A predictable Java profile - rationale and implementations

Thomas Bøgholm, René R. Hansen,
Anders P. Ravn, and Bent Thomsen
Department of Computer Science
Aalborg University, Denmark
{boegholm,rrh,apr,bt}@cs.aau.dk

Hans Søndergaard
VIA University College
Horsens, Denmark
hso@viauc.dk

ABSTRACT

A Java profile suitable for development of high integrity embedded systems is presented. It is based on event handlers which are grouped in missions and equipped with respectively private handler memory and shared mission memory. This is a result of our previous work on developing a Java profile, and is directly inspired by interactions with the Open Group on their on-going work on a safety critical Java profile (JSR-302). The main contribution is an arrangement of the class hierarchy such that the proposal is a *generalization* of Real-Time Specification for Java (RTSJ). A further contribution is to integrate the mission concept as a handler, such that mission memory becomes a handler private memory and such that mission initialization and finalization are scheduled activities. Two implementations are presented: one directly on an open source JVM using Xenomai and another, based on delegation, on an RTSJ platform.

Categories and Subject Descriptors

C.3 [**Special Purpose and application-based systems**]: Real-time and embedded systems; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Real-Time, Java

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [8] was a major breakthrough for Java as a language for programming embedded software, but soon after it had emerged, the discussion began as to whether it was too large or too dynamic to really support high integrity applications. This led to proposals for smaller profiles with a rationale presented in [27] and an implementation in the Ravenscar Java profile [25]; the focus was on embedded systems and on making programs amiable to analysis with state-of-the-art techniques. These formed a starting point for our own work with a predictable Java profile [32, 29] which considered using integrated analysis tools instead of programmer supplied parameters to provide predictable programs. More recently the Open Group has formed a committee to develop a profile for Safety-Critical Java [17, 4], and they have outlined their approach in a recent paper [20]. During the work in the committee we have had access to intermediate drafts which we have commented. This paper is a summary and consolidation of points where we find that the current draft may improve.

We have found the Open Group's approach refreshing, because they solve a major issue by settling for handlers as the schedulable entities in a real-time system. RTSJ supports both a handler paradigm and a thread paradigm, but the latter is hampered by inheritance from Java threads, for instance with unwanted asynchronous interrupts and conditional waits. Threads are not really suitable as logical processes. The handler concept is much closer in spirit to a logical process. In this, as in many other details, we agree with the SCJ draft.

The points where we would see further advances, and thus the contributions of this paper are:

1. An arrangement of the class hierarchy such that the proposal is a *generalization* of RTSJ, as we believe it should be, because it has far fewer details and options. The SCJ draft uses *specialization*.

2. The SCJ draft organizes handlers in missions. We propose to make missions first-class handlers, such that initialization, termination and transition between missions may be given a precise semantics. Furthermore it makes mission memory equal to handler memory.

In Section 2 we elaborate on constructs that make missions first-class schedulable entities which gives a structured replacement for the Scheduling Groups of RTSJ and the Thread Groups of Java. These concepts form the core of our proposal for a Predictable Java (PJ) profile[1].

It is evident that for practical reasons any viable profile must be compliant with RTSJ, but we observed that defining a profile by *specializing* or subclassing RTSJ leads to much clutter. That this must be so is rather clear if one considers

[1]We have avoided to name the profile SCJ in order to avoid confusion, but if the main ideas are taken up by SCJ, PJ has served its purpose.

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'09, September 23-25, 2009 Madrid, Spain
Copyright 2009 ACM 978-1-60558-732-5/09/09 ......$10.00.

which profile really extends the other: RTSJ is a very flexible and detailed theory, whereas what we are searching for is an abstraction or *generalization* of it. Thus in an ideal world, RTSJ would be a specialization or refinement of a smaller profile. Doing it the other way around is like deriving the class of natural numbers from a class of rational numbers: sign inversion would need to be disallowed, division would become a partially defined operator, subtraction likewise, etc. Yet, the world is not ideal, so in a first step, we define a profile by manually abstracting from RTSJ classes and interfaces for relevant entities. Then a few specific classes are introduced to deal with handlers and missions. This gives a very compact and orthogonal organization of a package for the profile which is explained in Section 3. For example, the RTSJ-class `AsyncEventHandler` has 7 constructors and 32 methods. In PJ, we need of those only 1 constructor and 6 methods in the superclass `ManagedEventHandler` - a considerable reduction. Another benefit compared with the *specialization* approach is that we avoid annotations like `@SCJAllowed` annotations to prohibit unwanted methods.

An application that uses the PJ package may compile and, given that semantics is preserved, run under RTSJ with an adapter layer that through delegation disallows some RTSJ methods and give default values for some parameters. This is demonstrated in Section 4. Furthermore we outline a more direct implementation with a modified JamVM [23] on top of Xenomai [15] and Linux.

Finally, Section 5 investigates what kind of static constraints are needed to make the profile truly predictable and what tool support would be feasible for checking the constraints; and in Section 6 we conclude.

