Ravenscar-Java: Java Technology for High-Integrity Real-Time Systems

Jagun Kwon

A thesis submitted as a partial fulfilment for the degree of
Doctor of Philosophy of the University of York

June 2006

Department of Computer Science
University of York
Heslington
York, YO10 5DD
England

Acknowledgements

It has been a long journey. During the course of the years, so many things happened in and out of my life, and I have been learning the hard way how to handle all those obstacles in life. Along the long way, my supervisor Prof. Andy Wellings has always been there for me. I cannot thank him enough for his invaluable support and patience.

I must also thank my assessor, Dr. Neil Audsley, for his constructive feedbacks and discussions. His input often played an important role in testing my ideas in this thesis. Furthermore, the Real-Time Systems Research Group at York provided a lively and accommodating environment, for which I am very much grateful. Especially, I thank Prof. Alan Burns, Erik Hu, Guiem Bernat, Hao Cai, Alexandros Zerzelidis, Andrew Borg, Iain Bate, Ian Broster, Rachael Baker, Michael Ward, Armando Aguilar-Soto, Yang Chang, Min-Sung Kim, Rob Davis, Michael Bennet, Ameet Patil, Konstantinos Bletsas, Osmar Santos, and Sue Helliwell.

The work presented in this thesis was funded by the European Commission's HIJA (High Integrity Java Applications) project and the EPSRC under the award number GR/M94113. The author gratefully acknowledges their generous support.

My special thanks go to the York Korean Church, Korean Students' Association, and my Golf buddies! I appreciate all their friendships and encouragement. Finally, I would like thank Kyungeun, my parents and family for their love and support!

Bless God for everything he has done for me!

Abstract

High integrity systems are those that must not fail. They play key roles in our society, for instance, in transportation and tele-banking. By all means, we strive to engineer the best possible approaches, so that the possibility of failure is kept to a minimum. Over the past few decades, such systems have become more and more complex, and the software technology is employed to better manage the complexity of application domains, reduce production cost, and increase flexibility in design and implementation. Choosing the right programming language is thus an important factor to any successful development and maintenance of modern high integrity systems.

In this thesis, Java with the Real-Time Specification for Java is examined. After gathering language selection criteria from influential standards and guidelines, a framework of 23 assessment criteria is developed that also reflect on advances in modern programming languages. The categorised criteria are then used to evaluate how close Java comes to the ideal requirements from various standards. The results are mixed; although Java scored high in several areas, it fails to do so in other areas mainly due to the complex features and overheads associated with the language and run-time.

The Ravenscar-Java profile is motivated by the assessment. By subsetting the full language, the profile overcomes most of the shortcomings revealed in the assessment, yet still maintaining the advantages of Java and RTSJ. The profile is not just a collection of APIs, but includes new rules and guidelines specific to Java and RTSJ. With the ordered computational model and new classes added, static schedulability analysis now becomes possible while concurrency related errors, such as deadlocks and race conditions, are minimised. Sequential features of Java are also addressed.

Additional extensions to the profile are then introduced, i.e., a new dynamic

memory allocation scheme, and annotations for documenting developer's intention and program proof. The first extension, called Single Nested Scoping, is proposed for increased flexibility in reusing memory areas, thereby reducing the overall memory footprint required. Code can be annotated to confirm assignment relationships of objects in different memory areas, maximum loop bounds, and so on. Loop bounds are vital in analysing the worst-case execution time and worst-case memory usage.

Having presented the profile and its extensions, the impact the profile has on analysis techniques are investigated. First, a deeper understanding is gained on what is meant by testing conformance. Second, we consider important analysis techniques regarding memory usage, shared objects, and memory area access. It is shown how to calculate the worst case memory usage of each thread. In doing so, we develop a new tree structure that consists of multiple IOT (Instantiated Objects Table) nodes. By traversing the tree, one can obtain the worst-case path by adding the cumulative sizes of objects on every path and comparing them. The tree structure is also beneficial in detecting shared objects as well as locating unnecessary assignment checks.

The profile is then assessed from two different perspectives, i.e., how well it meets the requirements against the criteria developed, and how expressive it is in developing realistic applications. First, the results from the assessment are convincing since the profile scored the highest possible ratings in most of the areas. Second, the case study demonstrates the expressive power of the profile. It is intuitive to assume the profile as a limiting factor to the development of complex software. However, we will show the profile's potential as a base development language for high integrity applications, where an appropriate balance between expressiveness and predictability/analysability is the key target to achieve.

Declarations

The following papers were written by the author in collaboration with Prof. Andy Wellings under his supervision. Chapter 2 and 3 is based on Paper 5 and 8, while Chapter 4 is derived from Paper 1, 2, 3, 4, 6 and 7. Paper 3, 4 were the key input to Chapter 5. The papers are in the order of publication time.

1. Motivations and Support for Single Nested Scoping in the Ravenscar-Java Profile
by J. Kwon and A. Wellings
To be published in the Proceedings of the 3rd Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS) April, 2006.

2. Ravenscar-Java: A High Integrity Profile for Real-Time Java
by J. Kwon, A. Wellings and S. King
Published in the Journal of Concurrency and Computation: Practice and Experience, John Wiley & Sons Ltd (2005; 17:681-713) Feb. 2005.

3. Memory Management base on Method Invocation in RTSJ
by J. Kwon, A. Wellings
Published in Lecture Notes in Computer Science (LNCS) 3292, Proceedings of the OTM 2004 Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES) Oct. 2004.

4. Predictable Memory Utilization in the Ravenscar-Java Profile
by J. Kwon, A. Wellings and S. King
Published in the Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Hakodate, Hokkaido, Japan, 2003.

5. Assessment of the Java Programming Language for Use in High Integrity Systems
by J. Kwon, A. Wellings and S. King
Published in ACM SIGPLAN Notices, April, 2003.

6. Ravenscar-Java: A High Integrity Profile for Real-Time Java
by J. Kwon, A. Wellings and S. King
Published in the Proceedings of the ACM Joint Java Grande-ISCOPE Conference, Seattle, U.S. 2002.

7. Ravenscar-Java: A High Integrity Profile for Real-Time Java
by J. Kwon, A. Wellings and S. King
Published as a technical report, YCS 342, Dept. of Computer Science, University of York, UK.

8. Assessment of the Java Programming Language for Use in High Integrity Systems
by J. Kwon, A. Wellings and S. King (May, 2002)
Published as a technical report, YCS 341, Dept. of Computer Science, University of York, UK.

Table of Contents

Chapter 1. Introduction

Today, computing technologies are increasingly used in high integrity systems, where failure can cause loss of life, environmental harm, or significant financial penalties. Examples of such systems include space shuttles, nuclear power plants, automatic fund transfers and medical instruments. They typically have hard real-time and stringent safety requirements, implying that correct computational results must be produced in a strictly timely manner. Any missed deadlines or malfunction of any components will have a direct impact on the safety of the whole systems.

Within such systems, there has been a growing trend of using software as opposed to traditional electro-mechnical subsystems, and the role of programming languages that we employ in developing such software is becoming ever more critical. Hence, choosing the right language with suitable features that reflect technological advances as well as a strong user base is vital. For this purpose, there have been a number of standards and guidelines that help select a high integrity programming language.

In this thesis, we consider Java – an object oriented programming language with an unsurpassed popularity in many application areas. Bearing in mind its enormous success in recent years and its advantageous features at hand, we believe

Java can become a major candidate for the next generation high integrity programming language. In particular, with the development of the Real-Time Specification for Java (RTSJ)[1] and Java 2 Micro Edition Specifications (J2ME), it is now targeting at embedded real-time systems, many of which are of high integrity in nature.

1.1. High Integrity Systems

As a modern society, we rely on many high integrity systems that must not fail. These systems can subjectively be interpreted as *safety-critical*, *security-critical*, *time-critical*, *mission-critical*, *cost-critical*, or collectively any combination of them depending on the essential role that they play for an associated system or the environment [Bowen1998, Isaksen1997]. Typical examples and sectors that embed high integrity systems include

- Space programs (e.g. space shuttles, space stations, satellites),
- Military applications (e.g. jet fighters),
- Transportations (e.g. aircrafts, air traffic control systems, interlocking systems for trains),
- Medical instruments
- Banking systems
- Nuclear power generation, and
- Industrial manufacturing systems.

In constructing these systems, a great deal of emphasis is placed on *dependability*, which is used as a synonym for the term '*high integrity*' [Storey1996].

[1] This thesis assumes that the reader is familiar with Java and the RTSJ. For those unfamiliar with the RTSJ, please refer to [RTSJ2005] or [Wellings2004b].

"***Dependability** is a property of a system that justifies placing one's reliance on it.*" [Storey1996]

This can be characterised as having the following four properties [Storey1996, Laprie1992, Leveson1986-1995]:

- Reliability – "*... is the probability of a component, or system, functioning correctly over a given period of time under a given set of operating conditions,*"
- Availability – "*The availability of a system is the probability that the system will be functioning correctly at any given time,*"
- Safety – "*... is a property of a system that it will not endanger human life or the environment,*" and
- Security – "*... is the ability of a system to manage, protect, and distribute sensitive information.*"

There are numerous generic or sector-specific guidelines that state requirements centered around the notion of dependability. Some of the well-known[2] are

[2] It is impossible to list all standards here, but interested readers may want to look at [Ippolito1995 Appendix A] for a comprehensive list, although the document is mainly focused on Software hazard analysis techniques of those standards.

Standards	Sectors
• UK MoD DEF Stan. 00-55, 00-56	U.K. Defence
• MIL-STD-882, DoD STD-2167A	U.S. Defence
• RTCA/DO-178	Airborne systems
• MISRA	Motor industry
• STANAG 4404	Munitions related
• IEC 60880	Nuclear power industry
• IEC 61508	Generic electrical/electronic systems
• NIST FIPS PUB 73	Security of computer applications

Most of these standards raise issues on the use of software and software safety analysis techniques. The simple reason for this is that modern high integrity systems are increasingly dependent on the computer and software technology as opposed to traditional electro-mechanical subsystems [Leveson1986, Leveson1991]. Some obvious benefits of exploiting software are [Leveson1986, Leveson1991, Parnas1990, Bowen1998]

- Improved functionality
- Increased flexibility in design and implementation
- Reduced production cost
- Enhanced management of complexity in application areas.

Whilst these benefits seem far more appealing over those of traditional counterparts, the use of software, however, inherently imposes a great responsibility in designing, developing and testing software systems. That is, due to its discrete nature and enormous number of (sometimes unmanageable) internal states, ensuring the safety and correctness of a large piece of software has always been a great challenge. Some advanced features of high-level programming languages or operating systems (e.g. multi-threading and exception handling) further complicate the verification or exhaustive testing of complex software systems [Parnas1990, Leveson1986, Leveson1991, Isaksen1997]. Therefore, a need for a powerful, cost-effective programming and validation mechanism naturally arises.

1.2. Java, RTSJ, and High Integrity Applications

The current pervasiveness of the term *Java* throughout academia, media, or even non-specialists confirms that Java is now an established programming environment in the software industry. Its relatively familiar linguistic semantics, the adoption of well-understood approaches to managing software complexity, and support for concurrency seem to have contributed towards its popularity. In addition to this, the advent of the Internet unquestionably boosted Java as a major development framework for a wide range of applications.

Initially designed with embedded systems in mind, Java's main goal was to provide engineers with a reliable and cost-effective platform-independent environment. The burden of learning a new language is kept to the minimum for existing C and C++ programmers, while helping them to discover errors earlier by means of strong type checking, array-bound checking, null-pointer checking, and so on [Gosling2000]. Further, its support for concurrency, i.e., multi-threading and synchronisation mechanisms, together with the use of portable code (*a.k.a.* the *bytecode*) opens up a number of possibilities for many other applications, including high integrity systems.

However, despite all these valuable features, Java has been criticised for its unpredictable performance as well as some security concerns [Appel1999, Azevedo1999, Amme2001]. 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 [Gong1999, Appel1999]. These fears make Java and its associated technology unsuitable for the development of high integrity systems.

Recently, however, there has been a major international activity initiated by *Sun MicroSystems* to address the limitations of Java for real-time and embedded systems. The *Real-Time Specification for Java* (RTSJ) [Bollella2000a] attempts to minimise any modification to the original Java semantics and yet to define additional concepts and 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 very complex to implement and have high overheads at run-time[3]. Software produced in this framework is also difficult to analyse with all the complex features, such as the asynchronous transfer of control (ATC), dynamic class loading, and scoped memory areas. This all prevents confident use of the full language and its associated technology in high integrity applications. The purpose of the work presented in this thesis is to rise to this challenge.

[3] Sources of run-time overhead in RTSJ include interactions between the garbage collector and real-time threads, assignment rule/single-parent rule checks for objects in different memory areas, and asynchronous operations.

1.3. Motivation and Central Proposition

As outlined above, the key motivation of this work comes from the idea that, despite certain shortcomings, Java can become a more reliable and productive language for developing high integrity software. In other words, we argue that

"use of a subset of Java augmented with a subset of the RTSJ and a set of static analysis techniques is an acceptable means for developing high integrity software. This will facilitate the production of more analysable and predictable Java software and reduce run-time overheads."

Consequently, we propose an approach in which a restricted programming model (or a profile) and a number of useful analysis techniques are developed. As a high integrity programming environment, the profile will offer a set of rationalised guidelines and a predictable computational model that will keep run-time overheads to a minimum while maintaining most of the advantages of Java over other languages. We will also argue that the profile promotes the production of more efficient and trustworthy analysis techniques because such techniques need not deal with difficult features to examine.

Since the profile removes language constructs and features that are complex to analyse and have high overheads, tools can analyse programs more efficiently with more confidence. This approach will make possible the verification of Java programs for their adherence to the safety rules provided by our profile and the RTSJ *per se*, as well as those naturally implied by the subject application. An emphasis is placed on statically assuring functional safety. This means that our approach will help remove unintended erroneous behaviours that a program may exhibit at run-time by analysing the code off-line. That way, we believe more predictability and reliability can be achieved. This thesis work will also explore a handful of novel analysis techniques for real-time Java programs; for instance, memory consumption analysis and dynamic memory access check analysis.

1.4. Scope and Assumptions

This thesis is only concerned with implementation and static analysis of high integrity software. More specifically, we propose the development of a tailormade language profile and program analysis techniques. Other more general issues including requirement analysis, design, and certification of high integrity software are not directly dealt with, nor are underlying implementation technologies for virtual machines and operating systems [Cai2004 and 2005]. Furthermore, we base our work on the previous research on fixed-priority pre-emptive scheduling and worst case execution time (WCET) analysis [Audsley1993, Burns2001, Liu1973].

The main focus is on *reliability* of the language[4] and analysis techniques, such that they will facilitate current state-of-the-art practices in the new programming environment. That way, functional and temporal *predictability* of programs will be achieved.

1.5. Contributions

Major contributions of the thesis include the following.

1. *A survey and analysis of programming guidelines/standards for high integrity systems.* Requirements on languages from the relevant literature are identified

[4] Reliability, in the sense of a programming language, means that such a language will support early detection of errors, have an unambiguous definition of syntax and semantics, and is well understood. That way, programs written in that language will be more probable to be reliable.

and a framework of assessment criteria is also developed. The framework is only concerned with modern languages and systems, thus excluding irrelevant criteria (for example, use of a particular character set). We also relax some restrictions, such as concurrency, in order to reflect on advances in analysis techniques.

2. *An assessment of Java and RTSJ for high integrity systems according to the criteria.* It is up-to-date in a sense that no other assessment has been performed on the combination of the new Java 1.5 release and the real-time specification. There are, however, assessments carried out on Java in the past, for instance, [Bentley1999].
3. *The development of a high integrity Java profile (called Ravenscar-Java) with rationalised guidelines.* The profile is developed based on the requirements and guidelines from influential standards identified previously. Earlier approaches laid the foundation for this work, but they are either inconsistent with the current RTSJ release and Java, or do not consider certain features of the language (e.g., concurrency related ones).
4. *An evaluation of the profile in terms of its expressive power and adherence to standards.* First, the framework of criteria is revisited to check where the profile stands, and improved areas are highlighted. Second, a realistic case study is developed to provide experiential evidence that the profile is expressive enough to implement such systems with a similar or even higher complexity level.
5. *An investigation into novel analysis techniques for the Ravenscar-Java profile*, in particular, those related to memory usage, conformance check, and program correctness in terms of sharing objects.

1.6. Outline of the Thesis

In Chapter 2, related work will be surveyed, which includes high integrity standards and guidelines, high integrity subsets of Java and Ada, as well as relevant analysis techniques for our work.

Chapter 3 will set up a framework of criteria derived from the important standards and guidelines introduced earlier. Then the Java language and RTSJ will be assessed against the framework, and intermediate conclusions will be drawn, which will then result in the motivations for a new language profile.

Based on the assessment and guidelines, a new language profile will be proposed in Chapter 4. The profile will contain various rules and guidelines specific to Java, plus a few new classes. Appropriate rationales for the decisions we have made will also be given.

Chapter 5 will then discuss several possible analyses that can be applied on Ravenscar-Java programs in a more straightforward manner (compared to normal Java and RTSJ programs), including

- Conformance test,
- Java Memory Model Integrity Analysis,
- Worst Case Memory Usage Analysis,
- Memory Area Access Analysis and Optimisation, and
- Impacts on Real-Time Scheduling.

