813-8/07/9 ...\$5.00.

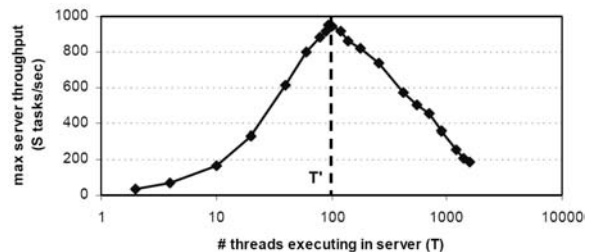


Figure 1: As the number of concurrent threads T increases, throughput increases until $T \geq T'$, after which the throughput of the system degrades substantially [13].

of executing software. In an attempt to provide the flexibility of threads and the efficiency of event handling, the RTSJ generalises Java's mechanism for Asynchronous Event Handling (AEH) by distinguishing between events, something that can happen, and event handlers, logic that will be scheduled for execution when those things happen. This distinction in the RTSJ is realised by introducing the notion of real-time asynchronous events, `AsyncEvent` class and their associated handlers, `AsyncEventHandler` class. In the RTSJ, asynchronous events are viewed as dataless occurrences that can be either fired periodically or only once by the application or associated with the triggering of interrupts in the environment. The relationship between events and handlers is many to many (i.e. a single `AsyncEvent` can have one or more `AsyncEventHandlers`, and a single `AsyncEventHandler` can be bound to one or more `AsyncEvents`). Also the handlers can be bound to POSIX signals for the case where the RTSJ is being implemented on top of a POSIX-compliant operating system.

Both `AsyncEventHandlers` and `RealtimeThreads` are `Schedulable` objects in the RTSJ. However, in practice, underlying real-time threads (called servers in this paper) provide the vehicles for execution of `AsyncEventHandlers`. Therefore, it is necessary to bind an `AsyncEventHandler` to a server and this binding will be performed dynamically at run-time. If this binding latency, which inevitably incurs when two objects are being attached to each other is not desired, `BoundAsyncEventHandler` objects can be used to eliminate it. Once a server is bound to a `BoundAsyncEventHandler` object, the thread becomes the only vehicle for execution for that particular handler. The use of `BoundAsyncEventHandler` objects, however, should be minimised as a server per `AsyncEvent`

`tHandler` takes no advantage of having the means of serving events by handlers rather than threads. However, they are appropriate for interrupt handlers. The point here is that making `AsyncEventHandlers` use far fewer system resources than actual servers use is a primary intention of the RTSJ. Regardless of how well the implementation is designed, as the number of threads in a system grows, operating system overhead increases, leading to a decrease in the overall performance of the system [13]. There is typically a maximum number of threads T' that a given system can support, beyond which performance degradation occurs. This phenomenon is demonstrated clearly in Figure 1. Therefore, the key challenge in implementing event handlers is to limit the number of servers without jeopardising the schedulability of the overall system [12]. Furthermore, the resulting implementation must provide efficient and predictable mapping between these two notions, event handlers and real-time server threads.

The RTSJ, however, does not provide any guidelines on how these events and their handlers can be implemented to guarantee their timing requirements. As a consequence, AEH in the RTSJ only requires to comply with the scheduling semantics and `AsyncEventHandler` objects are simply executed in the context of the currently available server. In [6, 12], possible implementation strategies for the RTSJ's AEH have been proposed and some of their corresponding feasibility analyses also have been derived. One of the implementation strategies appeared in both papers is to use multiple queues and multiple servers. There must now be a dynamic association of handlers to servers and a solution when handlers block.

As mature implementations of the RTSJ are becoming available, an assessment of the current implementations is timely. The goal of this paper, therefore, is to explore how the AEH of the RTSJ is implemented to achieve a lightweight concurrent mechanism [7] in some of the current implementations of the RTSJ, such as *RI*, *jRate*, *OVM* and *Java RTS*, and to present a new model of AEH. In Section 2, a brief overview of the RTSJ AEH model is presented. Section 3 discusses the current AEH implementations and the new model. A brief explanation of the UPPAAL tool, a modelling architecture for AEH and UPPAAL automata models are presented in Section 4. In the following section we formally analyse the models, and Section 6 draws conclusions.

2. THE AEH MODEL OF THE RTSJ

When an `AsyncEvent` object is triggered by a call to the `fire` method, all the `AsyncEventHandlers` associated with the `AsyncEvent` are scheduled for execution according to their execution eligibility. Each `AsyncEventHandler` has a count called `fireCount`. When an `AsyncEvent` occurs, the count is atomically incremented. The attached `AsyncEventHandlers` are then released for execution.

`AsyncEventHandlers` behave like `RealtimeThreads` as they have the same characteristics as `RealtimeThreads` do (i.e. they both are `schedulable` objects). Indeed, the two distinct concepts exhibit the same execution pattern from the scheduler's perspective. However, the RTSJ wants them tuned differently in terms of a performance profile. `RealtimeThreads` can be used to execute a small amount of code, but this may not be a good performance choice in terms of both memory and predictability [6]. The inten-

tion of the RTSJ is that `AsyncEventHandlers` do not suffer the same overhead as a `RealtimeThread` [11]. Therefore, it should not be the case that each `AsyncEventHandler` has a separate server `RealtimeThread`.

3. CURRENT AEH IMPLEMENTATIONS

Some implementations of the RTSJ such as the *RI* [10] and *OVM* use a simple scheme for AEH. The *RI* just creates a server for an `AsyncEventHandler` each time it is released [6] and the *OVM* bounds a server per `AsyncEventHandler`. These work perfectly but they cause a proliferation of server threads along with the associated per thread overhead when the `AsyncEventHandlers` are many and most of them are simultaneously active.