2. KEY CONCEPTS

In this section we introduce the key concepts in the Predictable Java profile: handlers and mission and the supporting resource concepts of memory and schedulers. The interplay between resources, handlers and missions determine predictability of applications.

2.1 Handlers

An application programmer must have the means to define temporal scopes [10] (period, deadline, execution time budget) and the specific algorithm for handling a periodic or aperiodic event triggered computation when developing real-time applications. This is succinctly encoded as a periodic or aperiodic event handler.

A periodic event handler in an application is a specialization of the `PeriodicEventHandler` of the profile, see Section 3.

```
class Periodic extends PeriodicEventHandler
{
  protected Periodic(PriorityParameters priority,
                     PeriodicParameters pp,
                     Scheduler scheduler,
                     MemoryArea memory)
  { super(priority, pp, scheduler, memory); }

  public void handleEvent() {
    // the logic to be executed every period
  }
}
```

It defines the temporal scope in the parameter `pp`. The `priority` parameter is for use by a `scheduler` and the handler has a `memory` to be used during execution of the algorithm. These parameters are concerns of the system programmer that assembles the application; the programmer of individual tasks focuses on giving an algorithm that specializes the `handleEvent` method.

The semantics is a periodic execution of the algorithm with the given period within its deadline which both are specified in the value `pp` of the `PeriodicParameters` object. The semantics is conditional on the algorithm completing within its execution time budget, included in `pp`, without declaring more temporary objects than can be accommodated in the `memory`, and refraining from declaring non-temporary objects. These semantic conditions can be checked by conservative approximation using abstract interpretation. This is discussed further in Section 5.

The aperiodic event handler is very similar:

```
class Aperiodic extends AperiodicEventHandler
{
  protected Aperiodic(PriorityParameters priority,
                      AperiodicParameters ap,
                      Scheduler scheduler,
                      MemoryArea memory)
  { super(priority, ap, scheduler, memory); }

  public void handleEvent() {
    // the logic to be executed when an event occurs
  }
}
```

It defines the deadline and cost, in the parameter `ap`. The remaining parameters are analogous to the ones for a periodic handler. The logic is given by specialization of the `handleEvent` method.

The semantics is an activation and execution of the algorithm within its deadline when an event is pending on an `Event` object to which the handler is attached. As for periodic handlers, the semantics is conditional on the algorithm completing within its execution time budget, without declaring too many temporary objects and refraining from declaring non-temporary objects.

A further condition is that event occurrences are sufficiently separated. A safe interpretation is that they are separated at least by the specified deadline. However, this may be unduly pessimistic, for instance with a shutdown handler, thus we have considered adding a minimal interarrival time to the parameters, as in RTSJ `SporadicParameters`. The troublesome point is whether this is statically checkable. For external events, this is clearly not possible; they must be assured by assumptions about the environment. However, for internal events, model checking based tools like TIMES [3] are able to combine analysis of event passing with schedulability analysis, again assuming some conservative approximation of the actual algorithm.

A bolder interpretation of aperiodic events is that they are just indistinguishable ticks, counting them is sufficient, and with a `long` counter, there should be space for even the most

lively interrupt generators. It is then up to the application to handle and reset the counter. Such liberal semantics could be considered.

2.2 Missions

The functionality of an application is made up of handlers; but they have to be executed according to a feasible schedule implemented by a scheduler. Handlers use private memories for temporary objects, but they may use more permanent shared objects protected through mutual exclusion mechanisms as specified by the Java `synchronized` method qualifier. These are placed in the memory of the mission. The embedded software systems programmer designs the architecture in terms of missions that encapsulates a set of handlers.

A mission is essentially a set of tasks that collaborate on providing a desired functionality. When application functionality changes over time - mode transitions - there are multiple missions. In very simple cases there is only one mission, which may need an initialization and termination. In more complex cases, missions may compose sequentially, conditionally or even in parallel, from which there is but a small step to having statically nested missions.

Already initialization and termination indicates that a mission is more than a simple container for a set of handlers. It is itself a *handler* for termination or in some cases an initialization event. Thus we collect the responsibilities of a mission in a handler class that contains handlers.

The alternative, the `Mission` as a simple container for handlers, we found, would further complicate the profile: a new class hierarchy is introduced along with special mission memory, and even new types of handlers for mission start, stop, initialize etc. might be introduced, to handle missions, and possibly more. This does not contribute to the framework since what is needed to express the mission concept already exists in handlers: private handler memory, used as mission memory, the handler concept allows functionality to be expressed, such as start, stop etc.

