Ravenscar-Java: A High Integrity Profile for Real-Time Java

Jagun Kwon, Andy Wellings, and Steve King

Department of Computer Science
University of York, UK
{jagun, andy, king}@cs.york.ac.uk

ABSTRACT

For many, Java is the antithesis of a high integrity programming language. Its combination of object-oriented programming features, its automatic garbage collection, and its poor support for real-time multi-threading are all seen as particular impediments. The Real-Time Specification for Java has introduced many new features that help in the real-time domain. However, the expressive power of these features means that very complex programming models can be created, necessitating complexity in the supporting real-time virtual machine. Consequently, Java, with the real-time extensions as they stand, seems too complex for confident use in high integrity systems. This paper presents a Java profile for the development of software-intensive high integrity real-time systems. This restricted programming model removes language features with high overheads and complex semantics, on which it is hard to perform timing and functional analyses. The profile fits within the J2ME framework and is consistent with well-known guidelines for high integrity software development, such as those defined by the U.S. Nuclear Regulatory Commission.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors – *Memory management (garbage collection), Run-time environments*; D.2.4 [**Software Engineering**]: Software/Program Verification – *Reliability*; J.7 [**Computer Applications**]: Computers in Other Systems – *Real time*

General Terms

Reliability, Languages, Performance

Keywords

High integrity systems, Real-time Java, Profile

1. INTRODUCTION

Increasingly computers are being used in high integrity real-time systems; that is, systems where failure can cause loss of life, environmental harm, or significant financial penalties. Examples include space shuttles, nuclear power plants, automatic fund transfers and medical instruments. They typically have high development and maintenance costs due to the customised nature

of their components. Within such systems, there has been a growing trend to use software, because it provides [25, 26, 30, 8]:

• improved functionality
• increased flexibility in design and implementation
• reduced production cost
• enhanced management of complexity in application areas.

Java [16] has proved to be an appropriate vehicle for a diverse range of applications including web-based intranets and embedded systems. Its relatively simple linguistic semantics, the adoption of well-understood approaches to managing software complexity, and support for concurrency seem to have contributed towards its popularity. Initially designed with embedded systems in mind, Java's main goal was to provide engineers with a reliable and cost-effective platform-independent environment.

However, despite all these valuable features, Java has been criticised for its unpredictable performance as well as some security concerns [1, 3]. The automatic garbage collection and dynamic class loading mechanisms are often considered problematic, especially under time or performance-critical situations. Moreover, a number of security bugs in the Java virtual machine have been discovered since its first appearance, especially in the bytecode verifiers and Just-in-Time (JIT) compilers [15, 1]. These fears make Java and its associated technology simply unsuitable for high integrity systems [22].

In recent years, there have been two main activities, initiated by Sun, to address the limitations of Java for real-time and embedded systems. The first is, *the Real-Time Specification for Java* (RTSJ) [7] which attempts to minimise any modification to the original Java semantics and yet to define many additional classes that must be implemented in a supporting virtual machine. The goal is to provide a predictable and expressive real-time environment. This, however, ironically leads to a language and run-time system that are complex to implement and have high overheads at run-time. Software produced in this framework is also difficult to analyse with all the sophisticated features, such as the asynchronous transfer of control (ATC) and dynamic class loading.

The second relevant activity is the *Java 2 Platform Micro Edition* (J2ME) [33]. This essentially defines a three layer architecture:

• a virtual machine layer (usually implemented on top of a host operating system),
• a configuration layer which defines the set of Java language features, a minimum set of virtual machine features and the available class libraries that can be supported by a particular implementation platform (for example, a mobile phone),
• a profile layer which defines a minimum set of Application Programmers Interfaces (APIs) targeted at a particular application domain.

The same configuration layer may support more than one profile, and different configuration layers may support the same profile. A configuration layer, called Connected, Limited Device Configuration (CLDC) has been defined for small, resource-constrained mobile devices (mobile phones, pagers, personal organizers etc.) typically with a memory capacity of up to 512 KB. The K (kilo bytes) virtual machine (KVM) is a virtual machine specifically designed to support the CLDC. The restrictions imposed on the Java language and the virtual machine include [33]: no support for floating point operations, no native interfaces, no user-defined class loaders, no thread groups or daemon threads, no object finalization, etc. The main motivation for these restrictions is to reduce the size of memory required to implement the virtual machine.

It is clear that the overheads of implementing the RTSJ makes it unsuitable for the CLDC configuration and consequently RTSJ, as it stands, is probably targeted at Java 2 Standard Edition (J2SE) or ideally another configuration (for example, the CDC – Connected Device Configuration) within the J2ME framework. However, a high integrity subset of the RTSJ model would be appropriate for J2ME and it is possible to imagine a high integrity KVM and CLDC along with one or more profiles.

Unfortunately, many language subsets for high integrity systems discourage the use of concurrent activities. For example, Ada is one of the most important programming languages for the high integrity systems application domain. The SPARK subset of Ada [4] (which removes many of the language's complicated or advanced features such as tasking, exceptions, overloading etc.) allows programs to be mathematically proven correct. In recent years, advances in real-time systems research and, in particular, in the area of schedulability analysis, have meant that it is now possible to show mathematically that a concurrent program will meet its deadlines. Of course, constraints must be placed on the particular concurrency mechanisms used to ensure predictability. However, it is no longer axiomatic that concurrency should be forbidden or even discouraged.