In this section we look at more efficient AEH mechanisms which actually take advantage of having a means of handling multiple `AsyncEventHandlers` by a server rather than the server-per-`AsyncEventHandler` basis. Firstly, the AEH of *jRate* [4] is presented and discussed. Then *Java RTS*'s AEH [8] is followed. After discussing them we present our own model of the AEH, the *monitor* model, and compare it with the other two techniques.

jRate's model.

The model uses a well-known architectural design pattern called *Leader/Followers* [4, 9]. Its main component is a single leader thread and a thread pool consisting of follower threads. The leader waits for a handler to be queued, at which point it starts to execute it. Right before the execution, it promotes a follower from the pool to become the new leader. Then the new leader waits for another handler to be queued. After electing a new leader, the former leader starts processing the handler with its state transformed into a processing state. Consequently multiple former leaders can execute handlers simultaneously while the newly elected leader waits for a new handler to arrive. When a processing thread completes execution (i.e. its `fireCount` goes to zero.) and there is a leader, the processing thread joins the pool with its role transformed into a follower. Otherwise the processing thread becomes the leader immediately upon completion. At any point in time, a thread's state can therefore be in one of the three states, leader, processing and follower.

Java RTS's model.

One of the particularities of this implementation is that the binding between a handler and a server thread is performed at execution time, not at the release time. When a handler is released, it is enqueued in a pending handler queue. After the queuing, the existence of the leader¹ thread is checked. If there is one and its priority is higher or equal to the handler's priority, no further action is required. If the leader thread's priority is lower than the handler's priority and it has not started to execute any handlers, then its priority is increased to the value of the handler's priority level. If no leader thread exists, a new one is selected from the thread pool, its priority is configured accordingly, and becomes the new leader thread. When the leader thread gets the processor, it binds itself with the highest priority

¹In the *Java RTS*'s AEH implementation, the leader thread is called the *ready* thread. We still use the term, leader, for the model to prevent confusion as they are essentially the same.

handler in the queue. Another particularity of this implementation is that it elects a new leader thread right before it starts the execution only if there is one or more handlers pending in the queue. Otherwise the leader thread starts to execute the handler without electing a new leader. At the end of the execution, the server checks if another handler is released and not bound yet. If it finds one, it will bind itself to this handler and will execute it. It can bind itself to a handler running at its current priority or find a handler with a lower priority, in this case its priority is automatically adjusted during the binding process. When no more released handlers are available, the server finally returns itself to the pool. The new leader thread ensures that it will continue the job if the server blocks.

Monitor model.

The primary objective of this model is that a server repeatedly executes multiple handlers changing its priority accordingly and dealing effectively with blocking handlers. The blocking will cause unbounded priority inversion [11] when a single server thread is used for the whole AEH system. The *jRate*'s AEH model does not have to be concerned about blocking handlers as it essentially allocates a server per handler. The *Java RTS*'s model makes another server ready when there are more than two pending handlers to take over the job if the server blocks. The monitor model uses a different mechanism to cater for blocking handlers. The monitor thread is used for this purpose along with a stack data structure called **serverStack**. The stack is used to store information about active server threads' priority levels from which they start to execute. The overall run-time behaviour of the monitor model can be described as follows:

- When an event is fired, the associated handlers are enqueued and the existence of the leader thread is checked. If there is no leader, a new thread is picked from the pool and becomes the new leader with its priority level put in the stack. If there is the leader, its priority is reconfigured except when the server is executing and the priority of the released handler is greater than that of the server. In such case, a new server is elected and made ready with the event's priority. When the leader is made ready in both cases, the priority of the leader is put in the stack.
- When the server starts executing, it takes the handler of the highest priority from the queue and executes it. Right before the server starts to execute it makes the monitor thread ready if the number of handlers in the handler queue is greater than the length of the stack which represents the number of active server threads in the system. Upon the completion, the server attempts to execute the next handler if there are pending handlers. Otherwise the server yields the processor. When it loops to execute another handler, it goes through the same process as when it starts to execute. This iteration lasts until the priority of the next handler is equal to the value of the second item in the stack (i.e. the next handler should be executed by another active server that has been preempted by the current server). In that case, the current server yields the processor and returns to the pool.
- There is a single monitor thread in the system. Its role

is essentially to replace a blocked server with another server from the pool to prevent unbounded priority inversion. The monitor thread is made ready only when the number of handlers in the queue is greater than the number of active servers (not in the pool). The rationale behind this is that we need another server thread only when all servers are blocked and there is one or more pending handlers in the queue.

4. FORMAL MODELS IN UPPAAL

UPPAAL is a toolbox for validation (via graphical simulation, simulator) and verification (via automatic model-checking, verifier) of real-time systems. It has been applied in many case studies from communication protocols to multimedia applications. In this paper, UPPAAL is extensively used to model, simulate and verify the AEH systems of two RTSJ implementation, *jRate* and *Java RTS*, and the proposed *monitor* model presented in Section 3. Readers are referred to the [1] for a more detailed explanation about the UPPAAL tool.