The code below gives the termination handler aspect of the mission in the `handleEvent` logic. Initialization is done in the constructor of a mission, where the container aspect is the vector of `eventHandlers`. Individual handlers belonging to the mission are added by the `addToMission` method. Note that handlers can be added only, and only during initialization, thus a mission contains a static and finite set of handlers. One could consider replacing the dynamic vector with a simple array since the length is known at initialization time.

```
public class Mission extends AperiodicEventHandler
{
  Vector<ManagedEventHandler> eventHandlers;

  protected Mission(PriorityParameters priority,
                    AperiodicParameters ap,
                    Scheduler scheduler,
                    MemoryArea memory)
  {
    super(priority, ap, scheduler, memory);
    eventHandlers = new Vector<ManagedEventHandler>();
  }

  public void addToMission(ManagedEventHandler eh)
  { eventHandlers.add(eh); }

  ...

  public void handleEvent() {
    // the logic to be executed to terminate a mission
  }
}
```

The elided part denoted by the ellipsis above are methods that interact with the scheduler.

```
  public Vector<ManagedEventHandler> getEventHandlers() {
    return eventHandlers;
  }

  public boolean add() {
    return getScheduler().add(this);
  }

  public boolean remove() {
    return getScheduler().remove(this);
  }
```

A mission is submitted to its scheduler by `add`. The scheduler knows that the calling handler is a `Mission` and contains a list of handlers. They can be accessed through `getEventHandlers`, and it is then up to the scheduler to schedule the set if it is feasible. The handlers of a mission are assumed to be started from a common start time.

Correspondingly, as a step in termination, the set may be removed from the scheduler through `remove`. There are several possibilities for a termination semantics. These are discussed below under schedulers in subsection 2.4. The method `getScheduler` is defined in the superclass, `ManagedEventHandler`.

2.2.1 Mission examples

A basic mission contains periodic handlers without any termination. It uses a cyclic scheduler declared in `main` and has a Linear Time `LTMemory` at its disposal. Since it never terminates and therefore does not need to be scheduled as a handler, it does not need any priority or release parameters.

```
public class Basic extends Mission
{
  protected Basic(PriorityParameters priority,
                  AperiodicParameters ap,
                  Scheduler scheduler,
                  MemoryArea memoryArea)
  {
    ... // initialization
  }

  public static void main (String[] args) {
    new Basic(null, null, new CyclicScheduler(),
              new LTMemory(10*1024));
  }
}
```

The interesting part is the initialization of the periodic handlers (the ellipsis above):

```
RelativeTime C = new RelativeTime(4,0);
RelativeTime D = new RelativeTime(20,0);
RelativeTime T = new RelativeTime(20,0);
PeriodicParameters pp = new PeriodicParameters(C,D,T);
addToMission(new Periodic(null,pp, getScheduler(),
                              new LTMemory(1024)));
addToMission(new Periodic(null,pp, getScheduler(),
                              new LTMemory(1024)));
add(); // mission to its scheduler
```

The priority parameters are not needed for a cyclic scheduler and are therefore left as null; but it sets up the release-parameters for the periodic handlers. The scheduler is the cyclic scheduler, and both periodic handlers get a private Linear Time memory. The concrete numbers in the parameters are arbitrary.

Next, we modify the example to include a termination event that can be triggered by a periodic handler. Since the mission is an aperiodic event handler, we use a `PriorityScheduler`. Furthermore a static `event` is used to signal a termination request from the application handlers. This event is handled by the `handleEvent` of the mission. Note that the mission as a handler is included in its handler set.

```
public class Extended extends Mission
{
  static AperiodicEvent event;

  protected Extended(PriorityParameters priority,
                     AperiodicParameters ap,
                     Scheduler scheduler,
                     MemoryArea memoryArea)
  { // set up periodic or other aperiodic handlers
    // with prioritites etc.
    ...
    event = new AperiodicEvent(this);
    add(); // start mission
  }

  public void handleEvent() {
    remove(); // from scheduler
    ...       // clean up etc.
  }

  public static void main (String[] args) {
    new Extended(new PriorityParameters(10),
                 null,new PriorityScheduler(),
                 new LTMemory(10*1024));
  }
}
```

With missions, it is possible to build sequential and concurrent missions, assuming that the selected scheduler is able to handle it. An outline of a sequential composition of three sub-missions is:

```
public class ThreeSequentialMissions extends Mission
{
  private Mission[] mission;
  private int active = 0;

  static AperiodicEvent event;

  public ThreeSequentialMissions(...) {
    mission = new Mission[3];
    // set up the three missions
    mission[0] = new Mission(...);
    // add handlers for mission 0
    // including the mission termination
    ...
    mission[1] = new Mission();
    ...
    mission[2] = new Mission();
    ...

    // start the first mission
    mission[active].add();

    event = new AperiodicEvent(this);
  }

  public void handleEvent() {
     mission[active].remove();
     active = (active + 1) % mission.length;
     mission[active].add();
  }
  ...
}
```

The outer mission removes and terminates the inner missions one at a time and starts the next one. Note that the handlers are initialized once. When the next mission is started, any shared objects will have the state in which they were left by the previous mission. If a reinitialization has to take place, the local missions must define a termination handler that takes care of reinitialization.

Analogously, missions may include sub-missions to be executed concurrently. That is, if the scheduler can accept it.

2.3 Memory

When the runtime system starts an application like `Basic`, an object of this class and objects for its arguments are created in what is usually called the heap (but here without GC). The lifetime of those objects is the lifetime of the application.