The whole work will then be evaluated in Chapter 6 in terms of its effectiveness in producing predictable high integrity software. The assessment criteria will be revisited to examine how the profile fits within the framework. A realistic case study is then to be introduced to evaluate the expressive power of the profile; we use European Space

Agency's Packet Utilization Standard (PUS) services, which is a reuse framework of standard services for on board systems.

Chapter 7 will draw a few conclusions, and discuss future research directions.

Chapter 2. Related Work

The development of high integrity systems is driven by certification standards and guidelines. Since computer software plays an essential role in numerous applications these days, many of the standards place requirements on the use of software implementation languages. The choice of language is important because those with appropriate features can benefit software development in many ways. Compared to the Assembly, for instance, the C language provides a number of abstractions that remarkably improve productivity and reduce programming errors. More benefits can be acquired by using modern object-oriented programming languages, such as Java, C++ and Ada [Burns2001, Gosling2000, Stroustrup1997].

A language's user base is also vital. Even if a language has rich features, it may not be an option if no expertise can be readily found. A popular language is often likely to have many supporting compiler/tool vendors, experts and designers, and a large community around. This means that the language has gone through a great deal of real-life test and common bugs are already known, which is an enormous advantage over developing an in-house language, compiler and tools.

In this chapter, we survey high integrity standards and guidelines relevant to software development and programming languages. Then a few critical languages will be reviewed, including Ada and Java, before we move on to program analysis techniques.

2.1. Standards and Guidelines

As mentioned previously, it is neither possible nor desirable to list all standards and guidelines here in one place. We survey only those that are relevant in the context of high integrity software or have a direct impact on selecting implementation languages for high integrity systems.

2.1.1. General Requirements of Programming Language

A study by Bentley [Bentley1999] summarises some of the well-known requirements of programming language for the development of high integrity systems including works by [Carré1990], [Cullyer1991], [USDoD1978], [USDoD1990] and [Hutcheon1992]. It carries out an assessment on Java against all the requirements, producing a series of comprehensive rationales. A subset of the language is also proposed, but only sequential features are included.

The outcome of the study is compatible to a large extent with our objective in this chapter as the requirements are still of significant importance these days, and the chosen language is Java. Therefore, we consider it as our starting point for a more complete and up-to-date assessment of the language. Below is a summary of the requirements used by Bentley [Bentley1999].

• Carré et al [Carré1990] identify six factors that can have an influence on a programming language's suitability for use in high integrity systems. These factors, although simple and abstract, have been the guiding principles for other more elaborate requirements. These are summarised by [Storey1996] as

- Logical soundness: is there a sound, unambiguous definition of the language?
- Simplicity of definition: are there simple, formal definitions of the various language features? Complexity in these definitions results in complexity within compilers and other support tools, which can lead to errors.
- Expressive power: can program features be expressed easily and efficiently?
- Security and integrity: can violations of the language definitions be detected before execution?
- Verifiability: does the language support verification, that is, proving that the code produced is consistent with its specification?
- Bounded space and time requirements: can it be shown that time and memory constraints will not be exceeded?

• Cullyer et al [Cullyer1991] define a checklist of eleven factors to help establish if a language has appropriate characteristics. The factors or questions to ask are

- Wild jumps: can it be shown that the program cannot jump to an arbitrary memory location?
- Overwrites: are there language features that prevent an arbitrary memory location being overwritten?
- Semantics: are the semantics of the language defined sufficiently for the translation process needed for static code analysis?
- Model of maths: is there is a rigorous model of both integer and floating point arithmetic?
- Operational arithmetic: are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?

- Data typing: are the means of data typing strong enough to prevent misuse of variables?
- Exception handling: if the software detects a malfunction at runtime, do mechanisms exist to facilitate recovery?
- Safe subsets: does a subset of the language exist which is defined to have properties that satisfy these requirements more adequately than the full language?
- Exhaustion of memory: are there facilities to guard against running out of memory at runtime?
- Separate compilation: are facilities available for separate compilation of modules, with type checking across the module boundaries?
- Well understood: will the designers and programmers understand the language sufficiently to write safety-critical software?

These questions, however, are high-level and do not cover some detailed issues like those in the Steelman requirements [USDoD1978] below.

• The Steelman requirements [USDoD1978] are both extensive and technically detailed. It was established by the U.S. Department of Defence after a number of reviews and refinements by military and civil communities in order to evaluate existing languages. This eventually led to the development of Ada, which satisfies all the requirements. Although most of the Steelman requirements are still desirable today for general-purpose languages when efficiency and reliability are important concerns, it does not reflect modern language features and paradigms, for example, object orientation [Wheeler1997]. Noteworthy areas of the requirements include

- · Language design aims
- · Syntax, expressions and types
- · Control structures, functions and procedures
- · Input-output control, parallel processing
- · Exception handling
- · Support for the language.

For the whole list of the requirements, see [USDoD1978] or [Bentley1999].

• [USDoD1990] shows a set of new and revised requirements for Ada9X, based on long industrial experiences with the original Ada83. It incorporates new language features and support for real-time, safety-critical, distributed systems by means of additional annexes. Major areas cover

- Issues on standardisation, understandability, efficiency in execution and storage management
- New language paradigms including object orientation (via type extension)
- Real-time requirements including alternative scheduling policies, synchronous transfer of control, and asynchronous communication
- Parallel and distributed processing
- Safety-critical and trusted applications.

A few of the requirements are specific to Ada, and may not be applicable to other languages.

• The work by York Software Engineering, British Aerospace and the U.K. Ministry of Defence [Hutcheon1992] is specific to safety-critical applications with emphasis on the military requirements of the INTERIM Defence Standard 00-55 [UKMoD1991].

Two levels of requirements are defined (i.e., one to represent mandatory and the other optional features that a language should have) and subsequently used to assess Ada9X in [Hutcheon1992]. The level one, mandatory requirements are

- L1 A high integrity software language must be well-understood, simple to understand, simple to learn, simple to use, simple to implement and simple to reason about.
- L2 A programming language for writing high integrity software must provide features appropriate to that application domain.
- L3 Prior to execution, it must be possible to predict the following properties of a program written in a high integrity software language:
    - functionality;
    - timing;
    - resource usage;
    - failure behaviour.
- · L4 It must be possible to verify that a program written in a high integrity software language is correct with respect to a specification expressed in a formal notation.
- · L5 There must be a high level of assurance in a high integrity software language's compilation system and associate tools.

Some of the requirements are rather abstract in that different interpretations could be derived. The second level includes optional requirements that should enrich the language's effectiveness. It covers issues on standardisation, portability, modularity, abstraction, error handling, concurrency, low-level input/output, strong typing, code/run-time system verification, and optimisation.

2.1.2. Additional Requirements

Along with the requirements listed above, we also consider the following guidance or standards because of their significance in systematically capturing requirements and language features.

• The Ada95 Trustworthiness Study [Craigen1995, Saaltink1997a, Saaltink1997b] is broad and analytical in that it defines a concrete framework for language analysis based on important standards, evaluates each language feature of Ada95 against the framework, and produces comprehensive guidance for the use of Ada95 in the development of high integrity systems. It first identifies four main themes for analysis, which are predictability, analysability, traceability, and engineering, and these themes lead to the development of ten analytical categories and ratings in each category, as shown below.

Categories	Ratings	Examples
Run-time Support Needed	1. No or minor run-time support. 2. Some run-time support. 3. Significant run-time support.	1. Scalar types 2. Simple tasking 3. Asynchronous Transfer of Control
Functional Predictability	1. Exact, which gives only one outcome. 2. Bounded, which gives only a small set of possible outcomes, which could be a few possible results or results within a small range. 3. Unpredictable, for all other cases.	1. Discriminants 2. Access types 3. Generalized access types
Timing Predictability	1. Tightly bounded, where the time-to-execute can be expressed in terms of a formula over the data, number of iterations, etc. 2. Loosely bounded, where a maximum time to execute can be determined, but actual execution times are usually much better. 3. Unpredictable, where we do not know how to predict the time bound.	1. Type conversion of numerated types 2. Arrays 3. Case statements

Space Usage Predictability	1. Exact, where we can develop a formula to determine exact memory usage. This requires implementation information on use of temporaries, stack, etc. 2. Worst-case analysis, where we can bound the space used, both immediately and over time. 3. Unpredictable.	1. Loop statements 2. Subprogram declarations 3. Task units and task objects
Formal Definition	1. Existing definition (for Ada83 and no changes to Ada95). 2. Potentially definable, where a definition exists in Ada83 but changes mean a re-definition is needed, or definitions exist in other languages. 3. Unknown.	1. Exception handlers 2. Static expressions and subtypes 3. Return statement for functions
Integrity and Security Issues	1. Enhances, for syntax and language rules that forbid or guard against violations. 2. Neutral. 3. Hinders, for a construct that facilitates the violation of integrity or access protections.	1. Package specifications and declarations 2. Object renaming declarations 3. Abort of a task
Reliability and Engineering Support	1. Enhances reliability (with explanation). 2. Neutral. 3. Problematic (with explanation).	1. Membership tests 2. Signed integer types 3. Derived types and classes
Robustness	1. Enhances (contributes to robustness). 2. Neutral (no effect on robustness). 3. Hinders (deleteriously affects robustness).	1. Predefined exceptions and language-defined checks 2. Concatenate operator 3. Task and entry attributes
Static Analysis	1. Tractable analysis. 2. Hard/Intractable analysis. 3. Unknown.	1. Enumeration representation clauses 2. Generic instantiation–subprograms 3. Unchecked type Conversions
Dynamic Analysis	1. Tractable analysis. 2. Hard/Intractable analysis. 3. Unknown.	1. Static expressions and subtypes 2. Dispatching subprograms 3. Generic formal objects

Figure 2.1. Ten analytical categories and ratings of the Ada95 Trustworthiness Study [Craigen1995]

• [ISO/IEC DTR 15942] authoritatively assesses all the language features of Ada95 based on verification techniques that are required by various standards and guidance. Such verification techniques are grouped as in figure 2 below. A rating is given for each of the language features to state whether a particular verification technique is directly applicable (Included), not straightforward but achievable (Allowed), or there is no current cost effective way (Excluded).

Approach	**Group Name**	**Technique**
Analysis	Flow Analysis (FA)	Control Flow
		Data Flow
		Information Flow
	Symbolic Analysis (SA)	Symbolic Execution
		Formal Code Verification
	Range Checking (RC)	Range Checking
	Stack Usage (SU)	Stack Usage
	Timing Analysis (TA)	Timing Analysis
	Other Memory Usage (OMU)	Other Memory Usage
	Object Code Analysis (OCA)	Object Code Analysis
Testing	Requirements-based Testing (RT)	Equivalence Class
		Boundary Value
	Structure-based Testing (ST)	Statement Coverage
		Branch Coverage
		Modified Condition/Decision Coverage

Figure 2.2. Verification Techniques employed in the assessment of Ada95 [ISO/IEC DTR 15942]

It also suggests that any language that may be used in implementing high integrity systems should

- be strongly typed,
- support a range of static types,
- have a consistent semantics that is defined in an international standard,
- support abstractions and information hiding, and
- have available validated compilers.

• The U.S. Nuclear Regulatory Commission (NRC) has produced a detailed study on the use of high level programming languages in high integrity and safety critical systems. The document [NUREG/CR-6463], entitled “Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems”, contains a framework of generic attributes significant to software safety that were gathered from many standards and research literature, and language specific guidelines derived from the framework for nine programming languages, i.e. Ada83, Ada95, C/C++, IEC 1131-3 Ladder Logic, IEC 1131 Sequential Function Charts, IEC 1131 Structured Text, IEC 1131 Function Block Diagrams, Pascal, and PL/M.

As listed below, four top-level attributes that define general quality of software were first identified, and appropriate intermediate and specific base attributes were developed.

Top-level attributes	Intermediate attributes	Base attributes
Reliability	1.Predictability of memory utilisation	· Minimising dynamic memory allocation · Minimising memory paging and swapping
	2.Predictability of control flow	· Maximising structure · Minimising control flow complexity · Initialising variables before use · Single entry and exit points for subprograms · Minimising interface ambiguities · Use of data typing · Accounting for precision and accuracy · Order or precedence of arithmetic, logical, and functional operators · Avoiding functions or procedures with side effects · Separating assignment from evaluation · Proper handling of program instrumentation · Controlling class library size · Minimising use of dynamic binding · Controlling operator overloading
	3.Predictability of timing	· Minimising the use of tasking · Minimising the use of interrupt driven processing
Robustness	1.Controlling use of diversity	· Controlling internal diversity · Controlling external diversity
	2.Controlling use of exception handling	· Handling of exceptions locally · Preserving external control flow · Handling of exceptions uniformly
	3.Checking input and output	· Input data checking · Output data checking
Traceability	1.Readability	See Maintainability
	2.Controlling use of built-in functions	None
	3.Controlling use of compiled libraries	None

Maintainability	1.Readability	· Conforming to indentation guidelines · Using descriptive identifier names · Commenting and internal documentation · Limiting subprogram size · Minimising mixed language programming · Minimising obscure or subtle programming constructs · Minimising dispersion or related elements · Minimising use of literals
	2.Data abstraction	· Minimising the use of global variables · Minimising the complexity of the interface defining allowable operations
	3.Functional cohesiveness	· Single purpose function and procedures · Single purpose variables
	4.Malleability	· Isolation of alterable functions
	5.Portability	· Minimising the use of built-in functions · Minimising the use of compiled libraries · Minimising dynamic binding · Minimising tasking · Minimising asynchronous constructs (interrupts) · Isolation of non-standard constructs

Figure 2.3. Generic Safe Programming Attributes [NUREG/CR-6463]

• The series of reports produced by the Motor Industry Software Reliability Association or MISRA [MISRA1994, MISRA1995a-h] cover virtually all areas of software development in motor industry. The major areas include project planning, assigning integrity levels, programming languages, verification, and quality assurance. Of our particular interest here is the selection criteria of programming language specifically stated in [MISRA1995f] and [MISRA1994], and the following are some of the important requirements.

- formally defined syntax and semantics
- a formal means of relating the code to the formal design

- block structured
- strongly typed
- run-time type and array bound checking
- conformance to an international standard
- use of a validated compiler
- well-understood
- exception handling
- extensive tool support, and tools that are trusted/validated.

The reports also contain some guidelines derived from relevant literature, such as [Cullyer1991] and [Carré1990], and these largely overlap other requirements described previously. The use of safer subsets is also emphasised.

2.2. High Integrity Programming Languages and Subsets

Although many different languages are used in high integrity systems, none of them may satisfy all the requirements reviewed thus far. Burns et al [Burns1998] suggest that a restricted programming model or profile can help produce efficient and predictable systems by removing language features with high overheads, and complex and erroneous semantics. Along these lines, there have been a few subsets or profiles suggested in the literature. This section reviews some of the popular ones including Ada, C/C++, Java, and their subsets or profiles.

There is an on-going argument as to which language is best suited for implementing high integrity systems. Developers consider Ada, C/C++, and Assembly among many, although none of the languages seems highly recommended in their full shape in a higher integrity levels (e.g., SIL 3 or 4). In fact, some argue that the choice of language *per se* does not directly contribute towards problems found in such systems; generally more problems arise from bad requirements analysis and design [Wichmann, http://www.cs.york.ac.uk/hise/sclist/cplussafety.html]. What matters, however, is whether there is a wide supporting community and expertise. The more successful a language is, the more likely that there are more experts, proven tools, and knowledge bases. Java has an enormous advantage in this area.

2.2.1. Ada 83 and 95[5]

Initiated by the U.S.'s Department of Defence, Ada has grown into a well-established language for the past twenty years in sectors like avionics, defence, and transportation. Its key features (of Ada95) include strong typing, exception handling, generics, multi-tasking, and support for object-oriented programming. Traditionally, Ada was developed with high integrity systems in mind. There are, nevertheless, some features that are difficult to analyse and verify, so that there have been efforts to develop a subset of the full language. Of notable is the Ravenscar profile by [Burns1998], which was standardised by the International Organisation for Standardization. In fact, this profile has been one of the key motivations for the idea presented in this thesis, i.e., the Ravenscar-Java profile.

The SPARK Ada [Barns2003] is also an attempt to restrict Ada in many ways, such that formal verification of programs is possible with attached annotations (mainly for pre- and post-conditions) and its tools. Now, the Ravenscar profile is being incorporated in the SPARK [Barns2003].

[5] Ada has recently undergone a further revision. The new language is called Ada 2005 and provides (amongst other things) better support for OOP (via Java-like interfaces) and advanced support for real-time systems development (for example, execution time monitoring).

2.2.2. C/C++

C/C++ [Kernighan1988, Stroustrup1997] has been the language of choice in many embedded systems thanks to supports from OS and tool vendors. But, its popularity is not without concerns; the language is weakly typed, programming errors can be passed unchecked, not many compilers implement the language correctly, and has no language level support for concurrency and exceptions. Especially, the pointer operators are complex to learn and use correctly, and the assignment operator can be mistakenly used instead of '= =' (logical comparison) in conditional statements, such as *if* and *while*. These simple syntactical errors are considered valid by the compiler, and the resulting code may have a completely different semantics.