4.1 Modelling Architecture

We propose an architecture to specify AEH system using the automata formalism of the UPPAAL tool. The idea of the architecture is to decompose the functionalities of the AEH system into several different components. Each component is modelled in a single automaton:

- Environment (Env): This represents the environment that non-deterministically generates events that release associated handlers with one of three possible priority levels, high(3), middle(2) and low(1). Note that in the RTSJ every occurrence of an event increments the **fireCount** in each attached handler. In this paper, however, we assume that the **fireCount** never goes beyond 1 (i.e. an event releases different handlers even if it is fired twice or more).
- Handler Queue (HQ): This component represents a handler queue and operational methods on the queue for adding and removing handlers. The queue is priority ordered.
- Server Thread(ST): This is responsible for the representation of four states that server threads can have, namely Idle, Ready, Executing, Blocked. When it is allowed to execute, it may block non-deterministically. A ST also can be preempted when there is a higher priority ST in the run queue of STs.
- Thread Pool (TP): This represents the pool of server threads. This component is responsible for managing servers preallocated in the pool at the initialisation phase. Server threads can dynamically join the thread pool and become a server. In the UPPAAL models, we have servers twice as many as the priority level in the system (i.e. if the priority level is set to 3, 6 server threads created and put in the thread pool). This is because a server may block during the execution and therefore it is necessary to have additional servers to be replaced with blocked ones. Of course it is also possible all the server threads may block. In such cases, it is necessary to create a new server thread at run-time. However this is not the concern of this paper.

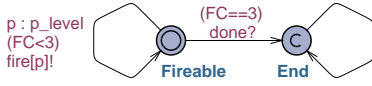


Figure 2: Env in the jRate AEH

- Base Scheduler (Scheduler): This component represents the basic functions and interactions of the base scheduler, PriorityScheduler of the RTSJ. The Scheduler allows a ST to run, yield, be preempted, be blocked and unblocked. Also it checks the current status of the AEH system and is responsible for activating and deactivating server threads accordingly.

4.2 UPPAAL Automata Models

In this section we presents the automata models for the two AEH systems used in *jRate* and *Java RTS*. Our proposed design for the AEH, the *monitor* model, is then given. The models have been designed with a number of modelling patterns recommended in [1] to speed up the verification. The patterns used in the models include variable reduction, synchronous value passing, and atomicity.

4.2.1 jRate's AEH

Environment (Env).

In Figure 2, the automaton representing the environment that randomly generates events regardless of the receiving automaton's status is presented. When an event is fired, the occurrence of the event with the associated handler's priority level is informed to Scheduler². Note that the transition **Fireable** \Rightarrow **End** occurs when the fire count (**FC**) reaches a maximum of 3 event firings. This allows us to explore 27 different combinations of event firings along with 3 priorities, low(1), middle(2), high(3). However the value of **FC** will vary from 3 to 6 when verifying the run-time behaviour in Section 5. After the transition, the whole model stops and no more transitions are allowed except for **End** \Rightarrow **End** (i.e. **End** is a committed location). This is to limit the number of event firings so that we could have a feasible boundary for verifying the model. Otherwise the verifier easily run out of memory due to too many generated states.

Server Thread (ST).

This represents the main behaviour of server threads, depicted in Figure 3. They can be one of the following four locations at any time:

1. **Idle**: refers to the state where it is created but has not been made **Ready**.
2. **Ready**: refers to the state where it could be selected to have its state changed to executing.
3. **Exec**: refers to the state where it is currently running on a processor. When it is changing its state from **Exec**,

²The self-looping edge with the channel **fire[p]!** in the location **Fireable** has a selection annotated (**p : p_level**). The edge will non-deterministically bind **p** to an integer in the range set by **p_level**. For the UPPAAL models, **p_level** is set to be [1,3]. Therefore **p** can be an integer in the range 1 to 3, inclusive.

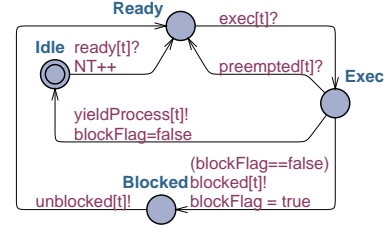


Figure 3: ST in the jRate AEH

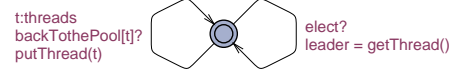


Figure 4: TP in the jRate AEH

the state proceeds to one of either states, **Ready** or **Blocked**. The **Ready** state could be reached when it is preempted by a higher priority server and the **Blocked** state may be reached for some reason (i.e. waiting for I/O etc.).

4. **Blocked**: refers to the state where it is not among those threads which could not be selected to have their state changed to executing.

In the automata models, 6 STs are created and put into the thread pool when the system starts. STs are, in turn, picked to be a leader (server) thread. Note that, a boolean variable **BlockedFlag** is introduced in the ST automata to indicate whether it is blocked. **BlockedFlag** is set to true when the ST is blocked and set to false when the ST yields the processor. This limits the number of blockings of STs to at most once per cycle. Also integer variable, **NT**, is used to keep track of the number of servers invoked from the pool as increased by 1 whenever a ST takes a transition from **IDLE** to **READY**. This variable will give us the maximum number of server threads used to handle a given number of event handlers when verifying the run-time properties (See Section 5).

Thread Pool (TP).

This automaton simulates the thread pool, depicted in Figure 4. In its local declaration, a queue structure called **threadPool** is defined along with operational methods such as **putThread()** and **getThread()**. The automaton manipulates the queue by using the methods at run-time. The initial location has two outgoing edges, **elect?** and **backToThePool[t]?**. The former transition is triggered to elect a new leader and the latter transition is invoked to put the yielding thread, **t**, back to the thread pool. When the channel synchronisation **elect?** is triggered, it elects a leader and sets the global variable **leader** to the value obtained by invoking the method **getThread()**.

Scheduler.