Since `Basic` extends `Mission` which is an aperiodic event handler this has as consequences:

- Immortal memory in the sense of RTSJ, which contains objects that has to live for the duration of the mission, is not needed, because the scoped memory belonging to the mission, here `Basic`, takes over this role.
- Just one kind of scoped memory is necessary, because there is no semantic difference between mission memory and private memory for a handler.
- Scoped memories are private for their handlers. In particular, a mission's memory contains objects shared by the set of event handlers of the mission.

This simplifies the memory hierarchy. Yet, in order to be compliant with RTSJ we have retained the following classes in a hierarchy.

The abstract class

```
public abstract class MemoryArea
{
  // dummy implementations,
  // exceptions not considered
  protected MemoryArea(long size) {}

  public void enter(Runnable logic) {}

  public static MemoryArea getMemoryArea(Object object)
  { return null; }

  public Object newArray(Class type, int number)
  { return null; }

  public Object newInstance(Class type)
  { return null; }

  public Object newInstance(Constructor constructor,
      Object[] args)
  { return null; }
}
```

the intermediate abstraction

```
public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size) {
    super(size);
  }
}
```

and the concrete

```
public class LTMemory extends ScopedMemory
{
  public LTMemory (long size) {
    super(size);
  }
}
```

Concrete implementations of `MemoryArea` are shown in Section 4.

2.4 Schedulers

A scheduler must support the mission. Therefore the constructor to a `Mission` has a parameter of type `Scheduler`, see Section 2.2.

The concrete schedulers specialize an abstract profile class:

```
public abstract class Scheduler
{
  private static Scheduler sc;

  protected abstract boolean add(Mission mission);
  protected abstract boolean remove(Mission mission);

  public static Scheduler getDefaultScheduler() {
      return sc;
  }
  protected static void
    setDefaultScheduler(Scheduler scheduler) {
      sc = scheduler;
    }
}
```

We have removed the feasibility checking of RTSJ, because it aims at systems where handlers are added and removed dynamically. Please note that even with nested missions, the overall structure is known at compile time.

New, is the concept of adding and removing missions as a whole. For a simple scheduler that accepts a single mission at a time, the most interesting semantics is termination. We would suggest a mode change semantics, where termination takes place only at points where no handlers are released [31]; but other mode change semantics are certainly possible.

An open point is whether missions are structured counterparts of a RTSJ `ProcessingGroup`. This requires further investigation [36].

Two concrete schedulers are shown in Section 4: a cyclic executive scheduler for a mission with solely periodic event handlers and a fixed-priority preemptive scheduler for missions with both periodic and aperiodic event handlers. These are the schedulers needed to run the examples of Section 2.2

2.5 Synchronization

Objects, shared between handlers, are placed in the mission's memory. In a simple profile, mutual exclusion locking is done by synchronization of the methods of the shared object. Synchronization at block level is not considered because of its added complexity, see the discussion in [10].

Priority inversion can be avoided, either by priority inheritance or priority ceiling emulation. The open source implementation in Section 4 uses priority inheritance because priority ceiling emulation is not implemented in Xenomai.

2.6 IO

Interfacing to devices is an important aspect of real-time applications, and we suggest a solution with Device Objects and Interrupt Handlers; this is discussed in detail in [30, 24].

3. BUILDING THE HIERARCHY

The Predictable Java profile we propose organizes the concepts discussed above in a hierarchy which is a generalization of RTSJ. Here it is useful to recall what inheritance may be used for. Inheritance is a way of forming new classes, allowing subclasses to inherit commonly used state and behaviour from a superclass. This can be interpreted in different ways. Budd [9] lists seven forms of inheritance, of which the most interesting for our purpose are:

Inheritance for specialization: Here a new class is a specialized class of the parent class. The new class satisfies the specification of the parent class. Thus, the new class is a subtype of the parent class. This corresponds to a refinement in a formal semantics setting or a conservative extension in logics.

Inheritance for limitation: (often called implementation inheritance), here the new class redefines the behaviour of the parent class. The new class does not satisfy the specification of the parent class. This means that in the subclass only some of the behaviour from the parent class is allowed in the subclass. This can be done

in different ways: a) overriding the unwanted methods in the subclass and let them throw an `IllegalMethod` exception; b) using Java annotations. This does not fit well with formal semantics or logics.

It is evident that since a predictable Java profile is smaller than RTSJ, it cannot be derived by specialization, and for semantic reasons, we are not happy to pursue a limitation approach.

3.1 The ideal profile

The Predictable Java profile is based on RTSJ. It is a restriction of RTSJ in line with Ravenscar Java [25] and Safety Critical Java [20], but those two profiles are both defined from RTSJ by "inheritance for limitation" which results in much clutter.