To encourage the use of concurrency in high integrity real-time systems, the Ada community has developed a subset of the Ada tasking model (including the Real-Time Annex) called the Ravenscar Profile [11]. The main aims of the subset are to support a predictable computational model and to enable a small efficient and predictable run-time support system to be produced. The Ravenscar Profile has attracted support from users and compiler (and run-time) vendors, and has become a *de facto* standard in the high integrity system domain. It will soon be incorporated into the Ada language standard.

Following the philosophy of the Ravenscar profile, we propose a high integrity profile for real-time Java (called Ravenscar-Java) that offers a more reliable and predictable programming environment. In other words, our profile eliminates features with high overheads and complex semantics, so that programs become more analysable and ultimately, more dependable.

This paper is structured as follows: the next section sets the scope and describes the organisation of the profile. In section 3 we show the computational model, before the actual profile is illustrated in detail. Section 5 briefly looks at implementation issues, followed by an example Ravenscar-Java program in Section 6. Related work is considered in Section 7 and the paper concludes with Section 8. The full description of the rules and guidelines of the profile can be found in [23].

2. SCOPE AND ORGANISATION OF THE PROFILE

There are many general and sector-specific standards that assist in the construction of high integrity systems (e.g. U.S. DO178B, U.K. DS 00-55, MISRA guidelines, IEC61508). Of particular interest here is the set of software guidelines produced by the U.S. Nuclear Regulatory Commission (NRC) [29] because it is specific to high integrity systems and because it has set up a systematic framework of guidelines by deriving many important attributes from existing standards. There are four top-level attributes:

- Reliability — defined as the "predictable and consistent performance of the software under conditions specified in its design." A key factor in obtaining reliability is to have predictability of the program's execution; in particular: predictability of control and data flow, predictability of memory utilization and predictability of response times.
- Robustness — defined as "the capability of the safety system software to operate in an acceptable manner under abnormal conditions or events." Often called fault tolerance or survivability, this attribute requires the system to cope with both anticipated and unanticipated faults. Techniques such as using replication, diversity and exception handling are commonly used [12].
- Traceability — relates to "the feasibility of reviewing and identifying the source code and library component origin and development processes" thus facilitating verification and validation techniques, which are essential aids to ensuring program correctness.
- Maintainability — relates to "the means by which the source code reduces the likelihood that faults will be introduced during changes made after delivery." All the standard software engineering issues apply here such as good readability, use of appropriate abstraction techniques, strong cohesion and loose coupling of components, and portability of software components between compilers and platforms [32].

The report also provides guidelines based on the framework for nine programming languages (including Ada95 and C/C++). Unfortunately, the guidelines do not consider Java.

The goal of this paper is to apply the NRC's framework to Java augmented by the RTSJ. The paper focuses on the *reliability* attribute as the rest of the attributes are concerned with general design decisions that are covered in the software engineering literature. However, we still give several Java specific guidelines in those areas where they have impacts on the RTSJ.

3. COMPUTATIONAL MODEL

The key aim of the Ravenscar-Java profile is to develop a concurrent Java programming model that supports predictable and reliable execution of application programs, thus benefiting the construction of modern high integrity software. Particularly, we follow the philosophy of the Ravenscar profile [11] and emphasise the *reliability* attribute of the NRC guidelines. This means that some language features with high overheads and complex semantics are removed for the sake of reliability, and programs are statically analysable in terms of functionality and timeliness before execution. Similarly, the Java virtual machine is also restricted to ensure predictability and efficiency. For example, a Ravenscar-Java VM (RVM) does not support garbage collection.

As in the RTSJ, the Ravenscar-Java profile allows concurrent execution of schedulable objects (threads and event handlers) based on pre-emptive priority-based scheduling. Schedulable objects have to be either periodic or sporadic with minimum inter-arrival times, and the priority ceiling protocol is required to be implemented in the runtime system. This profile facilitates the use of off-line schedulability analysis, which is associated with fixed priority scheduling (e.g. deadline monotonic or rate monotic analysis [2, 27]).

We assume two execution phases as suggested in [31], i.e. *initialisation* and *mission* phase, as shown below in Figure 1. In the initialisation phase of an application (i.e. the *main*() method and one *RealtimeThread*), all non-time-critical activities and initialisations that are required before the mission phase are carried out. This includes initialisation of all real-time threads, memory objects, event handlers, events, and scheduling parameters[1]. In the mission phase, the application is executed and multithreading is allowed based on the imposed scheduling policy.
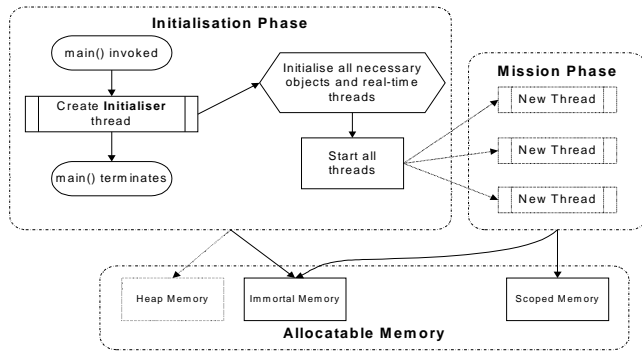

Figure 1. Two execution phases

4. THE PROFILE

The proposed profile can be classified into the following headings, and each of them is expanded below:

- Predictability of memory utilisation
- Predictability of timing
- Predictability of control and data flow.

We separate the rules and guidelines of the profile into the following categories (for a full list of rules and guidelines, see [23]), and the NRC framework is applied to each of them:

1. Programming in the Large
2. Concurrent Real-Time Programming
3. Programming in the Small.

In the first category, we give guidelines on the use of language features that support high-level decomposition and minimise the complexity of software, which involve object-orientation, and abstract data types. In the second, guidelines on the use of features provided by the RTSJ are presented, whereas in the third we discuss programming issues related to the production of small software components, such as control structures, methods, and expressions.