Some areas of the language are also not well defined, so behaviours may vary from a compiler to another. The C standard lists 201 issues that may vary in this way [ISO/IEC 9899:1990, Annex G]. This becomes a serious problem when portability is an important issue.

There have been a number of attempts to make C more secure. Examples are MISRA C [MIRA1998], Safer C [Hatton1995]. These define programming guidelines and rules that must be enforced when programming.

2.2.3. Subsets of Java and RTSJ

Java is a strongly typed object-oriented language with ever-growing popularity among developers. When used in high integrity systems, a number of benefits will be realised due to the features that increase productivity and reduce run-time errors. Classes are extended and reused, and exceptions allow ways to deal with abnormal situations at run-time. It also supports concurrency and inter-thread communication mechanisms.

However, as mentioned previously, Java has been criticised for its unpredictable performance. Many of the major concerns come from the automatic garbage collector and dynamic class loading mechanisms. Time critical threads may have to be delayed due to the garbage collector, and accurate analysis of memory footprint is often extremely difficult. In addition, the real-time specifications add more complexity in analysis and verification, although they were intended to make the language more suited to general real-time domains.

The additional features ironically lead to programs and run-time that are very complex to implement and analyse. The run-time will also suffer from high overheads for checking the correct use of memory areas. This all prevents confident use of the full language, so that the following subsets or profiles have been proposed in literature.

2.2.3.1. Sequential subset of Java by [Bentley1999]

Bentley [Bentley1999] defines a subset of Java after assessing the language. The subset consists of 21 rules that are effectively derived from [Hutcheon1992], [MISRA1998] 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 is not allowed as it may cause significant difficulties in analysing programs, due mainly to the thread synchronisation mechanisms. In addition, methods and constructors shall not be overloaded.

• 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), so that no default values are expected. All constraints, such as, those used in *for*-loops, must be static. This will greatly ease various analyses, for example, memory requirement and timing analysis. 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. Plus, all switch statements should contain a final default 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. Because numbers beginning with zero are treated as octal values in Java, it is easy to make a mistake, e.g. inserting zero before a decimal constant.

• **Rules Concerned With Identifiers**

All identifier names must be unique.

• **Rules Concerned With Operators**

All right-hand operands of the logical operator && and || shall not contain any side effects, since the evaluation and execution of the operands are dependent on the truth-value of the left-hand operand. What is more, assignment operators must not be used in expressions which return Boolean values, for example, in *if ((x=1) != y)*. Bitwise operations, including bitwise shifts, shall not be performed on signed integer types, and the evaluation of integer expressions should not lead to wrap-around.

While 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.

2.2.3.2. Profile for high integrity Real-Time Java programs [Puschner2001a]

Puschner and Wellings [Puschner2001a] suggest a *Ravenscar*-like profile for the Real-Time Specification for Java [Bollella2000a], 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 the key mechanism for mutual exclusion to shared resources in Java, 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 due to its long-debated unpredictability at run-time. Instead, only immortal memory and linear-time scoped memory are supported as defined in the RTSJ. Immortal memory is used by default to create objects during the initialisation phase, but is not allowed for further object creation afterwards. In addition to this, 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

[Bentley1999]) and object-orientation features that can be problematic in performing various static analyse.

This profile, especially the computational model, served as a major input to our proposed profile in this thesis. Yet, our approach differs in that it is based on our assessment and criteria, which are derived from influential standards and guidelines. Java specific rules are developed and rationales are given, and it is consistent with the current release of Java and RTSJ. A number of extensions are also introduced.

2.2.3.3. 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 [JConsortium2000]. The profile has not been released yet, but according to Dobbing [Dobbing2001] it will resemble the Ravenscar profile for Ada95 [Burns1998]. 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 developed from the necessity to isolate critical code and data from non-critical ones by means of firewall, so that less-trusted code will never be able to interfere with high integrity programs. No exchange of objects, as well as dynamic loading across the firewall will be allowed. 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 is not supported, nor is any memory compaction mechanism. The use of general heap memory is also not allowed. 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. In addition to these, 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, so that there is no requirement for any implicit *join* interface.

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 as they add considerable complexity to program analyses. Further, all the asynchronous thread-to-thread operations, including *stop(), setPriority(), suspend(), resume()*, and event-driven Asynchronous Transfer of Control (ATC) mechanisms, are not permitted, nor are synchronised objects and counting semaphores.

• 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 [Puschner2001a], this profile is mainly focused on sub-setting the Real-Time Core Extensions [JConsortium2000], but does not address issues on the use of problematic language constructs and object-orientation features of Java.

2.2.3.4. Formal subsets by [Drossopoulou1999]

Drossopoulou *et al.* 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 [Drossopoulou1999], as shown below.

```
Program       ::=  Def*
Def           ::=  class ClassId ext ClassName impl InterfName*
                   {ClassMember*}
              |    interface InterfId ext InterfName* {InterfMember*}
ClassMember   ::=  Field | Method
InterfMember  ::=  MethHeader
Field         ::=  VarType VarId ;
Method        ::=  MethHeader MethBody
MethHeader    ::=  (void | VarType) MethId ((VarType ParId)*) throws ClassName*
MethBody      ::=  {Stmts [return Expr]] }
Stmts         ::=  (Stmt ;)*
Stmt          ::=  if Expr then Stmts else Stmts
              |    Var = Expr | Expr.MethName(Expr*) | throw Expr
              |    try Stmts (catch ClassName Id Stmts)* finally Stmts
              |    try Stmts (catch ClassName Id Stmts)+
Expr          ::=  Value | Var | Expr.MethName(Expr*)
              |    new ClassName() | new SimpleType([Expr])+ ([])* | this
Var           ::=  Name | Expr.VarName | Expr[Expr]
Value         ::=  PrimValue | RefValue
RefValue      ::=  null
PrimValue     ::=  intValue  | charValue | boolValue | ...
VarType       ::=  SimpleType | ArrayType
SimpleType    ::=  PrimType | ClassName | InterfaceName
ArrayType     ::=  SimpleType[] | ArrayType[]
PrimType      ::=  bool | char | int | ...
```

Figure 2.4. Java^s programs [Drossopoulou1999]

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. [Hartel2001] surveys formal subsets and approaches aimed at improving the safety of Java programs.

2.3. Program Analysis

Although not directly a language issue, program analysis plays a paramount role in the development of high integrity systems, in which the cost of malfunction is often unaffordable. We view that there are two areas of analysis; i.e., program safety analysis, and timing analysis. In the former, efforts are made to find functional anomalies while, in the latter, timing constraints are checked and schedulability is tested.

2.3.1. Program Safety Analysis

Most traditional analysis algorithms and tools were focused on finding simple errors that could halt or damage a given software system. These tools, such as *LCLint* [Evans1994], typically locate mismatches of types and non-initialised variables, and monitor the scope of variables and simple pre- or post-conditions in order to make certain that all sequential operations are robust.

However, these tools alone may not be of much help in analysing modern software since recent programming languages are heavily equipped with a number of novel features, such as object-orientation, concurrency and synchronisation mechanisms. This fact adds a great deal of complexity to program analysis. Hence, any mechanism that attempts to find errors in Java software needs to provide additional facilities to accommodate such language features.

There has been a range of formal mechanisms designed with such systems in mind, in which multifarious concurrent threads may be explored either statically or dynamically with the aim of searching for any data races, deadlocks and other miscellaneous errors. Examples include the Extended Static Checker (ESC) for Java

[Flanagan2002], which employs a theorem prover and several annotations, the Eraser [Savage1997], which checks whether a multi-threaded program complies with the mutual-exclusion locking discipline, and the ExitBlock [Bruening1999, Bruening2000], which is guaranteed to find deadlocks and race conditions if present. However, some complex properties of Java programs cannot be checked due to the language's expressive power being too good. Recently, model checking is proving to be successful in verifying requirements, designs and even program code of Java. The next section deals with this.

2.3.2. Model Checking

Program analysis techniques have evolved over the past few decades. Model checking is a new area of research that deserves an attention from the high integrity/safety-critical community. The fundamental idea behind model-checking is building formal models of a target system in a given description language at design phases, and verifying that the models are sound in terms of safety and liveness, by means of exploring the whole finite state space of the models using a model checker. As a result of this, the existence of deadlocks, data races, or other concurrency errors may be determined and reported. Depending on applications, users may also check other specific properties by adding pre- or post-conditions, invariants, or some formulas in a supported logic description language, such as Linear Temporal Logic, into the model.

It is a generally held belief that building a model and checking it is a good way of gaining confidence early in design stages. In many cases, however, a gap is introduced between a model and its implementation; in other words, it is often difficult to implement a system that exactly matches its original model because of additional details and complexity incurred at lower levels. Therefore, even if we can prove that our model is free from any error, there may not be any way of proving that our implementation will also be free of errors unless certain formal translation techniques are developed.

Another obvious challenge is that as the size of a model increases its state space grows too, often exponentially. This fact was one of the major obstacles that prevented utilising model-checkers. Yet, on account of the improvements in this area over the recent years, modern model-checkers require less state space. The following sections briefly introduce some issues on conventional model-checkers, as well as some evolving ones that mainly attempt to reduce the gap mentioned above.

2.3.2.1. Conventional Model Checkers

As described above, conventional model checkers, such as SPIN [Holzmann2000, Holzmann1997, Holzmann2002] and UPPAAL [Larsen1997a-b, Larsen2001, Pettersson1999], normally require users to build abstract models of a system in full or in part. The models are then checked and verified by a model checker, which will either report errors or claim that the instances of the safety and liveness properties (i.e. user-specified assertions, and absence of deadlocks and race conditions in the model) are satisfied.

Amongst those popular model checkers, the SPIN model checker has successfully been promoted to interface with Java programs via some translation mechanisms; other model-checkers are not directly applicable to Java, but may still be worthy as a means of proving the design of Java programs. Within those translation mechanisms, the source code of a Java program is typically translated into PROMELA, the input specification language of SPIN. For example, see [Bandera2005, Corbett2000, Havelund1999a-c, Demartini1998, Stoller2000]. They are mostly aimed at finding race conditions, deadlocks and user-specified assertions.

Those automatic translation tools or model-extractors, however, normally provide only limited support for the advanced Java language features. For instance, dynamic memory allocation, exception handling, or polymorphism are not dealt with in most of the tools, nor are floating point numbers handled. Another challenge stems from the unavailability of the source code of libraries used, so that some assumptions are necessarily made. Moreover, these tools themselves are not sometimes rigidly constructed or tested, so that they can be disqualified for use in the development of high integrity systems.

2.3.2.2. Innovative Model Checkers

Considering the challenges found in most model-extractors and model-checking in general, it may be a good idea to perform model-checking on implementation or compiled programs themselves. This idea has been investigated for Java in [Brat2000a, Brat2000b, JPF2005], where they have developed the second generation of Java PathFinder.

The Java PathFinder 2 or JPF2 is a sophisticated model-checker in its own right. It contains a custom-made Java virtual machine, the MC-JVM, and depth-first-searching mechanism that all run on top of other standard Java virtual machines [Brat2000a], so that it is readily portable, like the Rivet virtual machine in the previous section [Bruening1999, 2000]. It takes Java bytecode as input, and then carries out model-checking on Java programs without having to worry about most of the problems found in model-extraction.

2.3.2.3. Mechanisms integrated

In order to reduce state-space, the model-checker employs various techniques in the literature and of its own. It is possible before model-checking for the programmer to abstract away concrete variables that may unnecessarily lead to an infinite number of states by specifying abstraction criteria in the source code. This is done by calling special library functions provided by the model-checker, such as *Abstract.remove()*, in appropriate classes and methods. In place of such variables, especially in conditions of if and while statements, Boolean abstraction predicates can be added, for example, by invoking *Abstract.addBoolean("EQ", intCount == x)* where *intCount* and *x* are both integer variables used in the condition of a while statement, and the whole expression will be replaced by EQ that will either evaluate true or false when model-checking is performed [Brat2000a]. An automated abstraction tool to assist this task is under development.

Moreover, JPF uses the slicing tool of the Bandera toolset [Bandera2005, Corbett2000] that statically identifies and removes irrelevant statements in Java programs to construct specialized minimum models for a particular specification of properties to be verified. The tool first analyses the given specification and produces a slicing criterion or criteria, which are then used to perform dependency analyses on the program. After that, a slice of the original program that only contains necessary statements for checking the specification is produced, again resulting in fewer states to explore when model-checked by JPF. However, the effectiveness of the slicing technique heavily relies on the organisation or structure of the program. That is to say, if the components of the program are tightly coupled, then the resulting slice may not be dramatically small.

Within the static analysis phase, safe program blocks are also identified along with the slicing procedure. By safe blocks, we mean a sequence of Java program statements that can safely be executed in parallel without being affected by interleavings of other threads. This identifying of safe blocks may be of great help in partial order reduction during actual model checking [Brat2000a].

Furthermore, JPF performs run-time analysis that assists model checking; Eraser and LockTree algorithms are implemented. Eraser [Savage1997] is an efficient algorithm for finding race conditions as explained before in 4.1.2, whereas the LockTree algorithm developed by the JPF team is a means of detecting potential deadlocks by looking into orders of locks [Brat2000b].

2.3.3. Timing and Schedulability Analysis

Many high integrity systems have hard real-time requirements. In order words, any missed deadlines are intolerable, and can cause a system-wide failure. Schedulability analysis takes timing properties of all threads/tasks of an application, blocking times, jitters into account, in order to determine whether the application is schedulable. Fixed priority assignment schemes are common, and well studied in the literature, e.g., [Burns2001, Audsley1993], while dynamic ones are being investigated mainly for soft real-time systems.

For schedulability analysis to be performed, the worst-case execution time (WCET) and deadline of each individual thread are essential. Based on that, priorities are assigned according to the priority allocation scheme used. Blocking times, which are caused by resources shared by more than one thread, should also be considered as well as any system-level jitters. Tasks or threads can be pre-empted by a higher priority one, so that the time caused by such interferences must also be taken into account.

2.4. Summary

This chapter has surveyed subjects relevant to the theme of the thesis, mainly, requirements on high integrity programming languages, a few languages and subsets with Java being the main focus, and program analysis techniques. An emphasis was placed on reviewing the standards and guidelines for selecting the right language, and the current status of Java and related subsets.

Undoubtedly, subsets of a complex language will make analysis less complex. By-product of this is the simplification of the analysis techniques per se, such that more confidence can be gained in the development of such tools. For instance, a model checker for a subset of Java, if developed, will not have to deal with all of the Java's complex features (especially the object queue and unconstrained synchronizations), so that state space required will be much smaller. When there are no synchronizations between threads at all, the model checker would only need to consider one thread at a time, reducing a great deal of states.

Based on the review here, the next chapter will set up a framework of criteria for selecting high integrity programming languages. This framework will then be used to assess Java and the Real-Time Specification for Java (RTSJ).

Chapter 3. Assessment of Java and RTSJ

Having reviewed important guidelines and standards, we now set up a framework of criteria in order to assess Java and RTSJ. By applying this framework, strengths and weaknesses will be revealed, such that they will help develop a high integrity profile of Java and RTSJ. In other words, we find out the areas that Java is strong as well as deficient.

The framework may be of use for other languages. It was developed by going through each of the standards and guidelines, and categorising them into two levels, i.e., mandatory and desirable (or advisory) requirements. Appropriate ratings are also given for the mandatory requirements, while desirable ones are not rated since they are advisory requirements and may be beneficial in that they help produce mor e efficient, comprehensible, and structured systems.

3.1. Assessment Criteria

Since some of the requirements introduced in the previous chapter are redundant and ambiguous, we inclusively categorise them into related assessment criteria along with appropriate references. However, it is important to note that this collection of criteria is only concerned with relevant requirements and guidelines[1], and it attempts to amalgamate many different requirements into a framework for the assessment of programming languages. As in [Hutcheon1992] we propose two levels of criteria, namely *Mandatory requirements* (Level 1) and *Desirable requirements* (Level 2).

3.1.1. Level 1 – Mandatory requirements

In Level 1 we identify as many mandatory requirements that a language must satisfy as possible in order to be considered for use in implementing high integrity systems. Appropriate justifications are made regarding each requirement. Readers are encouraged to refer to the references if in any doubt about rationales and specifics.

L1.1. Syntactical / Semantic Requirements

L.1.1.1	**Type safety / Strong typing rules**
References	[USDoD1978], [Cullyer1991], [USDoD1990], [Hutcheon1992], [Craigen1995], [ISO/IEC DTR 15942], [NUREG/CR-6463], [MISRA1995f]
Rationale	Strongly typed languages help reduce errors in programs at compile-time. Moreover, type safety is often considered to be sufficient for ensuring the minimum nontrivial level of program safety, i.e. control flow safety, memory safety, and stack safety [Kozen1999]. Thus it is imperative to use a type safe or strongly typed language, enhancing the integrity and security of software.
Specifics	Implicit type conversions must not be allowed. All data types should be statically analysable before program execution. Explicit type conversion rules should be clearly stated in the language standard or definition. There should be some ways to avoid access types or pointers that can cause dangling references.
Ratings	1. Strongly typed / statically analysable. 2. Strongly typed, but some types are analysable only at run-time, mainly due to the use of polymorphism in the language. 3. Not strongly typed and implicit type conversions are allowed.