This is the automaton represents the base scheduler of the RTSJ (i.e. PriorityScheduler) depicted in Figure 5. Essentially this automaton has seven outgoing edges from the initial location, synchronising with other automata (i.e. **Env** and **TP**). The transitions are explained as follows:

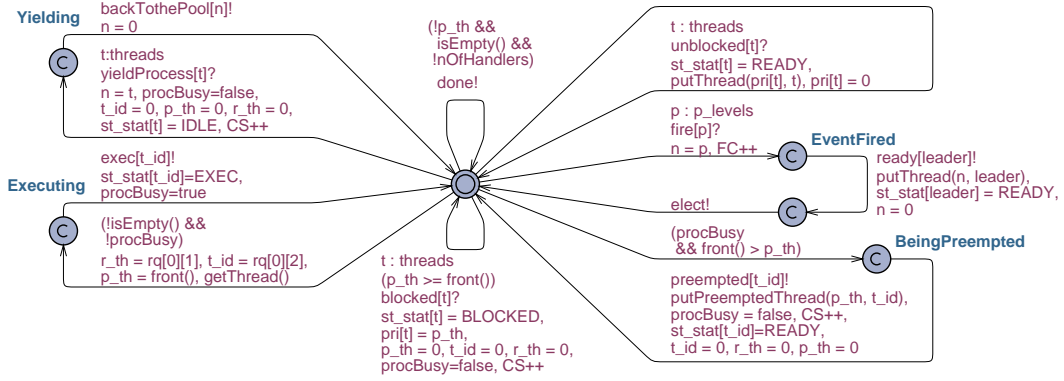


Figure 5: Scheduler in the jRate AEH

1. The transition with the channel `fire[p]?`: The first edge of this transition is triggered by `Env`'s firing an event. The `Scheduler` updates the variable `n` to the value of the released handler's priority level and increments the number of fire count (FC) by one. It then makes the leader ready by synchronising with the `ST(leader)` over the channel `ready[leader]!`. The last edge of this transition is to elect a new leader over the channel `elect!` that synchronises with the automaton TP.
2. The transition to the location **Executing**: This is guarded by the constraints `!isEmpty()` and `!procBusy`, which corresponds to the state where the highest priority thread is allowed to execute when the run queue of server threads is not empty, and when the processor is idle. The run queue of threads is always kept in a priority-ordered manner. Therefore the `Scheduler` always executes the highest priority thread first. On the first edge of the transition, three integer variables, `p_th` (thread priority), `r_th` (thread role) and `t_id` (thread id), are updated accordingly by taking the first thread out from the queue. These variables are used throughout the execution of the thread, especially to identify which thread executing with which priority level. Then the next edge from the location **Executing** to the initial location immediately takes place with the channel `exec[t_id]!`. The edge also updates the status of the system accordingly (i.e. `procBusy` is set to true and `st_stat[t_id]` representing the four states of thread with `t_id` is set to **Exec.**).
3. The transition with the channel `yieldProcess[t]?`: When a `ST` yields the processor, it synchronises with the `Scheduler`. This is done by the channel synchronisation `yieldProcess[t]`. The transition updates all the relevant information such that `procBusy` is set to false, `st_stat[t]` is set to **IDLE** and so on. The following outgoing edge with the channel `backToThePool[n]!`, synchronising with TP takes place immediately. The edge put the yielding thread into the pool.
4. The transition to the location **BeingPreempted**: As soon as the guard on the outgoing edge to the location **BeingPreempted** is satisfied, which corresponds to the state that there is a higher priority thread ready in the run queue while a lower priority server occupies

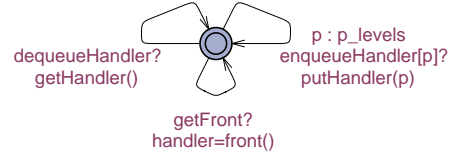


Figure 6: HQ in the Java RTS AEH

the processor, the edge is triggered. The outgoing edge from the location **BeingPreempted** is immediately performed with updating the status of the model. Note that, the method `putPreemptedThread()` puts the thread being preempted at the front of all the threads with the same priority in the run queue.

5. The transition with the channel `blocked[t]?`: When a `ST` is executing, it can be blocked and synchronises with the `Scheduler`. The edge updates the status of the model accordingly as depicted in Figure 5.
6. The transition with the channel `unblocked[t]?`: When a `ST` is blocked, it can be ready again after a certain amount of time and synchronises with the `Scheduler`. The edge updates the status of the model accordingly as depicted in Figure 5.
7. The transition with the channel `done!`: This transition which synchronises with `Env` is designed in the `Scheduler` to bound the number of the fire count (FC) and provide well-defined behaviour of the system (i.e. when FC reaches the predefined value, referred to Figure 2, all the handlers are served and there are no threads pending in the run queue, the transition will take place).

4.2.2 Java RTS's AEH

Here we discuss only different features used in *Java RTS*. Handler queue (HQ) is firstly presented. Then `Scheduler` will follow. Other automata used in *Java RTS* such as `Env`, `ST` and `TP` are the same as in the *jRate*'s AEH.

Handler Queue (HQ).

This component is modelled in the automaton HQ in Figure 6. This automaton is absent from the *jRate* AEH model.