---

[1] This includes loading all the classes needed in the application. In a JIT (Just-In-Time) compilation environment, all loaded classes will be compiled.

The resulting profile is targeted at a RVM; however, programs written according to the profile are valid RTSJ programs, which will execute without change under a RTSJ virtual machine (although perhaps not within their deadlines).

This section summarises the rules and guidelines. A fuller description is given in [23].

4.1. Predictability of memory utilisation
This attribute is concerned with ensuring that the software will not access unintended or disallowed memory locations, and ensuring that the use of memory space will be predictable and bounded.

4.1.1. Initialisation and mission phases
When an application is started, its *main*() method will first be invoked by the RVM and the base heap memory area will be used to allocate any objects within the method (as with standard Java).

The main method first creates a new *NoHeapRealtimeThread* with the highest priority in the system. This is required to ensure a well-ordered and controlled execution of the initialisation and mission phases, as illustrated in Figure 2.

```
import javax.realtime.*;
class Main implements Runnable
{
  public static void main(String [] args)
  {
    NoHeapRealtimeThread initializer = new
      NoHeapRealtimeThread(new PriorityParameters(
        PriorityScheduler.MAX_PRIORITY), null, null,
      ImmortalMemory.instance(), null, new Main());
    initializer.start();
  }

  public void run()
  {  // initialization phase of the program
  }
}
```

Figure 2. An illustration of the initialisation phase

The new thread must take a reference to the *immortal* memory area, so that all objects and references to other threads and memory objects defined in the *initializer* thread will be safely created and maintained throughout the life of the application. Once all initialisation activities are performed, the thread will allow other threads to execute by invoking the *start*() methods, and terminating itself. To encapsulate this initialisation phase, the Ravenscar-Java profile defines an initializer thread class, shown in Figure 3, which directly extends the **RealtimeThread** class.

```
package ravenscar;
import javax.realtime.*;

public class Initializer extends RealtimeThread
{
  public Initializer()
  {
    super( new PriorityParameters(
      PriorityScheduler.MAX_PRIORITY),
      null, null,
      ImmortalMemory.instance(), null, null);
  }
}
```

Figure 3. **Initializer** class of Ravenscar-Java profile

Now, the application can be created by extending the **Initializer** class in the following way.

```
import ravenscar.*;
public class MyApplication extends Initializer
{
  public void run()
  { // logic for initialization
  }
  public static void main (String [] args)
  {
    MyApplication myApp = new MyApplication();
    myApp.start();
  }
}
```

The mission phase begins as soon as the highest priority thread (the **Initializer)** terminates. From this moment, all application threads will be scheduled and despatched according to the imposed scheduling policy. Threads may only utilise immortal and linear-time scoped memory areas in this phase, unless their logics require access to physical or raw memory areas[2].

4.1.2. Memory Management
To facilitate predictable memory utilisation we define several rules in the three aforementioned areas (see [23] for the full list and rationales). The rules place restrictions on, for example, the use of class loaders in the mission phase, on the use of specific memory area objects (and garbage collector), and on recursive method calls. It is also disallowed to create or instantiate any schedulable objects in the mission phase as this will hamper static memory usage analysis.

The heap memory area may or may not exist in a supporting virtual machine. In fact, such memory space can be utilised as part of the whole immortal memory area, since no garbage collection is allowed in the profile.

• Use of immortal memory areas
By definition, objects in an immortal memory area cannot be freed or moved, and all schedulable objects in an application share the same memory area [7]. Hence, in an attempt to prevent memory exhaustion or corruption, objects (including memory area objects) that are needed for the lifetime of the application should be allocated in the area only in the initialisation phase.

• Use of linear time scoped memory areas
All memory area objects must be created during the initialisation phase (thus, in the immortal memory area), and other objects during the mission phase should make use of **LTMemory** areas. The size of all memory objects must be static and not be extended in the course of the program. Any other memory area objects defined in the RTSJ are disallowed, and the following simplified classes remain in the profile.

```
package ravenscar;
public abstract class MemoryArea
{
  protected MemoryArea(long sizeInBytes);
  protected MemoryArea(javax.realtime.SizeEstimator
```

```
                       size);
  public void enter(java.lang.Runnable logic);
        // throws ScopedCycleException
  public void executeInArea(java.lang.Runnable logic)
        throws InaccessibleAreaException;
  public static MemoryArea getMemoryArea(
                       java.lang.Object object);
  public long memoryConsumed();
  public long memoryRemaining();
  public java.lang.Object newArray(java.lang.Class
                              type, int number)
        throws IllegalAccessException,
                        InstantiationException;
        // throws OutOfMemoryError
  public java.lang.Object newInstance(java.lang.Class
                              type)
        throws IllegalAccessException,
                        InstantiationException;
        // throws OutOfMemoryError
  public java.lang.Object newInstance(
        java.lang.reflect.Constructor c,
        java.lang.Object[] args)
        throws IllegalAccessException,
                        InstantiationException;
        // throws OutOfMemoryError;
  public long size();
}
public final class ImmortalMemory extends MemoryArea
{
  public static ImmortalMemory instance();
}

public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size);
  public ScopedMemory(SizeEstimator size);
  public void enter();
  public int getReferenceCount();
}

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

Figure 4. Simplified memory area classes

To aid in the production of an efficient RVM and to simplify timing and memory usage analyses, access to **LTMemory** areas must not be nested and **LTMemory** areas must not be shared between **Schedulable** objects.

4.2. Predictability of timing
This attribute focuses on demonstrating that all schedulable objects meet their timing constraints at runtime. The restrictions enforce the computational model given in Section 3 and, thereby, allow schedulability analysis to be performed.

4.2.1. Scheduling and Threading Model
As suggested in the RTSJ, the minimum required scheduling base is by default a fixed-priority pre-emptive scheduler (represented by the **PriorityScheduler** class) that supports at least 28 unique priority levels. The specification also requires that an implementation makes available at least 10 additional native priorities for regular Java threads. However, the profile does not support regular threads by disallowing the use or overriding of the class **java.lang.Thread** to create threads. Therefore, we do not

assume any additional native priority levels for regular Java threads. As a result, the supported types of schedulable objects in the profile are

• Periodic threads (see **PeriodicThread** class below), and
• Sporadic event handlers (see **SporadicEventHandler** class below).

The **RealtimeThread** and **AsyncEventHandler** classes are not directly available to the applications programmer, as the former may use the heap memory, whereas the latter hinders accurate timing and memory analyses.

Attributes such as scheduling characteristics and memory areas must be statically allocated to schedulable objects in the initialisation phase, and shall not be changed afterwards, in order to facilitate fixed-priority scheduling algorithms and schedulability analysis. For this purpose, all methods whose names begin with '**set**' (for example, **setReleaseParameters**()) and some with '**get**' are excluded. Thus the **Schedulable** interface is defined as an empty interface, as shown below.