[1] Some requirements or guidelines are deliberately missed out because they are either not relevant with respect to high integrity systems, or considered not reasonable in the context of modern programming languages. Examples include requirements on the use of a particular character set [USDoD1978], and improvements in wording or program presentation (of Ada83) [USDoD1990].

L.1.1.2	**Side effects in expressions / Operator precedence levels / Initial values**
References	[USDoD1978], [NUREG/CR-6463]
Rationale	Side effects in expressions can cause programs to behave in an ambiguous, or, possibly, unpredictable way, thus are not desirable. The precedence levels of all operators must be specified in the language definition; otherwise evaluation orders may vary from system to system.
Specifics	There should not be any time-dependent side effects in expressions. Operator precedence levels must clearly be defined in the standard. There should be no implicit initial values for variables.
Ratings	1. All the above specifics are satisfied. 2. Not all the above specifics are satisfied, but there may be a subset of the language that meets the specifics. 3. The above specifics are not satisfied, and there is no reasonable way to improve the language.

L.1.1.3	**Modularity / Structures**
References	[USDoD1978], [Cullyer1991], [Hutcheon1992], [Craigen1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	It must be straightforward to code and maintain programs in a high integrity programming language, so that the complexity of software becomes manageable. This is often achieved by means of visibility control (or scopes), functions, and objects in many modern languages, in which the integrity and security of software are generally improved.
Specifics	There should be sound mechanisms to structure and modularise program code both syntactically (in some form of determinable blocks or scopes) and semantically with clear interfaces. There should be no wild/unbounded jumps between different modules. Separate compilation of modules should be possible.
Ratings	1. The language provides rich and precise means of structuring programs, and programs can be maintained in terms of modules or objects. 2. Such mechanisms are provided, but not cost-effective or efficient. 3. There is no reasonable approach.

L.1.1.4	**Formal semantics / International standards**
References	[USDoD1978], [Cullyer1991], [USDoD1990], [Hutcheon1992], [Craigen1995], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	A standardised language benefits the development of compilers and tools, and user training. Verification techniques can also be applied to a language with formally defined semantics.
Specifics	There should be a (international) standard definition of the language. There should be formally defined semantics of the language, or at least a subset of the language.
Ratings	1. An internationally standardised formal definition exists. 2. The language or high integrity subset of it can be formally defined. 3. Unknown.

L.1.1.5	**Well-understood**
References	[Cullyer1991], [USDoD1978], [Hutcheon1992], [USDoD1990], [MISRA1995f]
Rationale	A language with well-understood semantics and syntaxes will help to produce quality software, often cost-effectively. Subsets without complex features will also be easier to learn and be used to develop more reliable software.
Specifics	The language should be simple, well understood, easy to adopt, and easy to implement. At least, a better-understood subset must be developed.
Ratings	1. The language (or the subset) is well understood, and there are many trained developers and designers. 2. The language (or the subset) is well understood only by a limited number of people. 3. Unknown.

L.1.1.6	**Support for domain specific or embedded applications**
References	[USDoD1978], [Hutcheon1992]
Rationale	High integrity systems are often embedded systems that need to interface or control physical resources or (non-standard) peripheral devices. Therefore, a programming language designed with such applications in mind should be used.
Specifics	Robust mechanisms for controlling memory, I/O devices or other hardware are required.
Ratings	1. The language naturally supports embedded applications. 2. There is a limited support, but external libraries or language extensions can be utilised. 3. No support provided or Unknown.

L.1.1.7	**Concurrency / Parallel processing**
References	[USDoD1978], [Hutcheon1992]
Rationale	Although concurrency is one of the main sources of complication in program analysis and verification (classified as only a desirable, not mandatory feature in [Hutcheon1992]), it is invaluable in modelling or capturing real-world problems. Thus, we believe this has to be an essential requirement for modern high integrity language. However, this must not result in a complex and difficult programming model to analyse in terms of schedulability. A subset of a complex language may be considered.
Specifics	The following features must be included: Language-level support for multitasking or multithreading. Control over scheduling policy. Straightforward communication and synchronisation mechanism(s), plus facility to bound blocking.
Ratings	1. All the above specifics are satisfied. 2. Only limited support is provided at the language-level, but external libraries or run-time systems can be utilised. 3. No reasonable support provided or Unknown.

<u>L1.2. Application of verification techniques / Predictability</u>

L.1.2.1	**Functional predictability**
References	[Hutcheon1992], [Craigen1995], [ISO/IEC DTR 15942], [NUREG/CR-6463], [MISRA1995f]
Rationale	High integrity software must be proven to be predictable in terms of its functional behaviours.
Specifics	All or most of the following analysis techniques should be applicable. Control flow analysis Data flow analysis Information flow analysis Symbolic execution Formal code verification
Ratings	1. All techniques in the above specifics or feasible alternatives can be utilised. 2. Not all techniques can be utilised due to the complex features of the language, but sub-setting the language may improve such analyses. 3. Unknown or there is no cost-effective way of utilising such analysis techniques.

L.1.2.2	**Temporal predictability / Timing analysis**
References	[Hutcheon1992], [Craigen1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	In addition to the functional predictability, timely behaviours of such software and systems must also be guaranteed.
Specifics	Worst Case Execution Time (WCET) of each process must be obtainable, so that schedulibility analysis can be performed.
Ratings	1. Tightly bounded execution time(s) can be obtained. 2. Loosely bounded execution time(s) can be obtained. 3. Unpredictable or there is no known way to obtain WCET.

L.1.2.3	**Resource usage analysis**
References	[Cullyer1991], [Hutcheon1992], [Craigen1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	It is important to identify what resources are needed and how they are utilised, so that errors such as stack overflow may not occur, and system implementations may be kept economical.
Specifics	The following properties should be analysable. Memory (or heap) usage Stack usage Any other resources to be utilised in the application area.
Ratings	1. Exact prediction of the above specifics is possible. 2. Worst-case analysis is possible, but not practical. 3. Unpredictable.

<u>L1.3. Language Processors / Run-time environment / Tools</u>

L.1.3.1	**Certified language translators / Run-time environments**
References	[USDoD1978], [Hutcheon1992], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	There must be a high level of assurance in language processors, especially compilers.
Specifics	A formally certified compiler by an authoritative or trusted body should be used. Low-level code should be traceable in accordance with source code. Run-time environments should also be certified if used.
Ratings	1. There exist one or more certified language translators, and they are formally proven to be flawless. 2. Language translators may contain several known errors or malfunctions that are well documented, but they will not affect the development of high integrity software. 3. Unknown.

L.1.3.2	**Run-time support / Environment issues**
References	[USDoD1978], [Hutcheon1992], [USDoD1990], [Craigen1995], [ISO/IEC DTR 15942], [NUREG/CR-6463]
Rationale	Libraries (or any additional code) or run-time support may make it complex to perform some analyses, such as WCET and control flow analyses. Hence, all such additional code should be predictable and analysable in terms of safety and timeliness. Minimising implementation dependencies is also encouraged.
Specifics	All the behaviours of additional code should be well understood. All timing information of the underlying run-time system and libraries should be known and accurate.
Ratings	1. There exists concrete information on the functional and temporal behaviours of all libraries and run-time system. 2. Only worst-case analysis is possible. 3. Unknown.

3.1.2. Level 2 – Desirable Requirements

The requirements at this level are not immediately necessary but beneficial in that they help produce more efficient, comprehensible, and structured systems. Note that ratings are not provided at this level because they are meant to be advisory.

L2.1. Syntactical / Semantic Requirements

L.2.1.1	**Exception Handling / Failure behaviour**
References	[USDoD1978], [Cullyer1991], [Hutcheon1992], [Craigen1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	Handling errors while a high integrity system is operating is sometimes seen as undesirable on account of additional overheads and unpredictable behaviours. However, if any sort of error can occur, then the system should gracefully degrade, or recover after some corrections.
Specifics	Robust and analysable run-time error detection and handling mechanism should exist. Failure behaviours should be programmable.

L.2.1.2	**Model of Mathematics**
References	[Cullyer1991], [USDoD1978]
Rationale	As often required in some high integrity systems, the language should have a rigorous model of maths defined in the language standard.
Specifics	A model of both integer and floating point arithmetic should be defined within the language standard. Procedures for checking if operational arithmetic at run-time is correct should exist.

L.2.1.3	**Support for User documentation**
References	[USDoD1978], [USDoD1990], [NUREG/CR-6463]
Rationale	Languages that allow user comments will undoubtedly improve program readability and maintainability. Some language processors may make use of annotations to detect subtle logical errors in programs or to obtain extra information.
Specifics	There should be some way of commenting programmer's intentions within source code.

L.2.1.4	**Support for a range of static types including subtypes and enumeration types**
References	[USDoD1978], [USDoD1990], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	It is easier to perform any analyses or checks on static types than on dynamic types. Enumeration types with a limited number of values also help reduce errors.
Specifics	None.

L.2.1.5	**Coding style guidelines**
References	[Hutcheon1992], [NUREG/CR-6463], [MISRA1995f]
Rationale	Coding style guidelines may help reduce the gap between well-established Software Engineering principles and the actual practice of programming in a particular language.
Specifics	None.

L.2.1.6	**Support for abstraction and information hiding**
References	[ISO/IEC DTR 15942], [Hutcheon1992], [USDoD1990], [MISRA1995f]
Rationale	Employing abstraction or information hiding techniques (e.g. object orientation) can greatly decrease software complexity. Thus they are beneficial in program design, development, and maintenance.
Specifics	None.

L.2.1.7	**Assertion checking**
References	[USDoD1978]
Rationale	It may sometimes be desirable to check for user specified assertions before or while programs are executing.
Specifics	None.

L2.2. Language Processors / Run-time environment / Tools

L.2.2.1	**Certified (static/dynamic) analysis tools**
References	[Hutcheon1992], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	In order to gain more confidence in high integrity software it is imperative to use certified analysis tools, which may check for errors, such as, race conditions and deadlocks.
Specifics	None.

L.2.2.2	**Interface to other languages**
References	[USDoD1978]
Rationale	There are some situations where a program written in a high-level language needs to interact with existing libraries or other low-level routines that are written in different languages. In such cases there should be a means of interfacing our program with such routines.
Specifics	None.

L.2.2.3	**Code optimisation**
References	[USDoD1978]
Rationale	It is always advantageous to improve the efficiency of programs by means of optimising them. However, optimisation should not alter the semantics of correct programs, nor compromise the application of analysis techniques.
Specifics	None.

L.2.2.4	**Code portability**
References	[Hutcheon1992], [NUREG/CR-6463]
Rationale	Since there exists a diverse range of code-executing platforms, it is often considered beneficial to have a portable program representation, so that all necessary analyses may be applied once for all.
Specifics	None.

3.2. Assessment of Java and RTSJ

Based on the framework of criteria above, Java[2] is assessed in this section. Note that each criterion is numbered in the same way as appears in the framework. A summary of the assessment is provided at the end.

3.2.1. Assessment of Java against Level 1

L.1.1. Syntactical / Semantic Requirements

L.1.1.1. Type safety / Strong typing rules

Java is a strongly typed language. For all primitive types, implicit type conversions are not allowed (all possible conversions are stated in the language specification), and

[2] This assessment is consistent with the current release of Java, i.e., 1.5.

programs are analysable before running them. Yet, for dynamic reference types, it is not always straightforward to statically analyse code, but is generally possible only at run-time because of the use of, for example, inherited interfaces and local classes within different scopes.

Rating: 2. Strongly typed, but some types are analysable only at run-time, mainly due to the use of polymorphism in the language.

L.1.1.2. Side effects in expressions / Operator precedence levels / Initial values

Side effects can occur in Java if expressions contain embedded assignments, sub-operators, and method invocations. Many side effects, however, can be eliminated via the use of a code checker or analyser, and a subset of Java. Operator precedence levels are defined in the specification [Gosling2000], but the large number is at times seen undesirable as it becomes difficult for programmers to learn [Bentley1999, USDoD1978]. All types in Java have default initial values, but compilers issue warnings if any variables are used before initialisation. It should also be noted that some returned values of a method could be *quietly discarded* without any warning [Gosling2000], i.e., when there is no assignment expression for a method call that returns a value. Moreover, assignment expressions can be mistakenly used in the place of logical comparisons without any warning. See the following for an example.

```
public void caller(String args[])
{   …
   if (a = b) {  // Must be ==
      callee(); // Return ignored
   }
}
```

```
public anObject callee() {
    …
    return new anObject();
}
```

Rating: 2. Not all the above specifics are satisfied, but there may be a subset of the language that meets the specifics.

L.1.1.3. Modularity / Structures

In Java, programs are organised as objects that normally consists of visible and non-visible data fields and methods. Abstraction and encapsulation mechanisms are also provided through classes and interfaces, and packages (into which related classes are organised) also enhance modularity and structure of software. In addition to this, the language contains various means of controlling program flows, including the exception-handling mechanism. Separate compilation is always possible.

Rating: 1. The language provides rich and precise means of structuring programs, and programs can be maintained in terms of modules or objects.

L.1.1.4. Formal semantics / International standards

There are no stable standards for Java although the language specification [Gosling2000, JSR176] serves as an informal standard for the time being. There exist some formal semantics of Java (various versions), for example, in Action semantics [Watt2000, Brown1999], in Denotational Semantics [Alves-Foss1999b], and in other BNF-like notations [Alves-Foss1999a]; most of which are based on only parts of the language. Drossopoulou and Eisenbach [Drossopoulou1999] have also defined a series of subsets of Java and proved their type soundness.

Rating: 2. The language or a high integrity subset of it can be formally defined.

L.1.1.5. Well-understood

Java is a familiar programming language to many existing C/C++ programmers, which means that no extensive training is usually required and there may well be many trained engineers. In addition, some of the problematic features in C/C++ (such as pointer operations) are removed, which all results in a dramatic increase in productivity. However, the excessive number of APIs and other additional mechanisms can be hard to master.

Rating: 1. The language is well understood, and there are many trained developers and designers.

L.1.1.6. Support for domain specific or embedded applications

One of the main application areas for which Java was first developed was embedded systems. In pure Java, however, it is not possible to control underlying hardware without appropriate native methods implemented in different languages. Even then, it is still difficult to implement systems with rigorous safety and real-time requirements, thanks mostly to the overheads incurred by the garbage collection mechanism, and virtual machines *per se*. There has been much research on scheduling the garbage collector and improving the efficiency of code transformation, even though it has not proven particularly effective so far. In the recent years, the Real-Time Specification for Java [Bollella2000a] and Real-Time Core Extensions [Jconsortium2000] have been defined, so that real-time applications will certainly benefit from reference implementations of such specifications.

Rating: 2. There is a limited support, but external libraries or language extensions can be utilised.

L.1.1.7. Concurrency / Parallel processing

Java supports concurrent execution of multiple threads, as well as some key synchronisation mechanisms, for example, the monitor and synchronized blocks/methods. Programmers can also allocate a priority to threads, which nevertheless is not of any significant value, as they have no control over scheduling mechanisms implemented in the virtual machine and underlying kernel. Recently, two of the specifications for real-time Java, i.e., one from Sun Microsystems [Bollella2000a] and the other J Consortium [JConsortium2000], state various features that real-time systems require, especially with regard to scheduling, memory management, synchronisation, time, and exceptions.

Rating: 2. Only limited support is provided at the language-level, but external libraries or run-time systems can be utilised.

<u>L.1.2. Application of verification techniques / Predictability</u>

L.1.2.1. Functional predictability

Due to the recent development of sophisticated analysis algorithms and tools [Flanagan2002, Brat2000a, JPF2005, Bandera2005], it is now possible to some extent to analyse Java programs in terms of control and data flow. Nevertheless, some complex features of Java, such as the exception handing mechanism and monitors, are still not considered, or at least are immaturely handled. Formal verification is even harder for Java as there is no complete formal semantics. However, a constant progress is made in this area, and especially *Model-checking* technology is proving strong in the verification of Java programs. For example, the Java PathFinder 2 [JPF2005, Brat2000a] developed by the NASA can detect race conditions, deadlocks, and violations of user-specified assertions.

Rating: 2. Not all techniques can be utilised due to the complex features of the language, but sub-setting the language may improve such analyses.

L.1.2.2. Temporal predictability / Timing analysis

It is well known that with all the sometimes-superfluous features like the garbage collector and virtual machine support, it is hard to obtain tight execution-time bounds for Java threads, and such timing analyses are all dependent on eventual target architectures and base operating systems (if utilised). Some techniques, however, have been suggested (e.g. [Bate2000, Puschner2001b]), and the release of the specifications for real-time Java will certainly improve the current situation.

Rating: 2. Loosely bounded execution time(s) can be obtained.

L.1.2.3. Resource usage analysis

On account of the presence of the background garbage collector, it is generally difficult to predict how much memory space will be in use at a given moment in time, or even deducing the worst case can become impractical (and dependent on which garbage collection algorithms are employed). However, subsets of Java or of the Real-Time Specification for Java [Bollella2000a], such as [Puschner2001a] in which garbage collection is excluded, will ease this sort of analysis.

Rating: 2. Worst-case analysis is possible, but not practical.