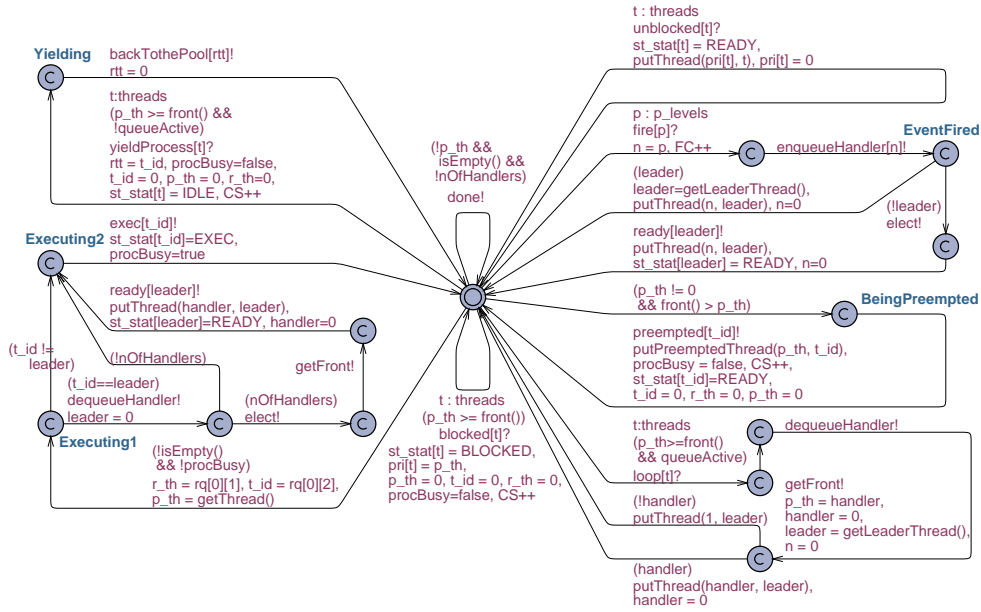


Figure 7: Scheduler in the Java RTS AEH

This is due to the assumption that there are always available follower threads in the pool and the *jRate* AEH system enqueues a handler only if there are no available threads in the pool. In the HQ's local declaration, a queue structure (hq) has been declared along with operational methods such as `getHandler()`, `putHandler()` and `getFront()`. The automaton manipulates the queue by using the methods at run-time except `getFront()` which returns the head of the queue without removing it. It has three outgoing edges with channel synchronisations, `dequeueHandler?`, `enqueueHandler[p]?` and `getFront?`. As the names of channels convey, it updates the queue along with their respective methods.

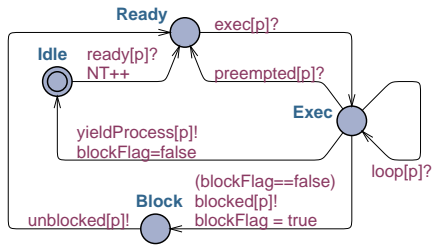


Figure 8: ST in the Monitor Model

Server Thread (ST).

The automaton represents a server thread in the *Java RTS* model, depicted in Figure 8. It has two distinct difference from the one used in the *jRate* model. First, it has an additional array variable called `threadActive[p]` for the purpose of letting the models know whether it is blocked or not. Second, it has a self-looping edge with the channel `loop[p]` in `Exec`. This is to simulate the behaviour that a server loops itself to execute another pending handler in HQ when it finishes executing the current handler.

Scheduler.

The automaton in Figure 7 represents the Scheduler for the *Java RTS*'s AEH system. The automaton also has seven outgoing edges from the initial location, synchronising with other automata. Some transitions are identical with ones presented in Figure 5. We only discuss different edges from ones used in the *jRate*'s Scheduler

1. The transition with the channel `fire[p]?`: Unlike the *jRate*'s Scheduler, it enqueues the released handler in the handler queue by synchronising with the HQ over the channel `enqueueHandler[n]!`. Then it evaluates the integer variable `leader`. If the `leader` is greater than 0, the Scheduler takes the `leader` out from the thread run queue and enqueues it with adjusted priority. Otherwise, it elects a new server, makes it ready, and updates the system status (i.e. `st_stat[leader] = READY`).
2. The transition to the location `Executing1`: The first edge of this transition is identical with the one in *jRate* AEH. In the next step the Scheduler evaluates whether the variable `leader` equals to the variable `t_id` which is used to hold the id of the thread about to execute. If they are not equal, the Scheduler executes the thread with the channel `exec[t_id]!` from the location `Executing2`. The edge updates the status of the system accordingly (i.e. `procBusy` is set to true and `st_stat[t_id]` is set to `Exec`). Otherwise, the first handler in the handler queue is dequeued for execution. Now the Scheduler measures the variable `nOfHandlers` that holds the number of handlers in the handler queue. If `nOfHandlers` equals to 0, it proceeds to the location `Executing2`. Otherwise, it elects a new `leader` and adjusts its priority to the priority level of the handler at the head of the handler queue and executes the former leader.
3. The transition with the channel `loop[t]!`: The first

edge in the transition is guarded by the constraints $p_th \geq \text{front}()$ and $\text{queueActive} == \text{true}$, which corresponds to the state where the priority of the currently executing thread is eligible for continuing to execute another handler pending in the queue. It then dequeues the highest priority pending handler from the queue and bounds it to the server. The priority of the leader thread in the run queue is adjusted to the value of the highest priority of the pending handlers. If there is no pending handler then the priority of the leader will be set to the lowest priority.

4.2.3 Monitor model

Although the monitor thread model is different from the previous two models in terms of how it deals with blocking handlers, most of the structural components it comprises are similar. Here we only present different or additional parts of the model compared to the above two. Obviously it has the additional automaton which represents the monitor thread and the Scheduler is also presented to show the different behaviour in the monitor model.

Monitor thread (MT).

This automaton, depicted in Figure 9, represents the monitor thread that is also a underlying real-time thread. However the Scheduler treats it differently as it has the specific mission to accomplish. A particularity of the monitor thread is that it is made atomic once it starts to execute. Therefore it can neither be preempted nor blocked. This is because firstly we know it does not block and second we assume that the time taken to replace the blocked server thread is negligibly small³.