```
package ravenscar;
public interface Schedulable extends java.lang.Runnable
{
}
```
Figure 5. Empty **Schedulable** interface

Only fixed priority-based scheduling is supported by the Ravenscar-Java profile. Furthermore, any subclass of the **Scheduler** including the default **PriorityScheduler** class is not allowed to perform any feasibility checks, leading to the classes in Figure 6. The **PriorityParameters** class also does not contain **setPriority**() method, and the **ImportanceParameters** class is not supported.

```
package ravenscar;
public abstract class Scheduler
{
}
public class PriorityScheduler extends Scheduler
{
  public static final int MAX_PRIORITY;
  public static final int MIN_PRIORITY;
}
```
Figure 6. Simplified **Scheduler** classes

Overall, this approach does not necessitate any dynamic feasibility test and admission control by the RVM at runtime. All schedulability analysis is performed before the initialisation phase of the program.

4.2.2. Use of release parameters
In order to support periodic or sporadic behaviours of real-time threads, the following simplified **ReleaseParameters** class and its subclasses are defined.

```
package ravenscar;
public class ReleaseParameters
{
  protected ReleaseParameters();
}

public class PeriodicParameters extends ReleaseParameters
{
  public PeriodicParameters(AbsoluteTime startTime,
```

```
                            RelativeTime period);
  protected AbsoluteTime getStartTime();
  protected RelativeTime getPeriod();
}

public class SporadicParameters extends ReleaseParameters
{
  public SporadicParameters(RelativeTime minInterarrival);
  protected RelativeTime getMinInterarrival();
}
```
Figure 7. **ReleaseParameters** and its subclasses

The **AperiodicParameters** class is undefined, as the profile does not support aperiodic activities.

4.2.3. Use of threads
Most of the methods and fields of the original **java.lang.Thread** class are obsolete in the context of the RTSJ and high integrity real-time applications. So, this class is defined as follows[3].

```
package java.lang;
public class Thread implements Runnable
{
  Thread();
  Thread(String name);
  void start();
}
```
Figure 8. Newly defined **java.lang.Thread** class

Along the same lines, the **RealtimeThread** and **NoHeapRealtimeThread** can be defined as:

```
package ravenscar;
public class RealtimeThread extends java.lang.Thread
              implements Schedulable
{
  RealtimeThread(PriorityParameters pp,
        PeriodicParameters p);
  RealtimeThread(PriorityParameters pp,
        PeriodicParameters p, MemoryArea ma);
  public static RealtimeThread currentRealtimeThread();
  public MemoryArea getCurrentMemoryArea();
  void start();
  static boolean waitForNextPeriod();
}

public class NoHeapRealtimeThread extends RealtimeThread
{
  NoHeapRealtimeThread(PriorityParameters pp,
        MemoryArea ma);
  NoHeapRealtimeThread(PriorityParameters pp,
        PeriodicParameters p, MemoryArea ma);
  void start();
}
```
Figure 9. **RealtimeThread** and **NoHeapRealtimeThread** class

• Periodic Threads
Periodic threads transparently invoke the **waitForNextPeriod** method of the **RealtimeThread** class at the end of their main loops to delay until their next periods. Other mechanisms (e.g. **sleep**() method) are prone to have an inaccurate timing model, thus should not be used.

---

[3] The profile changes some of the access modifiers of the classes, constructors, and methods in order to ensure they cannot be used directly by the programmer. The changes are always more restrictive and hence programs will always execute on non-Ravenscar implementations.

The profile defines an additional class to automate the management of periodic threads, which is shown below.

```
package ravenscar;
public class PeriodicThread extends NoHeapRealtimeThread
{
  public PeriodicThread(PriorityParameters pp,
        PeriodicParameters p, java.lang.Runnable logic);
  public void run();
  public void start();
}
```

Figure 10. **PeriodicThread** class

This class may be utilised as follows. Note that the class assumes the default memory area is the immortal one, and a recovery procedure from a missed deadline can be implemented (if supported by the implementation of the profile).