<u>L1.3. Language Processors / Run-time environment / Tools</u>

L.1.3.1. Certified language translators / Run-time environments

To the best of our knowledge, Java compiler and virtual machine validation is still an on-going research work. Whereas it may never be possible to formally exploit and validate such complex software, some attempts have been made to conduct conformity assessment of Java or Java-like language processors to the language specification and industry standards, for example see [PERENNIAL2001]. Reported errors are reasonably well documented and updated.

Rating: 2. Language translators may contain several known errors or malfunctions that are well documented, but they will not affect the development of high integrity software.

L.1.3.2. Run-time support / Environment issues

It is not easy to perform analyses on additional code, i.e., that of variable run-time systems, APIs, native methods, unless a sound standard for such program entities is developed.

Rating: 3. Unknown.

3.2.2. Assessment of Java against Level 2

L.2.1. Syntactical / Semantic Requirements

L.2.1.1. Exception Handling / Failure behaviour

Java has a wide variety of predefined exception classes, and programmers are also allowed to define customised (checked) exceptions and program's behaviours. Uncaught exceptions, i.e., unchecked exceptions or errors, can become problematic as they may result in the system halting.

L.2.1.2. Model of Mathematics

Java provides a rich set of integer and floating point data types, and the `java.math` package can be used to assist in more rigorous mathematical applications. While the utilisation of the standard IEEE 754 arithmetic semantics is seen as universally beneficial in terms of compatibility, it is occasionally not desirable as it hinders the utilisation of advanced hardware, for example, built-in co-processors [Bentley1999].

Another problem is when two positive integers are added and overflow occurs, the result value and the sign can be different from the mathematical sum of the two values [Gosling2000]. There is no warning generated in such a situation, and it is completely up to the programmer to ensure that this does not happen.

L.2.1.3. Support for User documentation

Java provides two ways of commenting source code. Furthermore, there is a facility for automatically generating on-line documentation of user classes, i.e. *javadoc* tool.

L.2.1.4. Support for a range of static types including subtypes and enumeration types

Generic and enumeration types are supported in Java 1.5. They are statically bound at compilation time. There are a number of primitive types, which are used to construct object types. However, the concept of sub-typing is not supported directly.

L.2.1.5. Coding style guidelines

There exist coding style documents available at the WWW site of Sun Microsystems [Sun1999]. However, none of them specifically addresses high integrity or real-time applications.

L.2.1.6. Support for abstraction and information hiding

As an object oriented language, Java offers abstraction by means of the abstract class type and interface, where no implementation details are allowed. Information hiding is also naturally supported.

L.2.1.7. Assertion checking

The assertion facility was added in Java 1.4. Although assertions are only checked when the program runs on a virtual machine (disabled by default), they are useful in testing. Complex assertions can be created by means of invoking static methods from assertions.

L.2.2. Language Processors / Run-time environment / Tools

L.2.2.1. Certified (static/dynamic) analysis tools

A large number of analysis tools have been developed to assist in debugging Java programs, but most of them are not certified by reliable bodies or standards. However, as mentioned above, tools such as Java PathFinder 2 (from NASA) and the Extended Static Checker for Java (from Compaq) appear to be successful in detecting many known errors.

L.2.2.2. Interface to other languages

Java cannot directly interface to programs written in other languages. But, it is possible to invoke native methods, mostly written in C, of the run-time environment. This will result in poor portability.

L.2.2.3. Code optimisation

Most of the available optimisation techniques are not applied until Java programs reach their target or virtual machine for security reasons. Different quality of code or performance may be generated depending on how code is processed, i.e. bytecode can be interpreted, compiled *Just-in-Time*, or compiled *Ahead-of-Time*. It is complex to statically analyse optimised native code in relation to high-level bytecode. Optimisation will also make the complexity of compiler and tool validation more difficult.

L.2.2.4. Code portability

Following the *"write once and run everywhere"* motto, Java has become a truly portable programming language for most of the well-known platforms. In addition, Java chips with an integrated virtual machine and processor also start to appear. However, a problem can arise when non-standard processors or operating systems are utilised, where the burden of developing a new virtual machine is left to the system developer.

3.2.3. Summary of Assessment

Most of the Level 1 criteria are not, or loosely met by Java. Below is a summarising classification of the strengths and weaknesses identified above.

<u>*Strengths*</u>

> Java is a strongly typed object-oriented language that provides an excellent means of modularising and structuring programs (L.1.1.3), and is well understood (L.1.1.5). It also supports concurrent execution of multiple threads as well as some key synchronisation mechanisms (L.1.1.7). This is an

advantage in that Java is expressive enough to model non-trivial concurrent objects. However, the model can be a weakness at the same time due to difficulties in analysis (see below).

Weaknesses

Reference types are not generally amenable to static checking (L.1.1.1). Furthermore, side effects can occur in expressions, and some returned values may be quietly discarded (L.1.1.2). There exist several formal definitions and semantics of Java, but they are predominantly concerned with parts of the language (L.1.1.4). Embedded applications, in which hardware control is essential, can only be supported by means of native methods (L.1.1.6), although implementations of the specifications for Real-Time Java are expected to solve this problem.

It is not straightforward to apply various analysis techniques directly to Java due to some of its complex features (L.1.2.1), but there appear to be some evolving analysis tools. Timing analysis is also difficult to perform on Java code (L.1.2.2), as is resource usage analysis (L.1.2.3).

There is no formally validated Java compiler and virtual machine, but conformity-checking tools do exist (L.1.3.1). It is also complex to perform any analyses on additional code, such as that of APIs and run-time systems (L.1.3.2).

Regarding Level 2 requirements, the following strengths and weaknesses have been identified.

Strengths

- Integrated exception handling mechanism (L.2.1.1)
- Rich set of integer and floating-point data types, and java.math package (L.2.1.2)
- Support for user documentation (L.2.1.3)
- General coding style guidelines (L.2.1.5)
- Support for abstraction and information hiding (L.2.1.6)
- Code portability (L.2.2.4)

Weaknesses

- Overhead and complexity of exception handling mechanism (L.2.1.1)
- The utilisation of the standard IEEE 754 arithmetic semantics can be overhead, and no exception is generated for particular operations, e.g., overflow in integer additions (L.2.1.2)
- No coding guidelines for high integrity applications (L.2.1.5)
- Shortage of certified analysis tools (L.2.2.1)
- Difficulty in interfacing to programs written in other languages (L.2.2.2)
- Complexity of analysing optimised code (L.2.2.3)

3.3. Motivations for a high integrity profile

Those weaknesses identified above are the key areas that need improvements, where most of which can be tackled by subsetting the language. Moreover, it will be achievable to define a formal semantics of a subset, and analysis techniques as well as the target platform will be more efficient and reliable. In short, considering all the strengths, Java can become a major candidate for implementing future high integrity software.

3.4. Summary

We have reviewed important requirements of programming language for the development of high integrity software, and defined 23 assessment criteria derived from the requirements. The criteria are divided into two groups, namely, *Mandatory requirements* (Level 1) and *Desirable requirements* (Level 2). Appropriate references and rationale for each criterion are given, and suitable ratings are also provided for the Level 1 requirements.

The Java programming language and its associated environments are then assessed against the two levels of criteria, and we conclude that Java is a good general language, yet not appropriate as a whole for the development of high integrity systems that require rigorous and predictable language features, compilation systems, and tools. However, Java may be able to qualify as a suitable vehicle in the future with the help of sub-setting the language and future developments of formal mechanisms, although none of the currently proposed subsets address all the necessary areas required for high-integrity real-time systems. The next chapter presents our approach, the Ravenscar-Java profile.

Chapter 4. Ravenscar-Java Profile

Based on the assessment and guidelines presented in Chapter 3, a language profile is proposed in this chapter, which is a development of the previous work in this area, i.e., [Puschner2001a]. It will contain various rules and guidelines specific to Java, plus a number of new classes that makes use of the existing RTSJ classes. The profile effectively eliminates features with high overheads and complex semantics, so that programs become more analysable and ultimately, more *dependable*. Yet, most of the key features of Java and RTSJ, such as dynamic dispatching and scoped memory areas, are still retained, resulting in a practical and flexible programming environment[1]. The profile is intended for use within single processor systems as of now. Possible extensions will be discussed at the end of this chapter as well as in Chapter 7. Appropriate rationales for the decisions made will also be given.

This chapter is structured as follows: the next section sets the scope and describes the guiding principles of the profile. In section 2, we show the overall computational model and organisation, before the profile is illustrated in detail in

[1] Some of the features may require a more complex analysis, but they are analysable according to recent research in this area. Examples are given in Chapter 5.

terms of the APIs. Section 4 briefly looks at implementation issues, followed by a simple example program. Possible extentions will be discussed afterwards. The full description of the rules and guidelines of the profile is provided in Appendix A.

4.1. Scope and Guiding Principles

As we have seen, 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) [NUREG/CR-6463] because unlike other standards it covers programming issues specific to high integrity systems. It has set up a systematic framework of guidelines by deriving many important attributes from influential standards. It identifies 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, diver-sity and exception handling are commonly used [Burns2001].
- 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 [Sommerville2000].

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.

Essentially, the goal here is to apply the NRC's framework coupled with our own assessment criteria to Java and the RTSJ (see Appendix A for a complete list of the profile's guidelines that are in line with the NRC's framework). The profile 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.

The work of Puschner and Wellings [Puschner2001a] has served as a starting point for the profile proposed in this chapter. The issues addressed in their work are very much relevant to high intergiry systems, and the Ravenscar-Java profile inherits most of the features, i.e., the model for tasking and object oriented features. Furthermore, attempts have been made to extend such features or address issues that were not dealt with. Some of the key areas that are new in this profile are as follows.

- Categorisation of features according to high integrity standards

- Definition of rules that minimize syntactic as well as semantic errors
- Support for a more flexible memory model that are still predictable
- Annotation support for temporal and memory usage analysis

4.2. Overview of the Computational Model and Organisation

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 [Burns1998] and emphasise the *reliability* attribute of the NRC guidelines, as just mentioned. 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 compliant virtual machine 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 must be either periodic or sporadic with minimum inter-arrival times, and the priority ceiling protocol is required to be implemented in the runtime system. Aperiodic events and handlers are removed due to the inability to perform realistic schedulability analysis. 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 [Audsley1993, Burns2001, Liu1973]).

There are two execution phases as suggested in [Puschner2001a], i.e. *initialisation* and *mission* phase, as shown below in Figure 4.1. This is to support a controlled way of program execution. In the initialisation phase of an application (i.e. the *main*() method and one *RealtimeThread*), all non-time-critical initialisations that are required before the mission phase are carried out. Hence the mission phase will not be interfered by system-level activities. This includes initialisation of all real-time threads, memory objects, event handlers, events, and scheduling parameters[2]. In the mission phase, the application is executed and multithreading is allowed based on the imposed scheduling policy.

The profile's APIs are organised in a way that is compatible with the RTSJ, but they are always more restrictive. Several methods of Java's original APIs (e.g., `java.lang.Thread`) are also removed or restricted, meaning that programmers cannot directly use or extend them. There are also new APIs defined, which are implemented using the existing RTSJ's services. Any Ravenscar-Java programs are valid programs executable on any reference implementations of the RTSJ, albeit some non-functional semantics may differ, especially timing attributes. Refer to Appendix B for the list of APIs of the Ravenscar-Java profile.

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

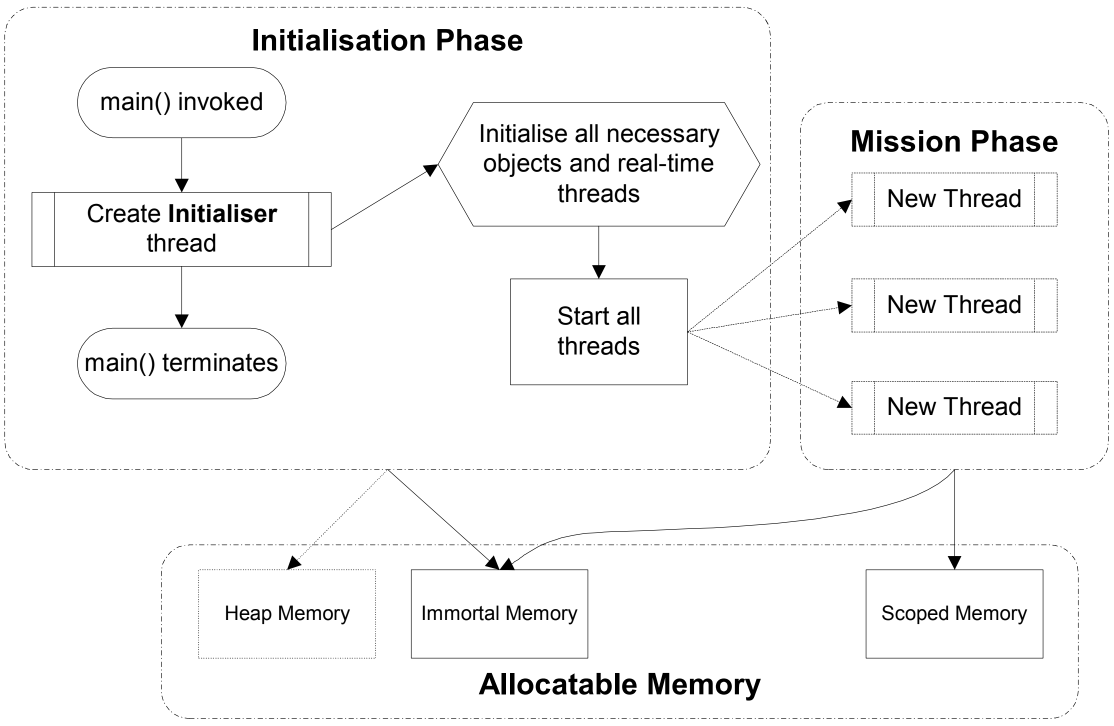


Figure 4.1. Two execution phases

4.3. The Profile

In this section, the rules and guidelines of the proposed profile are given in relation to the framework of the criteria developed in Chapter 3, i.e., the 23 requirements categorised into two levels. This section shows how the profile attempts to meet the requirements. Those areas that Java and RTSJ already excel are not elaborated further, but appropriate references to the assessment in Chapter 3 will be given instead. For a similar reason, guidelines are not given for certain areas that are considered out of scope of the profile, for instance, L.1.1.4 Formal semantics/International standards and L1.1.5 Well-understood.

In addition to the organisation of the profile in this chapter, Appendix A also provides the profile ordered from the perspectives of the following sub-categories, utilising the NRC guidelines' convention. The purpose of that is to allow practioners to quickly learn and put them into practice. Guidelines and rules are given under each heading.

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.

The following part of this chapter puts the profile into context in relation to the criteria from Chapter 3, and appropriate references are given in parentheses after the headings in order to keep the traceability. Please note that because some of the rules below overlap with others in different criteria, more than one references are given when required.

For the sake of presentation, the rules regarding syntax and control flows (i.e., issues on programming in the small) are not elaborated here, but are deferred until Appendix A.3. In the Appendix A, those that are identified as *Rules* all belong to Level 1, whereas the guidelines are for Level 2.

4.3.1. 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. In the case of Ravenscar-Java, the following issues are to be addressed.

a) Initialisation and Mission phases

(Requirements fulfilled: **L1.2.1, L1.2.2, L1.2.3)**

When an application is started, its *main*() method will first be invoked by the virtual machine 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, as illustrated in Figure 4.2. Any code that is executed in the *run*() method is considered to be in the Initialisation phase of the application program. Threads that are started as a result of the Initialisation phase are all said to be in the Mission phase, after which the execution of the initial *run*() method comes to the end. The distinction between two different execution phases is required to ensure a well-ordered and controlled execution of programs. Without this, critical threads may have to suffer from avoidable system level activities that could be moved to a non-critical phase, which include dynamic class loading, creation of memory objects and threads and so on.