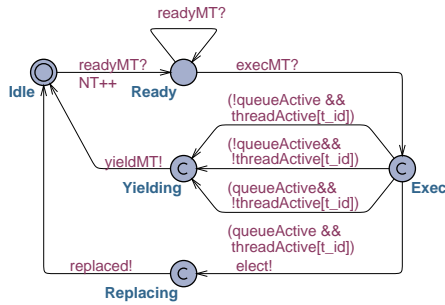


Figure 9: MT in the Monitor Model

When the Scheduler makes the monitor ready, it synchronises with the MT over the channel **readyMT?**. When the monitor thread starts executing in **Exec**, it checks two boolean variables, **queueActive** and **threadActive[t_id]**. If both variables are evaluated true, it replaces the thread whose id is set to **t_id** with a new server from the pool. Otherwise, it just yields the processor. The edge that loops itself in **Ready** is needed for the situation where the MT is made ready and the Scheduler tries to reconfigure the priority of the monitor thread. This is recurring when a server thread completes its execution without blocking. In such cases the

³The atomicity is optional. In an implementation, however, atomicity can be achieved by entering a critical region protected by some mutual exclusion mechanism [3], such as a semaphore or a monitor.

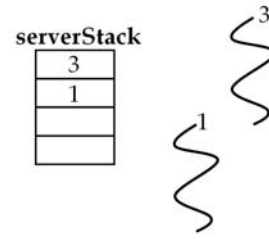


Figure 11: The serverStack in the Monitor model

Scheduler will take the MT out from the run queue and reenqueue it with its priority lowered to the lowest priority level over the channel **ReadyMT!**.

Scheduler.

In Figure 10, the automaton which represents the Scheduler for the *monitor* model is depicted. Only different transitions from ones explored previously are presented below.

1. The transition with the channel **fire[p]?**: The different behaviour of the transition appears when it evaluates **leader** as **true**. Otherwise it proceeds as done in Macinac's. If there is a leader, the Scheduler examine the state of the leader by evaluating the array variable **st_stat[leader]**. If the leader is **IDLE** or **READY**, the Scheduler takes the leader out from the queue and reenqueuees it with the adjusted priority as done in the Java RTS's AEH model. One of the different features of the model is that it maintains the additional stack data structure, called **ServerStack**, to keep track of priority levels with which servers should be allowed to execute. When reconfiguring the leader's priority, it puts the value of the priority level in the stack⁴. The Figure 11 illustrates a snapshot of the system at runtime, corresponding to the state where there are two active servers with their priorities to start executing with is 3 and 1 respectively. Curved lines with a number at the head of it indicate server threads and their respective priorities. Therefore the server thread with priority 3 is allowed to execute handlers in a priority range, 3 to 2. The server with priority 1 is eligible for the execution of the handlers with priority 1. The next edge evaluates the variable **leader**. If the leader is **EXEC** and the priority of the leader (held in **p_leader**) is smaller than that of the handler (held in **n**), it elects another leader thread with the priority of the released handler.
2. The transition to the location **Executing**: A specific characteristic of this automaton compared to ones that used in the previous two models is that the location **Executing** has two outgoing edges and the transition is taken depends on the role of the thread, indicated by the integer variable **r_th** (i.e. if the **r_th** is equal to 1, this indicates that the current thread's role is as the monitor thread.).

⁴This is to cater for the situation where there are multiple active servers. The information the stack holds about active servers in the system is then used by the Scheduler to determine the points in time when the currently executing server should yield the processor and give lower priority ready servers a chance to execute.