```
package ravenscar;
public class PeriodicThread extends
                            NoHeapRealtimeThread
{
  public PeriodicThread(PriorityParameters pp,
      PeriodicParameters p, java.lang.Runnable logic)
  {
    super(pp, p, ImmortalMemory.instance());
    applicationLogic = logic;
  }
  private java.lang.Runnable applicationLogic;
  public void run()
  {
    boolean noProblems = true;
    while(noProblems) {
      applicationLogic.run();
      noProblems = waitForNextPeriod();
    }
    // A deadline has been missed,
    // If allowed, a recovery routine would be placed
    // here
  }

  public void start()
  {
    super.start();
  }
}
```

Figure 11. An illustration of the **PeriodicThread** class

• **Sporadic Activities**
Event-triggered activities are supported by means of the **BoundAsyncEventHandler** class. Once an event and its handler are set up, they must remain unchanged permanently. For predictability, it is assumed that each handler is bound to one server thread and each server thread has only one handler bound to it. Again, we define a new class specifically designed for sporadic activities, as shown below. It is based on the **AsyncEventHandler** class hierarchy.

```
package ravenscar;
public class AsyncEventHandler implements Schedulable
{
  AsyncEventHandler(PriorityParameters pp,
        ReleaseParameters p, MemoryArea ma);
  AsyncEventHandler(PriorityParameters pp,
        ReleaseParameters p, MemoryArea ma,
        java.lang.Runnable logic);

  public MemoryArea getCurrentMemoryArea();
```

```
  protected void handleAsyncEvent();
  public final void run();
}

public class BoundAsyncEventHandler
            extends AsyncEventHandler
{
  BoundAsyncEventHandler(PriorityParameters pp,
        MemoryArea ma, ReleaseParameters p);
  BoundAsyncEventHandler(PriorityParameters pp,
        MemoryArea ma, ReleaseParameters p,
        java.lang.Runnable logic);

  protected void handleAsyncEvent();
}

public class SporadicEventHandler
                    extends BoundAsyncEventHandler {
  public SporadicEventHandler(PriorityParameters pri,
                      SporadicParameters spor);
  public SporadicEventHandler(PriorityParameters pri,
                      SporadicParameters spor,
                      java.lang.Runnable);
  public void handleAsyncEvent();
};
```

Figure 12. Event handlers and the **SporadicEventHandler** class

Classes associated with event handlers are shown in Figure 13 below.

```
package ravenscar;
public class AsyncEvent
{
  AsyncEvent();
  void addHandler();
  void fire();
  void bindTo();
}

public class SporadicEvent extends AsyncEvent
{
  public SporadicEvent(SporadicEventHandler handler);
  public void fire();
}

public class SporadicInterrupt extends AsyncEvent
{
  public SporadicInterrupt(SporadicEventHandler handler,
                      java.lang.String happening);
}
```

Figure 13. Associated classes to **SporadicEventHandler** class

Note that all event handlers are bound to their associated event when the event is created.

• **Processing Groups, Overrun and Deadline-miss handlers**
Processing groups (i.e. instances of the **ProcessingGroupParameters** class) are not supported in the profile, as they require runtime support for the scheduler to determine the feasibility of the temporal scope of a processing group (which thus hampers static timing analysis). Overrun and deadline-miss handlers are also not required as schedulability analysis has been performed off-line.

4.2.4. Synchronization
The *synchronized* construct in Java provides mutually exclusive access to shared resources or objects, and programmers are always encouraged to use it to avoid data races. However, excessive use

of this mechanism may result in poor response time, implying that high priority threads may have to wait until lower ones finish their *synchronized* methods or blocks. Therefore, in order to prevent unbounded priority inversions and possible deadlocks, the priority ceiling protocol (**PriorityCeilingEmulation** class) must be implemented and explicitly used for all objects with synchronized blocks or methods. Furthermore, the profile does not support *wait*, *notify* and *notifyAll* methods of the object class. All condition synchronization between real-time threads must be via sporadic event handlers as this ensures that the timing properties of the synchronization are properly addressed.

WaitFreeQueues are not required as they are provided in the RTSJ to enable communication between instances of **NoHeapRealtimeThread** and regular Java threads.

4.2.5. Representation of time
Supported representations of time are

- **HighResolutionTime**,
- **AbsoluteTime**, and
- **RelativeTime**.

These classes allow representation of time with up to nanosecond accuracy and precision [7].

4.2.6. Timer classes
In the presence of the aforementioned classes that offer timely periodic and sporadic behaviours of threads, the **Timer** and its subclasses are not necessary and not available.

4.2.6. Asynchrony
The Asynchronous Transfer of Control (ATC) mechanism is not allowed, as it is one of the most complicated features of the RTSJ and hinders timing and functional analyses [10].

4.3. Predictability of control and data flow
Predictability of control and data flow is required in order that static analysis techniques can be used to aid programming proof techniques and worst-case execution time analysis. Again, all the rules and guidelines are listed in [23], but noteworthy ones in each of the three areas include

• Programming in the large
- All user-defined classes must include constructors that initialise all internal variables and objects.
- Dynamic method binding should be minimised. In particular, method overriding and the use of interfaces should be minimised.

• Concurrent Real-Time Programming
- Asynchronous transfer of control (ATC) and any thread aborting mechanisms are disallowed.
- Use of *wait*, *notify*, and *notifyAll* methods is disallowed.

• Programming in the small
- Use of *continue* and *break* statements in loops is disallowed.
- All constraints, such as one used in a *for* loop, must be static.
- Compound expressions in parameter passing to methods must be eliminated.
- Expressions whose values are dependent on the order of evaluations should be disallowed.

All the rules and guidelines will facilitate or greatly ease the use of program analysis tools.

5. IMPLEMENTATION ISSUES
Along the same lines as the profile we present in this paper, it is essential to use a runtime environment (called RVM in this paper) that has been designed and implemented with highly dependable systems in mind. As mentioned earlier, however, programs based on our profile should be valid RTSJ programs and execute on a standard RTSJ platform with the same functional results (although not necessarily with the same response times).

In addition, tool support is essential to analyse code in terms of functionality and timeliness. A customised tool may be developed that incorporates all the rules and guidelines (see Appendix A of [23]) and throughout this paper. Such a tool may also be able to obtain the Worst-Case Execution Time (WCET) and worst-case memory consumption of each thread, thus enabling schedulability analysis [6, 18]. Standard Java tools or model checkers, such as ESC/Java [24] and Java Pathfinder 2 [9, 21] may be used, as well.

6. AN EXAMPLE PROGRAM
We present a simple and naive traction-control system that senses any difference between the front- and rear-wheel spin speeds, and reduces the engine output if the rear-wheels spin more quickly[4]. There is one periodic thread **spinMonitor** and one sporadic thread **powerCutHandler**, and as soon as an excessive rear-wheel spin is detected **spinMonitor** activates **powerCutHandler**. The real application logic is not given as the example is purely intended to illustrate how the profile is used.