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

  public void run()
  {
    // initialization phase of the program
    // Will start off application threads here
  }
}
```

Figure 4.2. An illustration of the initialisation phase

The new thread, *initializer*, must take a reference to the *immortal* memory area, so that all objects and references to other threads and memory objects created 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 the *Initializer* thread class, shown in Figure 4.3, which directly extends the **RealtimeThread** class.

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

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

Figure 4.3. **Initializer** class of Ravenscar-Java profile

Now, an 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: create threads, memory objects
    // Start application threads
  }

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

The mission phase begins as soon as the highest priority thread (i.e. the *Initializer* thread) 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[3]. In order to prevent overflows in immortal memory, threads are not allowed to create further objects in the area in the mission phase.

b) Memory Management

(Requirements fulfilled: **L1.2.3)**

To facilitate predictable memory utilisation we define several rules in the three aforementioned areas (see Appendix A for the full list and rationales). The rules place restrictions on, for example, the use of class loaders in the mission phase, the use of specific memory area objects (and garbage collector), and 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,

[3] In the profile, we do not attempt to restrict the use of physical or raw memory other than that implied by our restrictions on scoped memory areas. However, a potential implementation of an RVM might apply restrictions for security reasons.

since no garbage collection is allowed in the profile. Below are further details on the use of each type of memory areas.

• 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 [Bollella2000a, RTSJ2005]. Hence, in an attempt to prevent memory exhaustion or corruption, objects (including memory area objects) that are needed for the lifetime of the application must be allocated in the area only in the initialisation phase. In other words, in the mission phase, no threads are allowed to create further objects in immortal memory, such that memory leaks will not occur.

• Use of linear time scoped memory areas

The goal of scoped memory is to allow program logic to allocate objects with a limited lifetime and predictable reclamation pattern. In particular, linear time scoped memory is guaranteed by the underlying system to have linear allocation time [Bollella2000a, RTSJ2005].

In programs that conform to the Ravenscar-Java profile, 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 only make use of linear time scoped memory areas (i.e. *LTMemory* areas). The size of all memory objects must have a fixed upper bound and not be extended during the course of the program, since extending the size of an area will cause run-time overheads. 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 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 4.4. Simplified memory area classes

To aid in the production of an efficient virtual machine and to simplify timing and memory usage analyses, access to *LTMemory* areas must not be nested and such areas must not be shared between *Schedulable* objects. Otherwise, the virtual machine will have to perform heavy *assignment checks* and enforce the single parent rule at run-time [RTSJ2005, Bollella2000a], which will cause a significant amount of overheads. With this restriction, any assignment to memory in an *LTMemory* area can always be allowed without checks. This is because the referenced object must always have the same scope or be in immortal memory. However, any assignment to memory in immortal (or heap) memory area must be checked to ensure the referenced object does not reside in an *LTMemory* area as this could potentially result in a dangling reference. The goal is to perform static analysis to validate this rule before program execution [Kwon2003a].

RTSJ exceptions, such as *ScopedCycleException* and *OutOfMemoryError*, are not to occur at run-time because we rely on static analysis and the profile requires access to memory areas not to be nested. Hence, the methods will no longer throw such exceptions, thus excluded from the profile (commented out, as shown in Figure 4.4).

4.3.2. 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 previously and, thereby, allow feasible schedulability analysis to be performed.

a) Scheduling and Threading Model

(Requirements fulfilled: **L1.1.7, L1.2.2)**

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 *setter* methods, whose names begin with '*set'* (for example, *setReleaseParameters*()), and some *getters* 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 4.5. 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 on-line feasibility checks, leading to the classes in Figure 4.6. 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 int getMaxPriority();
  public int getMinPriority();
}
```

Figure 4.6. Simplified **Scheduler** classes

Overall, this approach does not necessitate any dynamic feasibility test and admission control by the virtual machine at runtime. All timing and schedulability analyses must be performed before the initialisation phase of the program.

b) Use of release parameters

(Requirements fulfilled: **L1.2.2)**

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 4.7. **ReleseParameters** and its subclasses

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

c) Use of threads

(Requirements fulfilled: **L1.1.7, L1.2.2)**

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. Hence, this class is defined as follows[4].

```
package java.lang;
public class Thread implements Runnable
{
  Thread();
  Thread(String name);

  void start();
}
```

Figure 4.8. 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 4.9. **RealtimeThread** and **NoHeapRealtimeThread** class

[4] 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.

• 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 must not be used.

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 4.10. **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 4.11. 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 the sake of 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 4.12. Event handlers and the **SporadicEventHandler** class

Classes associated with event handlers are shown in Figure 4.13 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 4.13. 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 is performed off-line.

d) Synchronization

(Requirements fulfilled: **L1.1.7, L1.2.1, L1.2.2, L1.2.3)**

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. 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 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 since they are provided in the RTSJ to enable communication between instances of *NoHeapRealtimeThread* and regular Java threads.

e) Representation of time

(Requirements fulfilled: **L1.2.2)**

Supported representations of time are

- `HighResolutionTime`
- `AbsoluteTime`
- `RelativeTime`

These classes allow representation of time with up to nanosecond accuracy and precision [Bollella2000a, RTSJ2005].

f) Timer classes

(Requirements fulfilled: **L1.2.2)**

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.

g) Asynchrony

(Requirements fulfilled: **L1.1.7, L1.2.2)**

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 [Brosgol2002].

h) Exceptions

(Requirements fulfilled: **L1.2.1, L1.2.2, L2.1.1)**

Exceptions are allowed in the profile in a limited way. All exceptions must be bound at the thread-level; they cannot go over the boundary of or between threads. In other words, each thread must be able to catch all exceptions that it can raise (the worst would be catching *Error*), so that such exceptions are contained within all subject threads. So, if one application thread cannot progress because of an exception occurred inside it, it must be able to catch it and either proceed with a recovery code, return to normal execution, or even terminate. That will not have any implication on the real-time scheduling. Of course, if other threads are dependent on an ill behaving thread with an exception raised, it could be a problem, but still such cases are analyzable by looking at the dependent threads and what exceptions they could raise. In fact, threads cannot be dependent on each other by locking or waiting, etc since they are not allowed in the profile.

4.3.3. 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. All the rules and guidelines are listed in Appendix A, but noteworthy ones in each of the three areas include the following, which will facilitate or greatly ease the use of program analysis tools. Note the distinction between the terms *must* and *should* here and in Appendix A; the former means it is a *rule* that must be obeyed, while the last a *guideline* that is a highly recommended practice.

- **Programming in the large**
  - All user-defined classes *must* include constructors that initialise all internal variables and objects (Requirements fulfilled: L1.1.1, L1.1.2).
  - Dynamic method binding *should* be minimised. In particular, method overriding and the use of interfaces *should* be minimised. It is unreasonable in many cases to disallow dynamic binding altogether (Requirements fulfilled: L1.1.1).

- **Concurrent Real-Time Programming**
  - Asynchronous transfer of control (ATC) and any thread aborting mechanisms are disallowed (Requirements fulfilled: L1.2.2).

- Use of *wait*(), *notify*(), and *notifyall*() methods is disallowed (Requirements fulfilled: L1.2.1, L1.2.2).

- **Programming in the small**
- Use of *continue* and *break* statements in loops is disallowed (Requirements fulfilled: L1.2.1).
- All constraints, such as one used in a *for* loop, must be static (Requirements fulfilled: L1.1.2).
- Compound expressions in parameter passing to methods must be eliminated (Requirements fulfilled: L1.1.2).
- Expressions whose values are dependent on the order of evaluations must be disallowed (Requirements fulfilled: L1.1.2).

4.4. An Example Program

Here, for the purpose of an illustration, we present a simple and naive traction-control system. It senses differences between the front- and rear-wheel spin speeds, and reduces the engine output if the rear wheels spin more quickly[5]. There is one periodic thread, *SpinMonitor*, and one sporadic thread, *powerCutHandler*. As soon as an excessive rear-wheel spin is detected, *SpinMonitor* activates *powerCutHandler*. Only a sketch of the real application logic is given as the example is purely intended to illustrate how an application may look in the profile. The class diagram is given below. For a larger example, refer to Chapter 6.

[5] A rear-wheel drive car is assumed for the purpose of illustration.

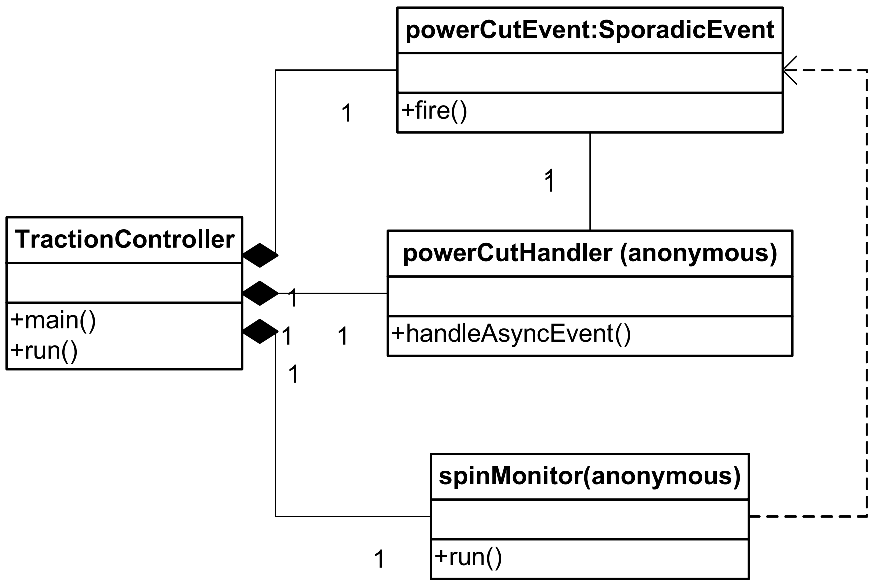


Figure 4.14. Class diagram of *TractionController*

```
import ravenscar.*;
import javax.realtime.AbsoluteTime;
import javax.realtime.RelativeTime;
import javax.realtime.ImmortalMemory;

public class TractionController extends Initializer
{
  public void run()                    // Initializer routine
  {
    // powerCutHandler
    SporadicEventHandler powerCutHandler = new SporadicEventHandler (
        new PriorityParameters(15),   // Priority:15
        new SporadicParameters(
          new RelativeTime(333, 0),   // Minimum interarrival time
          5)                          // Buffer size
    )
    {
      public void handleAsyncEvent() // Event handler routine
      {
        // Logic for handling powerCutEvent event
        // i.e. either cut the engine power or brake appropriate
        // wheels
        …
        long rpm = engine.getCurrentRPM();
        if ((rearWheelSpinRate+THRESHOLD)>(frontWheelSpinRate+LIM1){
            if ((rpm >= 4000) && (rpm < 5000)) {
                engine.slowDownByRPM(1000);
            } else if ((rpm >= 5000) && (rpm < 6000)) {
                engine.slowDownByRPM(2000);
            }
        } else // LIM1 and LIM2 are used to signify the difference
        if ((rearWheelSpinRate+THRESHOLD)>(frontWheelSpinRate+LIM2){
            if ((rpm >= 4000) && (rpm < 5000)) {
                engine.slowDownByRPM(1000);
                brakeSystem.applyRearBrake(5); // 5% of brake
            } else if ((rpm >= 5000) && (rpm < 6000)) {
                engine.slowDownByRPM(2000);
                brakeSystem.applyRearBrake(10); // 10% of brake

            }
        }
      }
    };

    final SporadicEvent powerCutEvent =
      new SporadicEvent(powerCutHandler);

    // spinMonitor
    PeriodicThread spinMonitor = new PeriodicThread(
      new PriorityParameters(10),     // Priority:10
      new PeriodicParameters(
        new AbsoluteTime(0, 0),       // Start time
        new RelativeTime(333, 0)),    // Period
      new spinMonitorLogic());

    spinMonitor.start();
  }

  public static void main (String [] args)
  {
    TractionController init = new TractionController();
    init.start(); // Start the initializer
  }
}

public class spinMonitorLogic implements Runnable {
    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
      …
      if ((frontWheelSpinRate + THRESHOLD) < rearWheelSpinRate){
           powerCutEvent.fire();
      }
    }
}
```

Figure 4.15. *TractionController*

In terms of memory usage, the main class that extends *Initializer* resides in immortal memory. The other two mission threads will be executed on their own scoped memory areas. The sizes of all memory areas will be fixed once set and the upper bounds must be correctly estimated.

4.5. Extensions to the Profile

The profile has been engineered for use in high integrity systems. It means that we took a very conservative approach, in which a number of useful features have been taken out due to their non-deterministic runtime characteristics or difficulties in analysis. However, with the help of programming techniques or annotations support, such valuable features can be incorporated without overheads. The following lists possible extensions to the profile. Annotations, in particular, make analysis more accurate by conveying additional information.

4.5.1. Single Nested Scoping

(Requirements fulfilled: **L1.2.3)**

Currently memory management in the profile is performed as follows: having prohibited the use of any automatic garbage collection mechanism, each thread has a single dedicated scoped memory area used in each release. Immortal memory is only used for the initialization phase and shared objects. The downside of this approach is that until a thread runs to the end of each release, used memory will never be reclaimed. One has to face the task of estimating the amount of memory that will be used during each release. This is trivial if the thread performs an undemanding job, for example, a simple sensor reader. However, this is not true when there are complex logics inside involving several loops and method calls to existing libraries. They may create a considerable number of temporary objects, whose sizes are difficult to estimate in advance. Hence, a limited form of dynamic memory allocation will facilitate more efficient memory usage while maintaining the predictability of the current model. Single Nested Scoping is a viable extension to the profile, as outlined below.

• Single Nested Scoping: Definition and Issues

By single nested scopes, we mean a stack of dedicated memory areas that only one thread can enter during each release. No cactuses are allowed, nor are multiple entries to a memory area (enforced by the RTSJ at runtime). This also means that threads will not share any memory areas apart from the immortal memory.

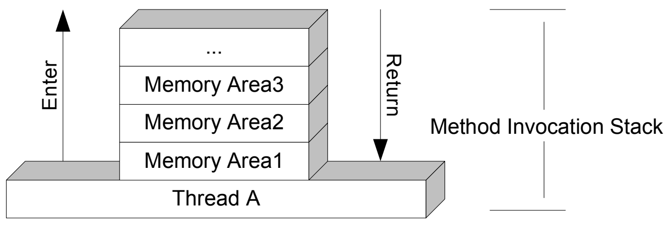


Figure 4.16. An illustration of Single Nested Scopes

If single nested scoping is allowed, the following advantages are observed.

1. Memory areas/space can be reused.
2. A limited form of “garbage collection” is performed at programmer’s will.
3. A single stack of memory areas will be created, i.e., no cactuses, so that reference checks become less demanding (e.g., the complexity of escape analysis), meaning that *executeInArea*() is not permitted.
4. Assuming that all memory areas are created in the Initialization phase, overheads for creating them at runtime are cast out in the Mission phase.

In short, this will result in a more flexible memory management model within each release of a thread. We still have the predictability of the current model. However, the time taken to finalise objects in all memory areas should be considered in WCET/schedulability analyses, which is required in our current model as well.

One must also take into account the following facts.

1. References to memory area objects could be accessed by other (unauthorised) threads, so that a memory area can be shared illegally by two or more threads.
2. The Single Parent Rule must be enforced in order to prevent dangling references.

These concerns can be eliminated by our proposed approach outlined below (see the updated *PeriodicThread* and *SporadicEventHandler* class below, and in Appendix C). Notice how it hides all memory areas and finds the most suitable, best-fit area for any particular request, i.e., see *enter*(). The Single Parent Rule is also unnecessary because threads cannot obtain the reference to any memory areas.

Our proposed approach adds a few methods to the *PeriodicThread* class as shown below in the simplified code. When an instance of *PeriodicThread* is created, the number and size(s) of memory areas to be used must be specified in the constructor. That way, the constructor will automatically create required memory areas in the Initialization phase. All subsequent use of memory areas will involve calling the *enter*() method provided, which takes a *Runnable* and a long value for the memory size required. The method will search for a most suitable area from the list of memory areas it maintains in a constant time span – the worst case is bounded by the number of scoped areas.

```
public class PeriodicThread extends NoHeapRealtimeThread {
…
      public PeriodicThread(PriorityParameters priority,
                  PeriodicParameters period,
                  Runnable logic,
                  long[] memSizes) {…};

      public final void run() {…};

      public final static void enter(Runnable logic, long memSize)
      {…};

      public final static Object cloneInParentArea(MACloneable o) {
            // Create an object in the parent MA and return the ref.
            return cloneInParentArea(o, 1);
      };

      public final static Object cloneInParentArea(MACloneable o,
                                                       int index){…};

                        public void start() {…};
}
```

When there is a need to copy an object in to a parent area, *cloneInParentArea*() should be used. The parameter object whose type is of *MACloneable* must override the *clone*() method. That method will be executed after entering an appropriate parent memory area (direct parent area by default). The one that takes an integer index value allows the program to store an object in any parent memory area: 0 meaning the current area, 1 the direct parent, etc.

```
package ravenscar;

public interface MACloneable extends Cloneable {
      public Object clone();
}
```

Similar to *PeriodicThread*, the *SporadicEventHandler* class can be modified to include the same methods, i.e., *enter*() and *cloneInParentArea*(). Its constructors will take an array of integers representing the number and size(s) of memory areas to be used.