To get a clean and compact PJ profile, "inheritance for specialization" was used. By this, RTSJ was regarded as a specialization of PJ. This means that the behaviour of a PJ-class is exactly those methods from the corresponding RTSJ-class that are necessary, and no more. Thus instead of using

```
RTSJ - class
{all the methods in RTSJ-class}
  |
  + -- PJ - class
       {with limitation of methods
        inherited from RTSJ, which
        are not part of PJ-class}
```

we have used:

```
PJ - class
{exactly those methods from RTSJ-
 class necessary to define PJ-class}
  |
  +--RTSJ - class
     {all the methods in RTSJ-class}
```

The resulting classes are shown in Figure 1. A few PJ specific classes are introduced: the `ManagedEventHandler` hierarchy, including the `Mission` subclass, and two subclasses to `AsyncEvent`.

For the `Scheduler` we have been forced to start afresh, because the current RTSJ does not recognize missions. We could have removed the mission specific methods and seen them as subclasses of a very abstract scheduler to be fully consistent with our goal. The `Scheduler` in the PJ would then be a specialization with further concrete specializations `CyclicScheduler` and `PriorityScheduler`. They might even be placed outside the profile.

3.2 RTSJ compliance: delegation

The ideal PJ profile described above requires some workarounds. Because RTSJ already exists, it cannot be defined as subclasses to PJ, but an application that uses the PJ package has to compile and, given that semantics is preserved, run under RTSJ with an adapter layer that essentially disallows some RTSJ methods and gives default values to some parameters. Hereby, the RTSJ compliance requirement is satisfied. An implementation of a concrete adapter layer is shown in the next sections.

4. IMPLEMENTATIONS

We present two open source implementations of the profile: one native implementation and one RTSJ compliant implementation. Source code for both implementations is available, as explained in Appendix A.

4.1 Native implementation

In this part we outline how the profile is implemented on an ARM controller using a modified JamVM on top of Xenomai and Linux. JamVM is both extremely small (∼200K) and optimized, and conforms to the JVM specification version 2 [23]. Xenomai is a real-time extension to the Linux operating system. The Native Xenomai API has different services for real-time tasks and task scheduling, synchronization support including mutexes etc. [39].

4.1.1 Schedulers and handlers

In a basic implementation with a `CyclicScheduler`, we have only one periodic Xenomai `RT_TASK` that implements the well-known cyclic executive model [5]. When the single mission that is allowed for a cyclic scheduler is given the list of handlers, it creates a cyclic executive table. In this table the logic for the periodic handlers are set up, in line with the model, and the period of the periodic Xenomai task is calculated as the greatest common divisor of the periods of the periodic handlers (minor cycle).

Furthermore, the `CyclicScheduler` maintains a list of pointers to the private memories of handlers.

When a `PriorityScheduler` is used, for instance when both periodic and aperiodic event handlers are in play, each handler is bound to a Xenomai RT task, has a memory area allocated by the operating system, and a list of locks belonging to synchronized objects in the memory area. This information is gathered in a vector for the set of handlers in the mission, where each element is defined by:

```
typedef  struct handler_info {
  ..
  MEM_AREA *memArea;  // private mem
  RT_MUTEX *rt_locks; // Xenomai mutex
  RT_TASK   task;     // Xenomai task
  ..
} HANDLER_INFO;
```

A periodic handler uses Xenomai TASK services, for instance `rt_task_set_periodic` and `rt_task_wait_period` . An example is periodic invocation of a handler that is implemented as:

```
for (;;) {
  JNI-callback-to-PeriodicEventHandler-run
  rt_task_wait_period(NULL);
}
```

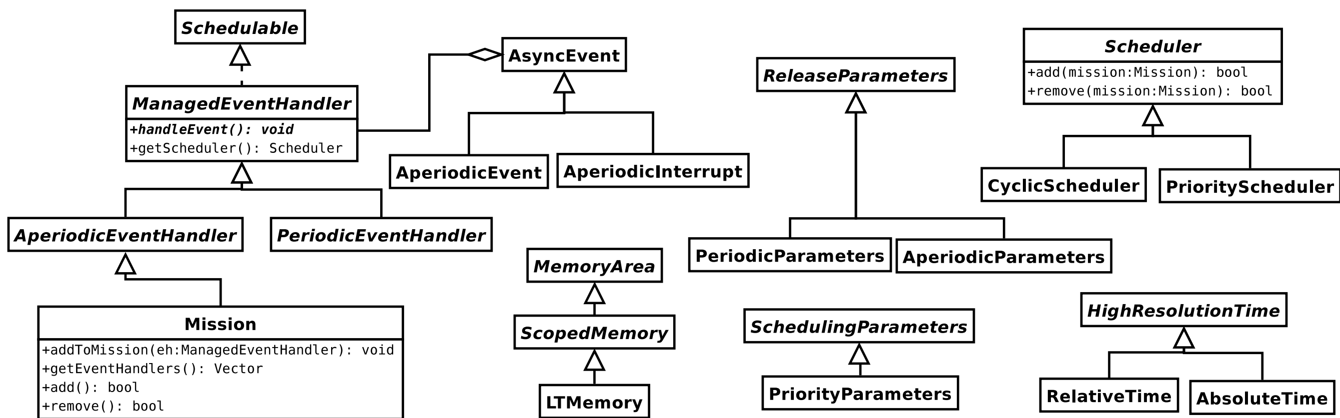

Figure 1: PJ class diagram

Similarly, Xenomai `EVENT` and `INTERRUPT` services are used to implement events or hardware generated interrupts.