```
import ravenscar.*;
…
public class TractionController extends Initializer {
  public void run()              // Initializer routine
  {
    // powerCutHandler
    SporadicEventHandler powerCutHandler =
      new SporadicEventHandler (
        new PriorityParameters(15),  // Priority:15
        new SporadicParameters(
          // Minimum interarrival time and buffer size
          new RelativeTime(333, 0), 5)
    (
    {
      // Event handler routine
      public void handleAsyncEvent()
      {
       // Logic for handling powerCutEvent event
       // i.e. either cut the engine power or brake
       // appropriate wheels
      }
    };

    final SporadicEvent powerCutEvent=
      new SporadicEvent(powerCutHandler);

    // spinMonitor
    PeriodicThread spinMonitor = new PeriodicThread(
      new PriorityParameters(10),    // Priority:10
      new PeriodicParameters(
        new AbsoluteTime(0, 0),      // Start time
```

---

[4] A rear-wheel drive car (e.g. a Formula 1 car) is assumed.

```
      new RelativeTime(333, 0)      // Period
    ),
   new Runnable() {            // Application logic
     public void run()
      {
       // Logic for checking front and rear wheel
       // spin speeds, i.e. obtain sensor readings
       // from front and rear wheels.
       // Once any excess of a predefined threshold
       // is detected, fire the following event
          powerCutEvent.fire();
      };
    }
   );
   spinMonitor.start();
  }