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

      public final static void enter(Runnable logic, long memSize)
      {…};

      public final static Object cloneInParentArea(MACloneable o) {
            // Create an object in the parent MA and return the ref.
            return cloneInParentArea(o, 1);
      };

      public final static Object cloneInParentArea(MACloneable o,
                                                        int index){…};
};
```

• Programming Constraints

For entering a memory area, we only allow the '*enter*()' method of the *MemoryArea* object (now implemented in *PeriodicThread*). It takes a *Runnable* object as parameter, and the only ways to pass objects (in fact, references) to the *Runnable* running in a new memory area are

1. At the creation time of the *Runnable* object through the constructor,
2. Through *setter* methods of the *Runnable* object before entering a MA, and
3. While the *Runnable* object is operational, it may call a method of an object in the previous allocation context, and gain access to the returned object. To do this, it needs to obtain the reference to the caller object or a call-back object created in a parent area. This is done based on case 1 or 2 above.

In case 3, there is a possibility that a reference to a local object can be passed to the parent context, which can lead to a dangling reference when the current memory area is exited[6]. For this reason, all call back methods to be used in such a context must not attempt to store parameters in the parent memory area. An illustration is given below in Figure 4.17, where

Parent Memory Area (Caller)	Child Memory Area (Callee)
`class TracCnt implements Runnable`	`class Logic1 implements Runnable`
`{ …`	`{`
`  // Callback obj. of some sort`	`  private callBack toCall;`
`  callBack a = new callBack();`	`  public Logic1(callBack a) {`
`  Logic1 l1 = new Logic1(a);`	`    toCall = a;`
`  …`	`  }`
`  // Logic1's Runnable is entered`	
`  ct.enter(l1, 10000);`	`  public void run() {`
`  // ct refers to CurrentThread`	`    localObj lo = new localObj();`
`  …`	`    toCall.get().set(lo);`
`  localObj result = a.get();`	`  }`
`}`	`}`

Figure 4.17. An illustration of incorrect object passing using call-backs between two memory areas

Unlike method invocations, *Runnables* cannot directly return an object back to the calling context. When necessary, this can be achieved through call-backs again, in which a *Runnable* takes an object as parameter from the calling context; then, call a *setter* method of that object. If such a *setter* method takes a new object, which is created inside the *Runnable*, and assigns the new parameter to an object in the parent memory area, then an assignment violation will occur. In other words, this sort of call-back mechanisms must be used with caution because, unlike method calls, memory areas are now involved between calls.

[6] This is essentially an assignment violation and must be checked at run-time by an assignment-check enabled VM. A Ravenscar-Java compliant VM may not need to perform such checks at run-time thanks to static analysis and annotations.

Reference assignments are subject to run-time checks, and it is one of our prime goals to statically eliminate such costly checks and exceptions. Since the newly proposed *PeriodicThread* class above does not let the user code gain access to *MemoryArea* objects, it is necessary to provide a functionality to cope with copying objects between parent and child memory areas without creating illegal references. The two *cloneInParentArea*() methods are added for this purpose. *MACloneable* is a new simple *tagging* interface that a cloneable object must implement. It documents the programmer's intention to copy a specific object to a parent memory area, and its *clone*() method must re-*new* or recreate all its internal objects, so that they will be created in the parent area.

These cloning methods are used in association with a call-back object. When a *Runnable* is to return an object to a calling context, a clone of that object will be generated in the parent memory area by calling a *cloneInParentArea*() method, and a reference to that new object will be passed to the calling context via the call-back practice. For an instance, assume there are two *Runnable*s, *R1* and *R2*, the former using memory area *M1*, the latter *M2*. While *R2* is being executed, a result from a complex calculation is produced and must be returned to *R1*. *R1* is expecting this, so that it has created a call-back object and passed it to *R2*, as shown below.

```
public class R1 implements Runnable {
  public R1(){
    // This object itself may be created in the Immortal Memory.
    // But enters a new Scoped MA selected by PeriodicThread’s enter,
    // which means every object created here in the constructor
    // will be created in the Immortal Memory Area.
  }

  public void run() {
    R2 r2 = new R2;     // R2 is to return a result

    callBack cb;        // A Call-back Object to use
    // This call back object will be used to pass object references
    // whose objects are created in a new MA, and to be cloned into
    // a parent MA.
    cb = new callBack();
    r2.setCallBackObject(cb);

    PeriodicThread ct = (PeriodicThread)Thread.currentThread();
    System.out.println("R1: Entering into R2.");
    ct.enter(r2, 10000);

    System.out.println("R1: Just Returned from R2.");

    Object result = cb.get(); // returned obj is created in this MA
    System.out.println("R1: Got the reference to result.");

    // Print out the result that was created in R2
    result.toString();
  }
}
```

Figure 4.18. *Runnable R1* making use of a callback object to obtain result from a child memory area

The code of *R2* is shown below. Notice how the cloning method is used, and the reference to the new object passed to the calling context, *R1*, via the call-back object. Without such a practice, the R1 has no other way to obtain the reference to the new object (except using the shared immortal memory). A usage of single nested scoping is also given in Chapter 6, where a realistic case study is explored.

```
public class R2 implements Runnable {

  callBack cb;

  public R2() {
    cb = null;
  }

  public void setCallBackObject (callBack c) {
    cb = c;
  }

  public void run() {
    // resultToReturn is what we want to return to the direct parent MA!
    data resultToReturn = new data();

    PeriodicThread ct = (PeriodicThread)Thread.currentThread();

    // So we clone it in the Parent MA, as in the following
    Object r = ct.cloneInParentArea(resultToReturn,1);
    // the index parameter in cloneInParentArea, if given, is the
    // index from the current MA. So 0=the current. 1=direct parent MA...

    System.out.println("R2: Just cloned data to the parent area!");

    // Now r is the reference to resultToReturn that is
    // cloned into the outer MA!
    // In order to return the reference, we use the callback object cb
    cb.set(r);
  }
}
```
Figure 4.19. *Runnable R2* making use of a callback object to return result to a parent memory area

4.5.2. Annotation Support for Program Analysis

(Requirements fulfilled: **L1.2.1, L1.2.2, L1.2.3)**

Annotations are of great help in conducting various analyses. For example, loop bounds for the worst-case execution time (WCET) and memory usage analysis can be documented and checked by an external tool; these are essential in analysing hard real-time systems, and users can provide such extra information by means of annotations.

Currently, there are two major annotation languages/mechanisms for Java, i.e., the Java Modelling Language (JML) [JML] and Java 5's Metadata [JSR175]. The JML has been around for some time now and has matured into an expressive language.

It is straightforward to check pre- and post- conditions, and users can even introduce *ghost* variables that, hidden from the program, are only used by annotations. A front-end of the compiler is required that translates annotations into Java code, which means annotations become part of the consequential code. At run-time, this may introduce significant overheads, not desirable in a resource sensitive system. Conditions should only be checked in testing.

The Metadata is a new addition to the Java 1.5 release. It is supported directly by compliant compilers and virtual machines, such that annotations can be embedded into *classfiles* at the developer's will and used by the program itself at runtime using the provided library. New annotations can be defined as required, and can be incorporated into and utilised by a static program analysis tool. One of the major drawbacks, however, is that only *declarations* (i.e., classes, methods, parameters, packages, parameters and annotation types) can be annotated, while arbitrary statements, such as *for*-loops, cannot.

Considering the run-time overheads of checking embedded JML annotations, it would be a step in the right direction to use the lightweight Metadata to annotate code, and develop a separate checker. That way, static analysis of code that does not hamper run-time performance can be developed. Obviously, annotations must first be defined for specific purposes, such as checking assignment violations and escaping objects from a given memory area. Loop bounds are not easy to annotate since loops, such as *while* and *for* statements, are not declarations. Yet, they are indispensable in the WCET and worst-case memory usage analyses.

• Annotations for Memory Safety

It is one of the main goals of the profile to reduce overheads caused by run-time assignment checks. This can be avoided by statically analysing code, and annotations make the analysis more efficient. Annotations can also be used to document the developer's intention on how methods should behave and be reused by others.

When considering a method call on an object, the following factors determine whether that call may result in an assignment violation.

- The memory area where "this" object (i.e., the invoked method's object) was created. If the defining class has any instance reference variables, then these can be used by the object to hold references to other objects which may reside in different memory areas.
- The memory area(s) where any parameter objects reside. A method or constructor may assign references to parameter objects in the defining class's field variables (instance of static), or field variables reachable from the defining class's reference variables. Also it may assign references to objects reachable from parameter objects.
- The memory area of the current allocation context of the calling and called method. The called method may allocate new objects and assign references to them in the defining class's field variables (instance or static) or field variables reachable from the defining class's reference variables or parameter objects.
- The memory area of objects whose references are returnable. Returned object references may be assigned by the calling method. These references may refer to objects newly created within the current allocation context (using *new*), or

they may refer to pre-allocated objects reachable from the defining class's reference variables (instance or static) or they may refer to pre-allocated objects reachable from method's parameter objects.

- The memory area of any objects whose references may be thrown by exceptions. Thrown exception object references may be assigned by the calling method. These references may refer to objects newly created within the current allocation contexts (using *new*), or they may refer to pre-allocated objects reachable from the defining class's field variables (instance or static) or they may refer to pre-allocated objects reachable from method's parameter objects.

We propose the following annotation for documenting the programmer's intention and checking any violations of memory safety/assignment rules. This annotation is used for each method because method invocation is one of the only two ways in which object references can be passed to and from different memory areas. The other way is to reference directly a public attribute of another object. However, this is not a good programming practice as it increases coupling between objects.

```
import java.lang.annotation.*;

@Retention(CLASS)
@Target({METHOD})
public @interface ScopeSafe {
  public Assign[] value() default {};

  public static @interface Assign {
    public int[] from();
    public int to();
  }

  // Identifiers for various objects
  public static final int PARAMETER = 0;
  public static final int ANY_PARAMETER = -1;
  public static final int STATIC = -2;
  public static final int RETURN = -3;
  public static final int EXCEPTION = -4;
  public static final int THIS = -5; // Method's Object
}
```

Figure 4.20. Definition of the *@ScopeSafe* Annotation

This annotation provides a means of documenting for the developer when it is safe to use a certain class or method and what rules apply to the implementation of subclasses. Objects are grouped into PARAMETER, STATIC, RETURN, EXCEPTION, and THIS. As the names imply, they mean parameters, static objects, return objects, exception objects, and *this* object (to which the method belongs) respectively. Parameters can be indexed by adding a natural number to PARAMETER, for instance, a third parameter in a method call can be represented as PARAMETER + 2 in the specific annotation for the method. Note that the index begins with 0. Static objects are, by definition, reside in immortal memory. The inner interface called *Assign* is there to describe the assignment relationships between objects that are dealt with by the subject method.

This annotation can be used without any arguments, meaning that the given method can be used in any scoped memory context. In particular, this means that the parameters given to the method do not escape the method in any way and the return value does not reside in scoped memory. *@Assign* phrases may be added to better describe or limit the method's behaviours. For instance, consider the following method. The method, *deposit*() can only assign the parameter to the object that the method belongs to.

```
@ScopeSafe(@Assign(from = PARAMETER, to = THIS))
public boolean deposit(Object element) {
…
}
```

A conforming checker will examine the correctness of the annotation by inspecting the subject method. Once all methods' annotations are checked, one can conduct inter-method analysis, taking into account memory area information. Inter-thread analysis is not required because memory areas are not shared between threads. In order words, this is a two-part analysis, i.e., checking the correctness of annotations for each method, and checking the interplay between methods in a thread. Call graphs will be constructed as a result of the latter, which will then be mapped onto the memory areas exploited.

If used without the single nested scoping approach proposed early, there are only two memory areas to consider. There are six possible combinations of memory areas and related objects in the mission phase, where the allocation context must always be within scoped memory for an application thread (because immortal memory is only used in the initialization phase). The cases are illustrated in the figure below. New objects, exceptions, returned objects do not appear because they are created in the current allocation context and will not affect the semantics in general cases[7].

Case 1. The reference variables of “this” object are held in scoped memory. The parameter objects are in scoped memory. Assignments in both directions are possible, so that the following annotations are valid (only considering parameters and “this” object).

```
@ScopeSafe(@Assign(from=THIS, to=PARAMETER))
@ScopeSafe(@Assign(from=PARAMETER, to=THIS))
@ScopeSafe()
```

Case 2. The reference variables of “this” object are held in scoped memory. The parameter objects are in immortal memory. “This” object cannot be referenced by the parameters, whereas the opposite is valid. The following annotation is valid.

```
@ScopeSafe(@Assign(from=PARAMETER, to=THIS))
```

Case 3. The reference variables of “this” object are held in scoped memory. Some of the parameter objects are in scoped memory, others are in immortal memory. The following annotations are valid.

[7] However, there are cases when this is not true. For instance, some exception objects may be pre-allocated in immortal memory in the initialization phase, and thrown in the mission phase.

```
@ScopeSafe(@Assign(from=THIS, to=PARAMETER+0))
@ScopeSafe(@Assign(from=PARAMETER+0, to=THIS))
@ScopeSafe(@Assign(from=PARAMETER+1, to=THIS))
```

Case 4. The reference variables of “this” object are held in immortal memory. The parameter objects are in scoped memory. The following annotations are valid.

```
@ScopeSafe(@Assign(from=THIS, to=PARAMETER))
```

Case 5. The reference variables of “this” object are held in immortal memory. The parameter objects are in immortal memory. The following annotations are valid.

```
@ScopeSafe(@Assign(from=THIS, to=PARAMETER))
@ScopeSafe(@Assign(from=PARAMETER, to=THIS))
@ScopeSafe()
```

Case 6. The reference variables of “this” object are held in immortal memory. Some of the parameter objects are in scoped memory, others are in immortal memory. The following annotations are valid.

```
@ScopeSafe(@Assign(from=THIS, to=PARAMETER+0))
@ScopeSafe(@Assign(from=THIS, to=PARAMETER+1))
@ScopeSafe(@Assign(from=PARAMETER+1, to=THIS))
```

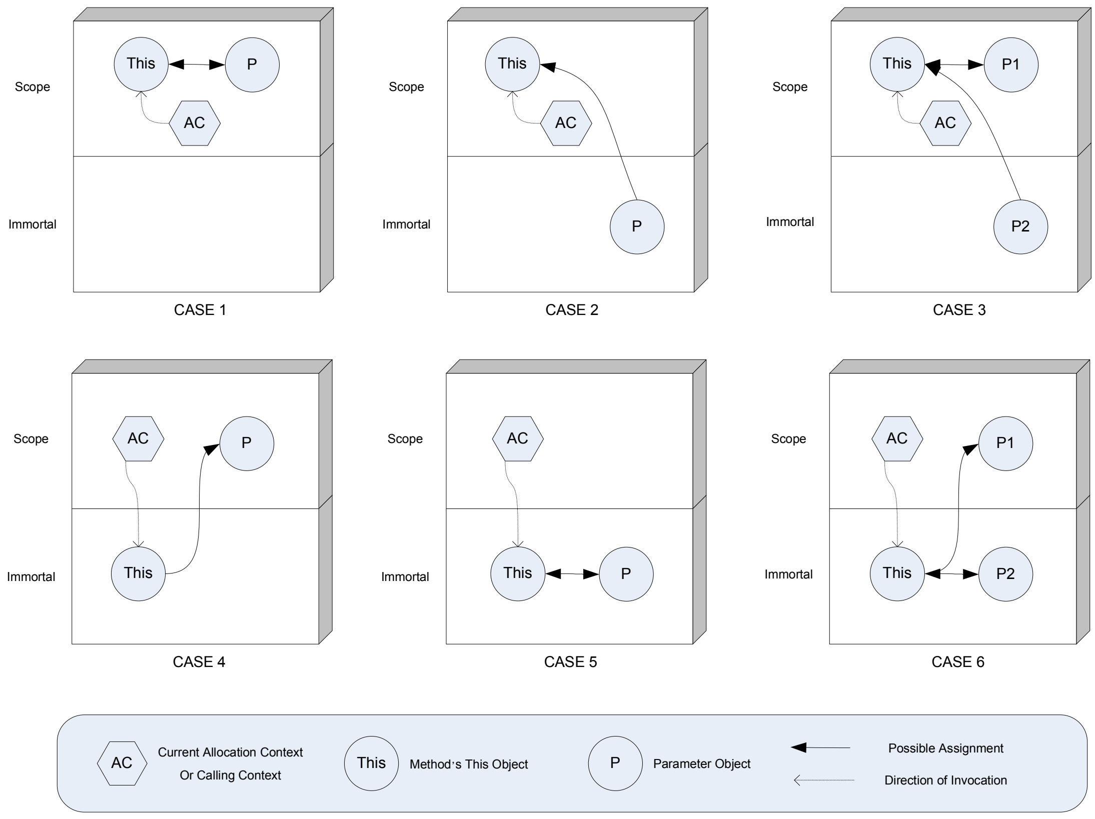


Figure 4.21. Six cases of possible allocations with two memory areas

In the Initialization phase, the allocation context is the immortal memory area. Until the mission phase begins, all allocations take place in this area. Assignments to both parameters and "this" objects are allowed, as in Case 1.

The goal the annotation is to enable static analysis tools to expose potential illegal assignment and reference relationships between objects, thus eliminating the need for run-time checks and exceptions.

• Annotating Loop Bounds

The Metadata can only annotate declarations, which is seen as a disadvantage. One way to annotate loop bounds is to declare a loop counter variable and specify its maximum value of it. In many *for* loops, this is straightforward as the counter is embedded in the construct, as shown in the following example. As the variable *i* is a new declaration, it can be annotated using a Metadata type.

```
…
for (@MaxLoopBound(1000) int i = 0; i < 1000; i++)
{
    // code to execute
}
…
```

We propose a new annotation *@MaxLoopBound*, whose definition is shown below. It is used to document the maximum loop bound of a loop. The *for* statement above has already been annotated.

```
import java.lang.annotation.*;

@Retention(CLASS)
@Target({FIELD, TYPE})
public @interface MaxLoopBound {
  public int value();
}
```

Figure 4.22. Definition of the *@MaxLoopBound* annotation

For *while* and *do-while* statements, such counter variables may be missing. Instead, certain Boolean expressions will be checked, leading either to more repetitions or halting of the loop. In cases like this, we can introduce a new (dummy) declaration of a variable only for documenting the maximum loop bounds. Such variables may be of any type as long as it can be annotated.