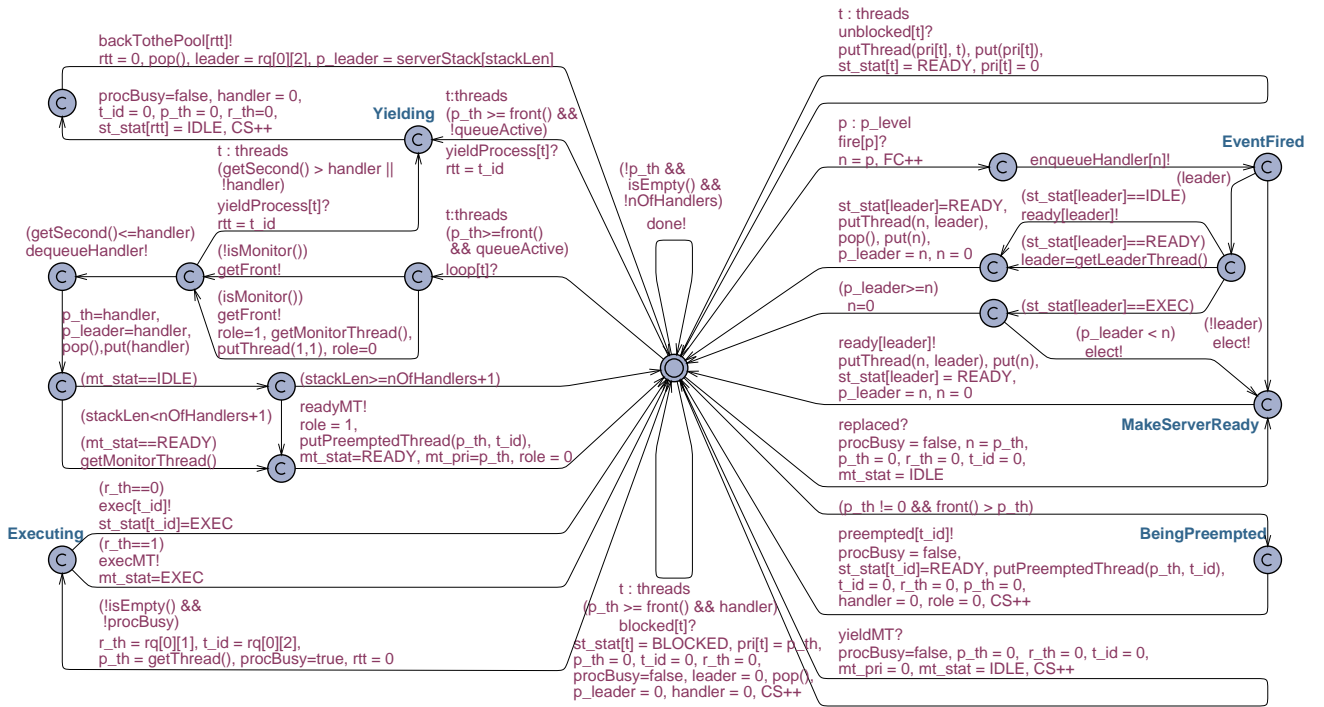


Figure 10: Scheduler in the Monitor Model

3. The transition with the channel **yieldMT?**: This is additional automaton used by the automaton MT to yield the processor.
4. The transition to **MakeServerReady**: When the MT replaces the blocked server with another server from the pool, it synchronises with the Scheduler over the channel **replaced?**. Then the Scheduler makes the newly elected server ready and puts it in the run queue.
5. The transition to **Yielding**: This transition is slightly different from ones used in the other two AEHS. The effect of the transition is the same. However, the first edge has been divided into two edges so that the middle location **Yielding** should be allowed to have the other incoming edge from the transition with the channel **loop[t]?**.
6. The transition with the channel **loop[t]!**: The first edge in the transition is guarded by the constraints $p_th \geq \text{front}()$ and $\text{queueActive} == \text{true}$ as it is in the *Java RTS* AEH. Then it retrieves the highest priority handler with the channel **getFront!** (referred to Figure 6) and also if the next thread to run is the monitor (i.e. by invoking the method **isMonitor()** which returns true if and only if the first thread in the run queue is the monitor), then the server lowers the priority of the monitor to the lowest priority level (1). It next measures the second value in the stack by **getSecond()** method, which gives the next server's priority level. If the second value is greater than the value stored in **handler**, then it proceeds to location **Yielding**. Otherwise it dequeues the handler from **hq** and adjusts the server's priority as well as changing the value of **ServerStack**'s first item accordingly.

The next step of this transition depends on the monitor's state. The Scheduler enqueues the MT only when the number of active servers is greater than the number of handlers pending in **hq**, expressed in the edge with the channel **ReadyMT!**. If the monitor is already in **Ready** state, the priority of the monitor is reconfigured accordingly.

5. FORMAL ANALYSIS OF THE MODEL

In this section we specify several properties for the models defined in Section 4. The query language used in the paper is a simplified version of CTL for UPPAAL. Readers are referred to the [1] for a more detailed explanation about the query language for UPPAAL. Simple reachability properties verified successfully for the models are not listed here. The properties presented here can be classified into three criteria as follows:

- Model Consistency to guarantee that the models comply with the RTSJ
- Model Safety to guarantee that the models are free from unwanted behaviour
- Run-time Behaviour to explore and test the run-time behaviour of the AEH models

Model Consistency.

- *Property 1. A thread with the higher priority handler will always execute in preference to a thread with the lower value handler when both are eligible for execution.*

```
procBusy==true && Scheduler.Executing
→ p_th >= Scheduler.rq[0][0]
```

- **Property 2. The currently executing thread will be immediately preempted by a higher priority ready thread.**

```
procBusy==true && Scheduler.BeingPreempted
→ p_th < Scheduler.rq[0][0]
```

- **Property 3. The logical release of an attached handler may occur before the previous release has completed.**

```
E<> p_th==2 && Scheduler.EventFired &&
Scheduler.n==2
```

Model Safety.

- **Property 1. The AEH models are guaranteed free from a deadlock.**

A[] not deadlock

- **Property 2. There is none or one server thread running on the processor at any point in time.**

A[] ST(1).Exec + ST(2).Exec + ST(3).Exec +
ST(4).Exec + ST(5).Exec + ST(6).Exec <= 1

Note that in the monitor thread model this property should reflect the use of the monitor thread. Therefore it becomes as follows:

A[] ST(1).Exec + ST(2).Exec + ST(3).Exec +
ST(4).Exec + ST(5).Exec + ST(6).Exec + MT.Exec
<= 1

Run-time Behaviour.

This set of properties has been tested with the edge with the channel `blocked[p]` in ST automata disabled. Otherwise an ST may block and unblock an infinite number of times during its execution. Therefore it is reasonable to bar the way to the blocking to have the feasible boundary of the models.

- **Property 1. A server that is preempted by a higher priority server is placed at the front of the run queue.**

```
Scheduler.EventFired && Scheduler.n == 6
→ HQ.list[0] == 6
```

- **Property 2. When an event occurs its attached handler is released for execution.**

```
Scheduler.BeingPreempted && RTT(6).Exec
&& p_th ==3 → Scheduler.rq[0][2] == 6
```

- **Test 1. The maximum and the minimum number of threads required to serve a certain number of handlers.**

E<> NT == i && Env.End

Models	FC	Maximum i	Minimum i
jRate	3	3	3
	4	4	4
	5	5	5
	6	6	6
Java RTS	3	3	2
	4	4	2
	5	5	2
	6	6	2
Monitor	3	3	1
	4	4	1
	5	5	1
	6	6	1

Table 1: AEH Models' Results for the number of server threads (NT) used as the fire count (FC) increases

The variable i is replaced with a natural number when verifying the property. The results from the three models are shown in Table 1. To observe how many servers required for a certain number of handlers we vary the variable **FC**, that holds the number of the fire count for the whole AEH system, from 3 to 6 (See Figure 2). As mentioned earlier, the automata models does not specify the time constraints and therefore the simulator will give us the worst and the best event firing combinations for the specified event firing numbers.