  public static void main (String [] args)
  {
    TractionController init = new TractionController();
    init.start();
  }
}
```

7. RELATED WORK

There have been a few subsets or profiles for Java suggested in the literature[5]. None of them, however, is as complete or analytical as the Ravenscar-Java profile described in this paper; they are surveyed below.

7.1. Sequential subset of Java

Bentley [5] defines a sequential subset of Java after assessing the language. The subset consists of 21 rules that are effectively derived from [19], [28] and his assessment. All the rules are categorised into six groups, as shown below with a summary of rules for each group.

• Rules Concerned With Verification
Multithreading, as well as method and constructor overloading is not allowed.

• Rules Concerned With Comments
Comments shall not be nested.

• Rules Concerned With Predictability
Variables or objects must be statically initialised (by constructors of appropriate classes). All constraints, such as those used in *for*-loops, must be static. The *continue* and *break* statements shall not be used, except to terminate the cases of a switch statement, for which a break statement is required for every non-empty case clause. The *return* statement should only appear as the last statement of a method. Further, methods must not have any side effects and not be recursively invoked. The result of a method should never be an unconstrained array type object.

• Rules Concerned With Constants
Octal constants (other than zero) shall not be used.

• Rules Concerned With Identifiers
All identifier names must be unique.

---

[5] In fact, there are subsets of Java defined for other purposes than for use in high integrity systems, for example, [14].

• Rules Concerned With Operators
All right-hand operands of the logical operator && and || shall not contain any side effects, and assignment operators must not be used in expressions that return Boolean values, for example, in *if ((x=1) != y)*. Bitwise operations, including bitwise shifts, shall not be performed on signed integer types, while the evaluation of integer expressions should not lead to wrap-around.

Whereas this subset will undoubtedly help produce analysable and predictable *sequential* programs, it can be criticised for its restriction on multithreading, one of Java's inherent elements. Without the language-level support for multithreading and all the associated synchronisation mechanisms, Java may not be considered as a great evolution from its predecessors. In addition to this, the subset also fails to address issues on the object-oriented programming model of the language, as well as real-time issues.

7.2. Profile for high integrity Real-Time Java programs

Puschner and Wellings [31] suggest a *Ravenscar*-like profile for the Real-Time Specification for Java [7], and the following is a brief summary of each of the key areas.

• Threading Model
There are two execution phases, i.e. *initialisation* and *mission* phases. In the initialisation phase, all necessary threads, event handlers, and memory objects are created in a non time-critical manner. No threads will be allowed to start until the top-priority thread with *main()* method finishes its execution. In the mission phase, threads may not change their own or other thread's priority except when forced by the underlying implementation of the priority ceiling protocol. Sporadic or event-triggered activities are implemented as event handlers, and only one handler is allowed per event. All periodic threads must be an instance of *NoHeapRealtimeThread* class and need to invoke *waitForNextPeriod* method to delay execution until the start of their next periods. Asynchronous Transfer of Control (ATC), overrun and deadline-miss handlers, and delay statements are not supported by the profile; nor is dynamic class loading during the mission phase.

• Concurrency
The *synchronized* methods and blocks are allowed, and the priority ceiling protocol should be implemented in the run-time system in order to avoid deadlocks. For similar reasons, *wait*, *notify*, and *notifyall* are not supported, avoiding any queue management.

• Memory Management and Raw Memory Access
The heap-based garbage collection mechanism of Java is not supported; instead, only immortal memory and linear-time scoped memory are supported. Immortal memory is used by default to create objects during the initialisation phase, but is not allowed for further object creation afterwards. In addition, all other memory objects must only be created in the initialisation phase. The RTSJ classes for raw memory access are also supported, so that device drivers, memory-mapped I/O, and other low-level functions can be programmed.

• Time and Clock
All the RTSJ classes for the representation of time and real-time clocks are included while the timer classes are not.

The profile is primarily focused on leaving out complex features of the RTSJ. However, little attention is paid to the Java's sequential language constructs (unlike [5]) and object-orientation features that can be problematic in performing various static analyses. Furthermore, the profile is not consistent with the current version of the RTSJ.

7.3. High integrity profile by the J Consortium

A sub-committee has been formed within the Real-Time Java Working Group of the J Consortium to produce a high integrity profile based on the Real-Time Core Extensions [20]. The profile has not been released yet, but according to Dobbing [13] it is likely to resemble the Ravenscar profile for Ada95 [11]. It consists of four main themes: partitioning, memory management, concurrency, and error recovery, respectively. Up-coming information will be found at http://www.j-consortium.org/hip/index.shtml.

• Partitioning
The main idea has developed from the necessity to isolate critical code and data from non-critical ones by means of a firewall, so that less-trusted code will never be able to interfere with high integrity programs. No exchange of objects or dynamic loading will be allowed across the firewall. This idea also extends to the temporal requirements of such software, i.e. temporal firewall, which means deadlines of critical threads must be met.

• Memory Management
The automatic garbage collection and any memory compaction mechanism are not supported. There are three memory allocation strategies, which are
· stack allocation for method local objects that are automatically reclaimed
· fixed size "allocation contexts" for local objects in each thread
· global allocation at initialisation time for immortal objects.

• Concurrency
Three types of priority-based tasks are supported, namely, periodic, sporadic, and interrupt tasks. Additionally, the profile defines a subclass of the basic *CoreTask* that must explicitly be started by another thread. All threads are created at program start-up, e.g. as part of the initialisation code for classes, and it is not allowed to declare a thread class as an inner class.

Shared resources and inter-thread synchronisations are managed through *protected objects*, which rely on the underlying implementation of the Priority Ceiling Protocol. However, no mutual exclusion locks or synchronised methods are supported in the profile. Further, all the asynchronous thread-to-thread operations, including *stop, setPriority, suspend, resume* methods, and event-driven Asynchronous Transfer of Control (ATC) mechanisms, are not permitted. Synchronised objects and counting semaphores are also not supported.

• Error Recovery
The standard exception handling mechanism of Java (i.e. *throw-catch* clause) is maintained. It also supports access to specific physical addresses to allow objects to be mapped, in order to, for example, save program state for fast recovery purposes.

Like the one proposed in [31], this profile is mainly focused on sub-setting the Real-Time Core Extensions [20], but does not address issues on the use of problematic language constructs and object-orientation features of Java.

7.4. Formal subsets