The implementation does not support nested missions or, at the moment, mission termination.

4.1.2 Memory management

The garbage collector in JamVM is switched off so that the heap is used as immortal memory.

For the implementation of a `MemoryArea`, the standard `C malloc` is used to allocate a `MEM_AREA`, which is a new struct we have defined in JamVM. It is similar to the existing heap memory structure in JamVM.

Thus, the essential part of the Java implementation of memory area becomes:

```
public abstract class MemoryArea
{
  long memSize;
  int memID; // a reference to the MEM_AREA

  public void enter (Runnable logic) {
    Native.enterNativeMemArea (memID);
    logic.run();
    Native.leaveNativeMemArea (memID);
  }
  ...
}
```

When new objects are created by `logic.run`, they are placed in the memory area belonging to the handler. This memory area is reset by `Native.leaveNativeMemArea`.

4.1.3 Synchronization

Shared objects are represented by classes with synchronized methods. Synchronized blocks are not allowed which means that the Java bytecodes `monitorenter` and `monitorexit` which implement a Java synchronized block [26], are not rewritten in JamVM.

In the virtual machine synchronized methods are marked with the `ACC_SYNCHRONIZED` flag. This means that in JamVM we only need to replace the local objectLock/objectUnlock functions with two new functions called `rt_object_lock` and `rt_object_unlock`.

Those two new functions uses Xenomais `MUTEX` services with methods like `rt_mutex_aquire` and `rt_mutex_release` on elements in the list `rt_locks`.

Xenomai's `MUTEX` services enforce priority inheritance. Priority ceiling emulation is not enforced yet. It is a weakness with Xenomai.

4.2 The RTSJ adapter layer

This section describes the RTSJ implementation of the profile on top of the `timesys 1.0.2` RTSJ implementation [34]. The technique used is encapsulation by delegating responsibility to appropriate RTSJ classes.

4.2.1 Schedulers and handlers

The cyclic scheduler schedules a single mission, containing periodic event handlers only, thus the `add` method expects a `Mission` object satisfying this constraint. From this mission, periodic handlers are retrieved and a cyclic executive table is built: an array where each entry contains a list of handlers for the given minor cycle. Execution is performed by a single periodic thread, with a period of *minor cycle*, implemented using the `NoHeapRealtimeThread` from RTSJ. Each period the `handleEvent` methods are executed on behalf of each handler in the current minor cycle, in the context of the private memory associated with each handler. The periodic thread will loop through the executive table forever. Thus the `remove` method has no effect and returns `false`.

The priority scheduler is implemented using the RTSJ priority scheduler. For each event handler an appropriate schedulable entity is created upon instantiation of the handler. It has a simple logic responsible for executing the event handler logic in the context of its private memory. For periodic handlers this is done using a periodic thread, `NoHeapRealtimeThread`, and for aperiodic handlers, the RTSJ `AsyncEventHandler` is used. When a mission is added to the priority scheduler, using the `add` method, priorities of the RTSJ schedulables are set using deadline monotonic priority assignment, and all periodic threads are started.

Periodic event handlers internally contains a simple implementation of an RTSJ `NoHeapRealtimeThread` as illustrated in the `PeriodicEventHandler` constructor:

```
protected PeriodicEventHandler(...) {
  super(priority, pp, scheduler, memoryArea);
  ...
  final PeriodicEventHandler pevh = this;
  rtsj_thread = new NoHeapRealtimeThread(
    null, rtsj_releaseParameters, null,
    ImmortalMemory.instance(), null,
    new Runnable()
      {
        PeriodicEventHandler _pevh = pevh;
        public void run() {
          for(;;){
            _pevh.memoryArea.enter(_pevh);
            NoHeapRealtimeThread.waitForNextPeriod();
          }
        }
      } // logic
  );
}
```

The `run` method of the `PeriodicEventHandler` class invokes the logic of the event handler, i.e. the `handleEvent` method. The `rtsj_releaseParameters`, supplied to the thread constructor contains the RTSJ equivalent of the PJ periodic parameters. The `NoHeapRealtimeThread` is executed in the context of immortal memory since it requires no additional memory during execution.

Similarly, the aperiodic event handler is implemented using the RTSJ `AsyncEventHandler`:

```
protected AperiodicEventHandler(...) {
  super(priority, ap, scheduler, memoryArea);
  ...
  final AperiodicEventHandler aevh = this;
  rtsj_asyncEventHandler = new AsyncEventHandler(
      null, null, null,
      ImmortalMemory.instance(), null,
      true, // nonheap
      new Runnable()
      {
        AperiodicEventHandler _aevh = aevh;
        public void run() {
          _aevh.memoryArea.enter(_aevh);
        }
      } // logic
    );
}
```

Here, the `rtsj_asyncEventHandler` is a package visible field of `AperiodicEventHandler`, and hence is accessible to the class `AperiodicEvent`. The `fire` method of `AperiodicEvent` then delegates the responsibility of fire to an instance of the RTSJ `AsyncEvent` class associated with `rtsj_asyncEventHandler`.