The AEHs for *jRate* and *Java RTS* require the same number of servers as the number of handlers fired regardless of timings of event firings. As the number of handlers increases, the number of servers required is increasing. We observed better results from the *Monitor* model. The maximum number of servers required for the model is the same as the other two models. However, the minimum number is 1 irrespective of the numbers of fired handlers. The best case of the model is observed if events are fired in turn (i.e. an event is fired after the previous event completes.).

- **Test 2. The maximum and the minimum number of context-switches incurred to serve a certain number of handlers.**

E<> CS == j && Env.End

Again the variable j is replaced with a natural number when verifying this property. The results from three models are shown in Table 2. To observe how many context-switches incurred for a certain number of handlers we vary the variable **FC** (3 to 6). The maximum number of the context-switches are all the same for the models and increases as the number of handlers increases. This is because the context-switches are incurred when a thread yields or is preempted. The worst case number of preemption will occur when the low priority thread is released first and other events are released in a priority-ordered manner before the event completes. Therefore the worst case number of context-switches is:

$$CS = \sum_{j \in hpt(i)} \left\lceil \frac{R_i}{T_j} \right\rceil + n$$

Models	FC	Maximum j	Minimum j
jRate	3	5	3
	4	7	4
	5	9	5
	6	11	6
Java RTS	3	5	2
	4	7	2
	5	9	2
	6	11	2
Monitor	3	5	1
	4	7	1
	5	9	1
	6	11	1

Table 2: AEH Models' Results for the number of context-switches (CS) used as the fire count (FC) increases

where $hpt(i)$ is the set of higher priority threads than i , R_i is the worst case response time of the thread i , T_j is the period of the thread j [3] and n is the number of active servers. Hence if there are 6 handlers released, the worst case context-switches is 11. This applies to all the models. However fewer number of context-switches is observed in the *monitor* model. Again the minimum number of the context-switches for the *monitor* model is observed when events are fired in turn (i.e. an event is fired after the previous event completes.).

6. CONCLUSIONS

This paper used the UPPAAL tool to evaluate the AEH of different RTSJ implementations, *jRate* and *Java RTS* along with our own AEH model for the RTSJ, the *monitor* model. The three automata models have been shown to be consistent with the RTSJ and the properties including reachability, safety and liveness have been verified successfully. The run-time behaviour of the models have generated some measurements for resource usage and overheads. Although much work remains to ensure predictable and efficient AEH implementation under heavy workloads and high contention, our test results indicate that the two AEHs implementations could be optimised on average to use a fewer number of servers. The *monitor* model is an example of such an optimised model and on average requires smaller number of servers and context-switches. The *monitor* model, therefore, would be best suited for high-volume embedded systems such as mp3 players and mobile phones, which require production costs as low as possible. Less servers and context-switches will result in staying within on-chip memory limits (typically 512 bytes to 4 K RAM) [5]. In addition to this, indirect cost of context switches such as performance degradation due to cache sharing can be reduced.

As future work, we intend to extend the models to have time constraints. By doing so, we expect to have the exact run-time behaviour based on a certain sequence of event firings. Also we initially assumed that there are always available servers. In reality, this assumption is not always true and in such cases the AEH implementations create a new server at run-time, requiring significant system resource. Another issue associated with the AEH is to have a matrix that indicates the number of servers for a certain number of handlers. This is required as some implementations, such as

Java RTS, allow the programmer to specify the number of servers at initialisation. The complexity of the scheduler in the *monitor* model is higher than the two other implementations. This is mainly due to its higher volume of run-time system information required for the *monitor* model to function correctly. Therefore it is essential to implement the design with an RTSJ implementation for it to be justified. Hence, we intend to implement the optimised model with an open source RTSJ, such as *jRate*.

7. ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of Frederic Parain and Bertrand Delsart to some of the work presented in this paper.

8. REFERENCES

- [1] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. *Lecture Notes in Computer Science*, 3185:200–236, 2004.
- [2] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. Real-Time Specification for Java. <http://www.rtsj.org/>.
- [3] A. Burns and A. J. Wellings. *Real-Time Systems and Their Programming Languages*. Addison Wesley, UK, 2001.
- [4] A. Corsaro. *Techniques and patterns for safe and efficient real-time middleware*. PhD thesis, Washington University, St. Louis, MO, USA, 2004.
- [5] R. Davis, N. Merriam, and N. Tracey. How Embedded Applications Using an RTOS can stay within On-chip Memory Limits. In *12th EuroMicro Conference on Real-Time Systems*, pages 71–77, 2000.
- [6] P. Dibble. *Real-Time Java Platform Programming*. Sun Microsystems, California, USA, 2002.
- [7] P. Dibble and A. J. Wellings. The Real-Time Specification for Java: Current Status and Future Direction. In *7th International Conference on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 71–77, 2004.
- [8] F. Parain. Asynchronous Event Handling in Java RTS. Private Communications, March 2007.
- [9] D. C. Schmidt and C. O’Ryan. Leader/Followers - A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching, 2000.
- [10] TimeSys. Real-Time Specification for Java Reference Implementation. www.timesys.com/java.
- [11] A. J. Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, UK, 2004.
- [12] A. J. Wellings and A. Burns. Asynchronous Event Handling and Real-Time Threads in the Real-Time Specification for Java. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, page 81, Washington, USA, 2002. IEEE Computer Society.
- [13] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A Design Framework for Highly Concurrent Systems. Technical Report UCB/CSD-00-1108, EECS Department, University of California, Berkeley, Aug 2000.