Drossopoulou *et al.* [14] define three formal subsets of Java, i.e. that of the source language (Java^s), high-level representation of bytecode (Java^b), and enriched version of Java^b (Java^r). They present operational semantics, type system, and a proof of type soundness for the subsets.

Java^s is a substantial subset of the Java programming language, and it includes some primitive types, interfaces, classes with instance variables and instance methods, inheritance, hiding of instance variables, overloading and overriding of instance methods, arrays, implicit pointers and the *null* value, object creation, assignment, field and array access, method call and dynamic method binding, exceptions and exception handling [14].

In order to observe run-time behaviours of programs in Java^s, they are formally converted into Java^b and Java^r respectively, which are high-level representations of bytecode with all necessary compile-time type information. Having done this, it is possible to obtain operational semantics of each high-level language construct and prove the soundness of the type system of the source-level subset, Java^s.

While these subsets contain many important language constructs of Java that are often omitted in other formal subsets (e.g. exceptions), they still overlook some of Java's inherent features, such as the multithreading and synchronisation models. [17] surveys formal subsets and approaches aimed at improving the safety of Java programs.

8. CONCLUSIONS

In this paper, we have presented the *Ravenscar-Java profile*, a high integrity profile for real-time Java. This restricted programming model excludes language features with high overheads and complex semantics, on which it is hard to perform timing and functional analyses. Several classes in the RTSJ are redefined, and a few new classes are added, all resulting in a compact, yet powerful and predictable computational model for the development of software-intensive high integrity real-time systems.

The profile is categorised into three areas, i.e. Programming in the large, Concurrent real-time programming, and Programming in the small. These are then structured based on the guideline framework developed by the U.S. Nuclear Regulatory Commission, which derives many important attributes from existing standards and is specific to high integrity systems. Various rules and guidelines, centred around the *reliability* attribute, are given in each of the three following sub-attributes:

- predictability of memory utilisation,
- predictability of timing, and
- predictability of control and data flow.

A simple example illustrating the use of our profile was also provided in Section 6, before we reviewed four existing subsets of Java or the RTSJ. Most of the subsets, however, overlook some important elements of the language, for example, multi-threading and object-oriented programming model. Thus, many of the advantages of Java are lost.

We believe that our profile is expressive enough to accommodate today's demanding requirements for a powerful

programming model, yet concise enough to facilitate the implementation of underlying platforms or virtual machines with great ease. A subset of Java and the RTSJ, along the lines presented in this paper, would be a powerful motivation to develop high integrity systems in Java, rather than in a subset of C, C++ or Ada.

9. ACKNOWLEDGEMENTS

10. REFERENCES

[1] A. W. Appel, *Protection against untrusted code: The JIT compiler security hole, and what you can do about it*, http://www-106.ibm.com/developerworks/library/untrusted-code/, as of January 2001.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*, Software Engineering Journal, 8(5), 284-92, 1993.

[3] A. Azevedo, A. Nicolau, and J. Hummel, *Java Annotation-Aware Just-In-Time (AJIT) Compilation System*, ACM 1999 Java Grande Conference, 1999.

[4] J. Barnes, *High integrity Ada: the SPARK approach*, Addison Wesley, 1997.

[5] S. Bentley, *The Utilisation of the Java Language in Safety Critical System Development*, MSc dissertation, Department of Computer Science, University of York, 1999.

[6] G. Bernat, A. Burns, A. Wellings, *Portable Worst Case Execution Time Analysis using Java Bytecode*, In Proceedings of the 12th EUROMICRO conference on Real-Time Systems, 2000.

[7] G. Bollella, et al, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[8] J. P. Bowen and M. G. Hinchey, *High Integrity System Specification and Design*, Springer-Verlag, 1998.

[9] G. Brat, K. Havelund, S. Park, and W. Visser, *Model Checking Programs*, In IEEE International Conference on Automated Software Engineering (ASE), Sep. 2000.

[10] B. M. Brosgol, S. Robbins, and R. J. Hassan II, *Asynchronous Transfer of Control in the Real-Time Specification for Java*, In Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2002.

[11] A. Burns, B. Dobbing, and G. Romanski, *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*, In L. Asplund, editor, Proceedings of Ada-Europe 98, LNCS, Vol. 1411, pages 263-275, Berlin Heidelberg, Germany, Springer-Verlag 1998.

[12] A. Burns, and A. Wellings, *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*, 3rd Ed., Addison Wesley, 2001.

[13] B. Dobbing, *The Ravenscar Profile for High Integrity Java Programs?*, ACM Ada Letters, Vol. 21, Issue. 1, March 2001.

[14] S. Drossopoulou and S. Eisenbach, *Describing the Semantics of Java and Proving Type Soundness*, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[15] Li Gong, *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley, 1999.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd Ed., Addison Wesley, 2000.

[17] P. H. Hartel and L. Moreau, *Formalizing the Safety of Java, the Java Virtual Machine, and Java Card*, ACM Computing Surveys, Vol. 33, No. 4, December 2001.

[18] E. Y-S Hu, G. Bernat, and A. Wellings, *Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems*, In Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2002.

[19] A. Hutcheon, B. Jepson, D. Jordan, and I. Wand, *A Study of High Integrity Ada: Language Review*, Technical Report SLS31c/73-1-D, Version 2, York Software Engineering, University of York, July 1992.

[20] J Consortium, *International J Consortium Specification: Real-Time Core Extensions*, Revision 1.0.14, www.j-consortium.org, September 2000.

[21] *Java PathFinder*, http://ase.arc.nasa.gov/visser/jpf/, last accessed in April 2001.

[22] J. Kwon, A. Wellings, and S. King, *Assessment of the Java Programming Language for Use in High Integrity Systems*, York Technical Report (YCS 341), Department of Computer Science, University of York, 2002, available at http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-341.pdf.

[23] J. Kwon, A. Wellings, and S. King, *Ravenscar-Java: A High Integrity Profile for Real-Time Java*, York Technical Report (YCS 342), Department of Computer Science, University of York, 2002, available at http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-342.pdf.

[24] K.R.M. Leino, G. Nelson, and J.B. Saxe, *ESC/Java User's Manual*, SRC Technical Note 2000-002, Compaq Systems Research Center, Palo Alto, CA, 2000.

[25] N. G. Leveson, *Software Safety: Why, What, and How*, Computing Surveys, Vol. 18, No. 2, ACM, June 1986.

[26] N. G. Leveson, *Software Safety in Embedded Computer Systems*, Communications of the ACM, Vol. 34, No. 2, February 1991.

[27] C. Liu and J. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*, Journal of ACM, 20(1), 46-61, 1973.

[28] The Motor Industry Software Reliability Association, *Guidelines for the use of the C language in vehicle based software*, The Motor Industry Research Association, 1998.

[29] H. Hetcht, M. Hecht, S. Graff, et al, *Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems*, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997.

[30] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan, *Evaluation of Safety-Critical Software*, Communications of the ACM, Vol. 33, No. 6, June 1990.

[31] P. Puschner and A. J. Wellings, *A Profile for High Integrity Real-Time Java Programs*, In Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2001.

[32] I. Sommerville, *Software Engineering*, 6th Ed., Addison Wesley, 2000.

[33] Sun Microsystems®, *Java™ 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices*, White paper, Sun Microsystems® 2000.