The `remove` method has not yet been implemented. We consider using the mode change semantics of [31] and implementations as suggested in [28].

4.2.2 Memory management

The memory area classes are implemented by delegation using their corresponding RTSJ classes. Event handlers has their own scoped memories in which their logic can create temporary objects. Since all handlers must be referenceable by the RTSJ scheduler, the handler objects (Mission, PeriodicEventHandler, and AperiodicEventHandler) are allocated in immortal memory. This violates the hierarchical memory structure, where the handlers of a mission shares the mission memory. Instead immortal memory is used, a temporary solution which in the future could be reworked using techniques inspired by those presented in [2].

5. PROFILE COMPLIANCE

The most important tasks in checking that a program is in compliance with the profile as described above are: ensuring schedulability, verifying that (temporary) memory consumption is within bounds, and checking that no non-temporary objects are allocated. For these purposes Worst Case Execution Time (WCET) and Worst Case Memory Consumption (WCMC) analyses are needed and may be combined with model checkers, such as UPPAAL [35], to perform a full schedulability analysis.

Additionally, a compliance check may also need to enforce syntactic and structural requirements and constraints demanded by the profile, e.g., that all loops are explicitly bounded and that the program is not recursive. Standard analyses, such as control flow, data flow, and information flow analyses combined with simple syntactic checks are sufficient for this.

The design of our Java profile is intended to facilitate comprehensive tool support for most or all aspects of the profile. In particular, we believe that abstract interpretation, and similar static analysis techniques, combined with model checking can be leveraged to provide automated analysis and verification of important properties such as resource usage and profile compliance. Going beyond resource and compliance checking, static analysis techniques have also been applied with great success in the area of compile time verification of safety and security properties, e.g., prevention of race conditions and deadlocks, guaranteed secure information flow, and bug hunting.

Below we discuss in more detail the relevant analyses and how they may be applied here.

5.1 Resource usage

A prime example of an analysis in this category is the WCET analysis [38, 12, 33, 37] mentioned above. A WCET analysis can statically compute a sound over-approximation of the worst possible execution time behaviour of a program. This is needed, among other things, to determine if a given program can safely be scheduled. Dually, a *best case execution time* [38] computes a conservative under-approximation of the execution time of a given program. This may, in certain cases, be used to give a (conservative) lower bound on the execution time for aperiodic event generators and thus on the minimal interarrival time between aperiodic events.

In [7] a more direct approach to schedulability analysis is taken: here it is shown how to automatically extract a timed-automata based model of a Java bytecode program. The extracted model is then model-checked, using UPPAAL, to determine directly if it is schedulable.

Other analyses in this category include analyses to determine worst case memory (WCMC) usage [16, 11] and maximum stack depth, both of which are instrumental for ensuring that a given program can be executed within the bounds set by the platform.

5.2 Safety and security

By computing standard control flow, data flow, and information flow analyses a number of safety and security properties can be guaranteed at compile time. Including the absence of certain bugs, e.g., null pointer exceptions [21], initialisation failures [22], secure information flow [19, 18, 14, 6], and avoiding race conditions [1, 13].

Combining a best/worst case execution time analysis with analyses that extract abstract timing models from a program, e.g., in the form of timed automata [7], it may be possible to give a compile time proof that a given program cannot possibly end in a deadlock state nor a livelock situation.

6. CONCLUSION

We have presented Predictable Java (PJ), a Java profile suitable for development of high integrity real-time systems. PJ shows that it is beneficial to define a specialized profile as a generalization of RTSJ. This is in contrast to other proposals, such as Ravenscar Java [25] and Safety-Critical Java [17, 4] which are (extended) subsets defined as limitations of RTSJ using subclassing. PJ uses only the handler paradigm, having periodic and aperiodic handlers. Those handlers are grouped in missions which are first-class objects as the `Mission` class is a subclass of the `AperiodicEventHandler` class.

Each handler has a private memory for allocation of local objects during execution of its `handleEvent` method. This private memory is reset every time this method completes. Because a mission is an event handler there is no semantic difference between mission memory and handler memory. This also means that the memory belonging to a mission is reset when a mission comes to end.

In order to ensure that it is a valid profile, it has two prototype implementations on different platforms; An RTSJ platform, showing that PJ is compliant with RTSJ, and a "native" platform.

7. REFERENCES

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] M. Alrahmawy and A. Wellings. Design patterns for supporting rtsj component models. In *Proc. of the 7th international workshop on Java technologies for real-time and embedded systems (JTRES'09)*. ACM Press, 2009.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In K. G. Larsen and P. Niebert, editors, *Proceedings of FORMATS 2003*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer-Verlag, 2004.

[4] Aonix. Aonix research and development - safety critical java specification initiative. `http://research.aonix.com/jsc/index.html`, 6 2009.

[5] T. P. Baker and A. Shaw. The cyclic executive model and ada. *The Journal of Real-Time Systems*, 1:7–25, 1989.

[6] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2007.

[7] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proc. of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES'08)*, pages 106–114. ACM Press, 2008.

[8] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, 2000.

[9] T. Budd. *Understanding Object-Oriented Programming with Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[10] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 4 edition, 2009.

[11] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *Procóf 13th International Symposium on Formal Methods (FM'05)*, number 3582 in Lecture Notes in Computer Science, pages 91–106. Springer-Verlag, 2005.

[12] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in WCET analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation. Theory and Practice. Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[13] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM, 2000.

[14] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, Paris, France, Jan. 2005. Springer Verlag.

[15] P. Gerum. The xenomai project. implementing a rtos emulation framework on gnu/linux. In *Third Real-Time Linux Workshop*, 2001.

[16] P. Giambiagi and G. Schneider. Memory consumption analysis of Java smart card. In *In Proc. of Conferencia Latinoamericana de Informatica (Latin American Computing Conference), CLEI'05*, Santiago de Cali, Columbia, Oct. 2005.

[17] T. O. Group. Jsr 302: Safety critical java technology.

http://jcp.org/en/jsr/detail?id=302, 2006.

[18] R. R. Hansen. A Hardest Attacker for Leaking References. In D. Schmidt, editor, *Proc. of European Symposium on Programming, ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 310–324, Barcelona, Spain, Mar./Apr. 2004. Springer Verlag.

[19] R. R. Hansen and C. W. Probst. Non-Interference and Erasure Policies for JavaCard Bytecode. In *Workshop on Issues in the Theory of Security, WITS'06*, pages 174–189, Vienna, Austria, Mar. 2006.

[20] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[21] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of the 10th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer-Verlag, 2008.

[22] L. Hubert and D. Pichardie. Soundly handling static fields: Issues, semantics and analysis. *Proc. of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09), ENTCS*, 2009. To appear.

[23] JamVM. JamVM 1.5.3. http://jamvm.sourceforge.net/, 4 2009.

[24] S. Korsholm, M. Schoeberl, and A. P. Ravn. Interrupt handlers in Java. In *11th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 453–457. IEEE Computer Sciety Press, 2008.

[25] J. Kwon, A. Wellings, and S. King. Ravenscar-java: a high integrity profile for real-time java. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140, New York, NY, USA, 2002. ACM.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2 edition, 1999.

[27] P. Puschner and A. Wellings. A profile for high-integrity real-time java programs. pages 15–22. IEEE Computer Society, 2001.

[28] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[29] M. Schoeberl, H. Søndergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In *11th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 94–101. IEEE Computer Sciety Press, 2007.

[30] M. Schoeberl, C. Thalinger, S. Korsholm, and A. P. Ravn. Hardware objects for Java. In *11th IEEE International Symposium on Oriented Real-Time Distributed Computing (ISORC)*, pages 445–452. IEEE Computer Sciety Press, 2008.

[31] H. Søndergaard, A. P. Ravn, B. Thomsen, and M. Scoeberl. A practical approach to mode change in real-time systems. Technical Report 08-001, Department of Computer Science, Aalborg University, 2008.

[32] H. Søndergaard, B. Thomsen, and A. P. Ravn. A ravenscar-java profile implementation. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 38–47, New York, NY, USA, 2006. ACM.

[33] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, pages 625–632. IEEE Computer Society Press, June 2003.

[34] Timesys. Rtsj reference implementation (ri) and technology compatibility kit (tck). http://www.timesys.com/java/, 2009.

[35] UPPAAL. Uppaal. http://www.uppaal.com/, 2009.

[36] A. J. Wellings and M. S. Kim. Processing group parameters in the real-time specification for java. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 3–9, New York, NY, USA, 2008. ACM.

[37] R. Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[38] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrøm. The determination of worst-case execution times—overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

[39] Xenomai. Xenomai native skin API reference manual, 2006.

APPENDIX

A. SOURCE CODE

The implementations mentioned in this paper are open source. The source code for both implementations, and further instructions, is available at `http://pj.boegholm.dk`. Please refer to the individual `README` files for further instructions.

The RTSJ layer implementation includes two examples, `CyclicExample` and `PriorityExample`. These examples are using the two scheduling mechanisms together with a few handlers. The reference implementation of RTSJ used is the timesys 1.0.2, available at: `http://timesys.com/java`. A `Makefile` exists for running and compiling the examples, the latter using an `ant` build configuration. Instructions are found in the `README` file.

The implementation using `JamVM`, `Xenomai`, and `Linux`, contains a modified version of the JamVM virtual machine, JNI functions, and the PJ implementation, with examples. Instructions on how to compile and run the examples are found in the `README` file.