```
…
while (index < THRESHOLD)
{    @MaxLoopBound(1000) boolean dummyLoopBound = true;
     // code to execute
}
…
```

Analysis tools will have to locate the beginning and end of each loop, and read the annotation's value, in order to calculate the WCET or worst-case memory usage.

• Annotating Modes

As explained in [Chapman1994], a mode is a subset of states that can be exhibited by a subprogram's preconditions or invariant assertions. So, depending on different modes that must be mutually exclusive to each other, various assertions can be made during timing analysis. For more information on the concept of the mode annotation, please refer to [Chapman1994]. To implement the mode annotation, the following interface type has been devised.

```
import java.lang.annotation.*;

@Retention(CLASS)
public @interface Mode {
  public String mode_label();        // Mode Label
  public String assertion();         // Assertion
}
```

Figure 4.23. Definition of the *@Mode* annotation

The annotation could be used in the following example.

```
public void foo(int a, int b)
{
  @Mode(mode_label = "A", assertion = "a <= 0")
  @Mode(mode_label = "B", assertion = "a > 0")
  int x = 0;

  for (@MaxLoopBound(30) x = 0; x < b; x = x + a) {
    System.out.println("x = " + x);
    … // code to execute
    assert conditions_to_assert( Extract_Mode(a) );
  }
}
```

The method `Extract_Mode` () in the code would be a static method that returns the current mode set. `conditions_to_assert`() could be any arbitrary statements or expressions that must be asserted, which may be dependent on the current mode set.

• Annotating Prohibited Classes in the Profile

In the profile, certain methods and classes are prohibited for various reasons. To indicate such entities and force the developer not to use them, we propose the following annotation.

```
import java.lang.annotation.*;

@Retention(CLASS)
@Target({FIELD, TYPE})
public @interface RavenscarProhibited {
  public int value();
}
```

Figure 4.24. Definition of the *@ravenscarProhibited* annotation

The specification of the profile in Appendix B is annotated with this annotation appropriately, as shown in the following example.

```
package java.lang;

@RavenscarProhibited
public class Thread implements Runnable
{
  @RavenscarProhibited
  Thread();

  @RavenscarProhibited

  Thread(String name);
  void start();
}
```

4.6. Implementation Issues

Along the same lines as the profile, it is indispensable to utilise a runtime environment that has been designed and implemented with highly dependable systems in mind. However, programs based on our profile should be valid RTSJ programs and execute on a standard RTSJ platform (e.g. [TimeSys2002]) with the same functional results (although perhaps not within their deadlines). In association with this profile, there is an on-going project that modifies Sun's KVM to create a more predictable Ravenscar-Java compliant virtual machine [Cai2003].

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 listed in Appendix A and throughout this chapter. Such a tool may also be able to obtain the Worst-Case Execution Time (WCET) and worst-case memory consumption for each thread, thus enabling schedulability analysis [Bernat2000]. Standard Java tools or model checkers, such as the ESC/Java [Flanagan2002, Leino2000] and Java Pathfinder 2 [JPF2005, Brat2000a-b] can also be used.

4.7. Summary

In this chapter, 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 software. 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 extensions and implementation issues are discussed. The single nested scoping approach is extremely useful in dynamic systems where memory allocations are frequent. Yet, it avoids run-time overheads by eliminating the need for checking the single parent rule, and other reference consistency checks.

The 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 chapter, would be a powerful motivation to develop high integrity systems in Java, rather than in a subset of C, C++ or Ada. Having presented the profile, we shall now move on to discuss useful analysis techniques.

Chapter 5. Static Analysis of Ravenscar-Java Programs

The inherent complexity in the verification of non-trivial software means that unsafe programs could be produced and used under critical situations. This is more so as today's programming models become more complex. The Ravenscar-Java profile has been developed with such concerns in mind, and one of the main aims is to make analysis more straightforward and reliable. Without a language's complex (and often unpredictable) semantics, an analysis tool needs to examine fewer combinations of features, so that the tool itself can be built more reliable. In other words, it is not only the target application software and the run-time platform that become more dependable, but also the supporting analysis tools/techniques in which we gain more confidence. As long as a program passes the conformance test, analysis tools no longer have to consider, for instance, the garbage collector, dynamic class loading, asynchronous transfer of control (ATC), object locks and *synchronize* blocks or statements. Unsurprisingly, however, the language's full expressive power becomes limited at the cost of analysability and predictability, which can be well justified in many high integrity systems.

In this chapter, a number of important analysis techniques in the context of real-time and high integrity systems will be discussed. Although many of the existing techniques in literature are applicable, it is impossible to examine them all for the sake of space. Only those that challenge Java's applicability to high integrity systems will be considered, mainly in the area of predictable memory management which has an impact on timing analysis. We start with some strategies for the conformance test by investigating various ways to check the essential rules and guidelines of the Ravenscar-Java profile. Then, those related to memory safety will be considered. Other issues, including timing analysis, will also be taken into account briefly at the end.

5.1. Conformance test

Before any analysis is performed, it is natural to check whether the subject program conforms to the profile. Only when it passes this test, can we proceed to further analyses; this always has to be the first step in analysing every Ravenscar-Java program.

In general, by statically analysing a program, we obtain a high degree of assurance that a program will behave according to its functional and temporal specifications, and that it will not exhibit erroneous actions throughout its lifetime. Erroneous actions include data races, deadlocks, and memory overflows. For Ravenscar-Java programs, checking for erroneous actions in the presence of the rules becomes a much easier job since common sources of such defects are eliminated at compilation time. The rules can be checked statically when programs are compiled and tested for conformance. Effectively, a conformance test alone will remove many possible errors in the program.

Conformance test may only be performed once the main application class has been defined. This forces all of the schedulable objects and every object they access to be identified. When needed, call graphs from the main thread can then be generated for each thread in a hierarchical way. The way one can enforce the rules is that, for example, if two or more schedulable objects attempt to enter the same memory area concurrently, an error will be flagged[1].

With the various rules, a few strategies can be developed when designing a conformance checker. Roughly speaking, there are two categories of rules and guidelines in the profile, i.e., *syntactical* and *contextual*. Those that fall into the syntactical category can only be checked at the source code level. A front-end of a Java compiler or a syntax checker should be used here, examining potential syntactical errors or misuses of the language according to the profile. The aim of the rules in this category is at reducing sources of syntactical errors and analysis complexity, while increasing readability, analysability and maintainability of the source code. Appendix A.3. *Programming in the small* contains most of the rules and guidelines in this category (given with appropriate rationales). However, some of the rules are general programming guidelines, and could be difficult to check mechanically. There are a number of Eclipse *plug-in*s [Eclipse] that can be customised to check such rules, for example, *CheckStyle*[2] and *FindBug*[3][Checkstyle, FindBugs].

Contextual rules, whose goal is to remove semantic or contextual misuses, are more demanding to check; often specific to the profile, these require the entire hierarchies of classes that an application uses, and a list of threads as well as their

[1] The profile requires that no two schedulable objects attempt to enter the same memory area at the same time.
[2] http://eclipse-cs.sourceforge.net/ or http://sourceforge.net/projects/eclipse-cs
[3] http://www-128.ibm.com/developerworks/java/library/j-findbug1/

memory areas. These checks are performed on either the *bytecode* or the source code. The *Initializer* class of the application is a starting point where all the threads, event handlers, and memory areas are created. From its *run*() method, the checker will first build a list of *Schedulables* and memory areas, and examine that there is exactly one to one matching between them. One must also check that dynamic class loading is not performed in any part of the *Schedulables*, and memory areas and *Schedulable* objects are not created after the initialization phase. This can be done, as illustrated in Figure 5.1, by investigating all instantiated objects in each *Schedulable* to see if they are of the prohibited APIs or their subclasses (see below for such classes and interfaces).

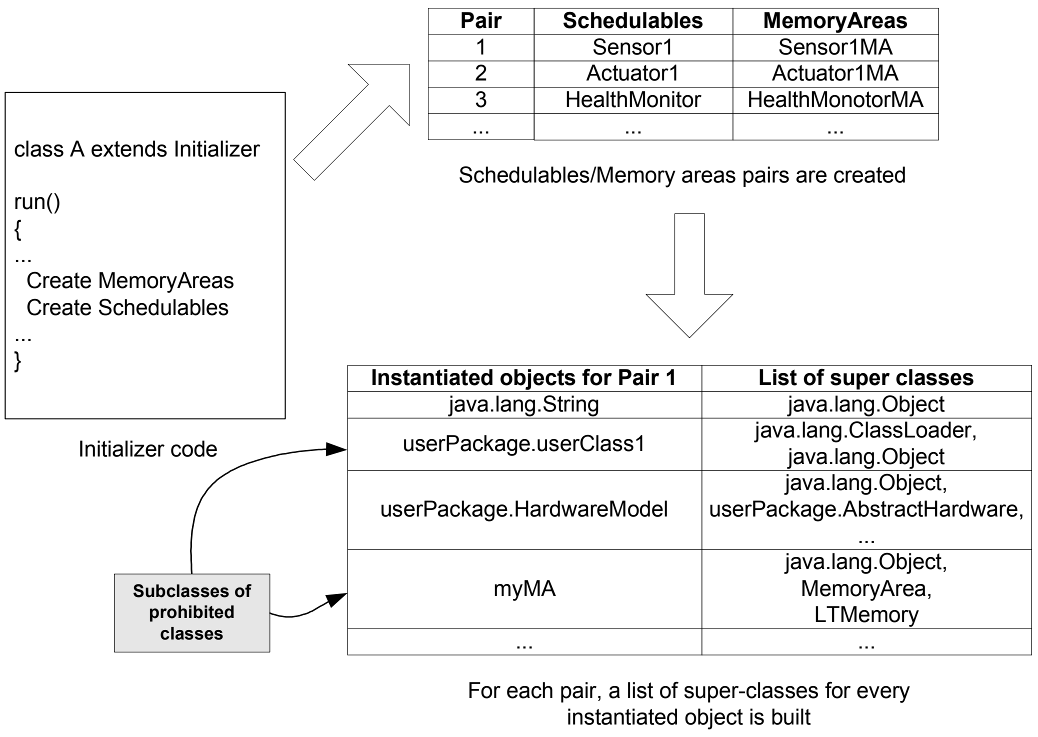

class A extends Initializer
run()
{
...
Create MemoryAreas
Create Schedulables
...
}
Initializer code
Pair | Schedulables | MemoryAreas
1 | Sensor1 | Sensor1MA
2 | Actuator1 | Actuator1MA
3 | HealthMonitor | HealthMonotorMA
... | ... | ...
Schedulables/Memory areas pairs are created
Instantiated objects for Pair 1 | List of super classes
java.lang.String | java.lang.Object
userPackage.userClass1 | java.lang.ClassLoader, java.lang.Object
userPackage.HardwareModel | java.lang.Object, userPackage.AbstractHardware, ...
myMA | java.lang.Object, MemoryArea, LTMemory
... | ...
Subclasses of prohibited classes
For each pair, a list of super-classes for every instantiated object is built

Figure 5.1. An illustration of *Schedulables*' class hierarchy checks

It is also required to check whether or not memory area objects are referenced by more than one *Schedulable*. This is done in order to avoid illegal sharing of objects between threads and reduce the complexity of checking assignments between memory areas. However, if the proposed extension to the *PeriodicThread* class is used, which was introduced at the end of Chapter 4, it is impossible for others to gain access to a memory area object of any thread, meaning that such an analysis is unnecessary while making better use of memory at run-time.

Rules that must be checked specifically in this category are listed below, classified into five different areas. The checker (or a possible combination of tools in reality) should be engineered to deal with all the rules, but may be dependent on external analysis methods for a few reasons, for example, optimised analysis techniques already exist or can be used as a basis for our analysis, or some of the rules are considered out of scope.

Prohibited classes and methods

In the code for the mission phase, the following classes and interfaces must be searched for in every object instance's class hierarchy because they are the potential class loaders, threads, or memory areas. If they are found, then appropriate warnings are to be generated by the checker. Note that *java.lang.Runnable* is allowed because its instance *per se* cannot become a new thread.

Packages	Classes and Interfaces (including specific methods)
java.lang	*Thread, ClassLoader, Class.forName(),*
javax.realtime	*Schedulable, SchedulingParameters, ReleaseParameters, MemoryArea, AsyncEvent, AsyncEventHandler, Interruptible, AsynchronouslyInterruptedException, ProcessingGroupParameters, GarbageCollector*

Figure 5.2. Prohibited classes and methods

In the initialization phase, *Schedulables* must only be instances of *PeriodicThread* (or *NoHeapRealtimeThread*) for periodic activities, and *SporadicEventHandler* (or *BoundAsyncEventHandler*) for sporadic activities.

Restricted classes and methods in the Mission Phase

As shown in the previous chapter, some of the existing Java classes are redefined to be more restrictive. Refer to Chapter 4 and Appendix B for a complete list of all the redefined APIs. Some of the most restricted classes are

- *javax.realtime.MemoryArea* – methods that modify the size of a memory area are removed.
- *java.lang.Thread* – nearly all methods are deprecated except the constructors and *start*().

Rules that need composite analyses

The rules below are considered part of the conformance test since they are so essential to the successful operation of any Ravenscar-Java program. Some of them can be checked with external tools, or efficient analyses can be implemented using techniques from the literature, e.g., [Blanchet2003, Choi1999, Cytron1991, Demartini1998]. The profile's computational model unquestionably helps develop efficient and reliable analysis techniques. Note that most of the rules are related to the use of memory; these are the topics of the following sections in this chapter.

Areas	**Rules/Guidelines**	**Remarks**
Legitimate Use of Memory Areas	*A.2. Rule 5.* Access to memory areas must not be nested	Reduce assignment checks complexity, and remove single parent rule checks.
	A.2. Rule 6. Memory areas must not be shared between *Schedulable* objects	A.2. Rule 7 ensures that all memory areas are created in the initialization phase. One only needs to investigate the *Initializer* thread.
	A.2. Rule 1, 2, 5, and 6 Proper use of immortal memory	Immortal memory is used for objects whose lifetimes are permanent. Objects in that memory must be created only in the initialization phase.
	A.2. Rule 11 and A.3. Rule 9 Identification of shared objects and race conditions	Implementation of the priority ceiling emulation or priority inheritance protocol required.
Exception Handling	*A.1. Guideline 3 to 6, and 9* Appropriate handling of *Exceptions*	Localised handling of all exceptions.
Recursions	*A.3. Rule 1* Detection of recursions	Call graphs must be inspected.

Figure 5.3. Rules that need composite analyses

By means of annotations, programmers can specify which classes are subject to analysis. Such information will be used as input by tools in order to simplify analysis. For example, when classes in a certain package are bound to be correct and need no further analysis, one can annotate the package with a predefined annotation type recognised by the tool. However, one must develop a means to verify the correctness of annotations. Annotations types are discussed in the previous chapter.

Restrictions to the language semantics

There are several restrictions on the semantics of the language, which relate to the use of the garbage collector, finalizers, synchronized statements or blocks, and object locks. Any form of garbage collection is excluded from the profile. Hence, the conformance test must check whether application programs obtain an instance of the *GarbageCollector* class. The *javax.realtime.RealtimeSystem.currentGC*() method returns an instance of the garbage collector currently in use, so that this method must be in an exclude list of the profile (in fact, such classes and methods are not part of the profile).

While *finalizers* are allowed, they must not invoke blocking operations. This means finalizers will not be dependent on objects and locks inside and outside their own objects. In addition, all operations on the object queue, i.e., *wait*(), *notify*(), and *notifyAll*(), are eliminated, such that potential deadlocks caused by object queues are cast out. Synchronized statements are also not to be used because they can lock any arbitrary objects, which can cause deadlocks as well. All of these are checkable by locating invocations of the subject methods and statements in the program.

Programming Guidelines

General guidelines are difficult to check mechanically, for example, whether exceptions are documented with comments, and whether methods are written defensively. Hence, such guidelines should be manually checked by inspecting source code.

In the five categories listed above, strategies for checking the rules are organised and discussed. Except the last one concerning general guidelines, most (if not all) can be mechanically checked with an analysis tool. These days, many IDEs (Integrated Development Environments), such as Eclipse [Eclipse] and its plug-ins, e.g., [Checkstyle, FindBugs], are powerful enough to check use of forbidden classes and methods, and can generate call graphs. Call graphs are useful in detecting recursions, illegal access to memory areas, and illegal object sharing.

Along the lines of the conformance test, a number of useful analyses are applicable on Ravenscar-Java programs. Due to the controlled computational model and restrictions of the profile, analysis algorithms can be made more efficient, as outlined in the subsequent sections.

5.2. Memory Usage Analysis

Overflow or shortage of memory space at run-time can be devastating in high integrity systems, but at the same time, oversupply of it will be costly especially when the target application is mass-produced. Considering the new memory areas introduced in the RTSJ, we need a different means of estimating the worst-case memory space that a program requires at run-time, so that only required amount of memory for each area will be allocated.

In the original Ravenscar-Java's memory model, one scoped memory area is attached to each thread. The total cumulative amount of memory that will need at runtime is determined by adding up the sizes of all scoped memory areas and the immortal memory. Unlike other resource types, even when the memory areas are not used by their threads (either when preempted or not released), they still take up the physical memory once created in the initialization phase. This is so because in a typical implementation of *MemoryArea*'s constructors, a required amount physical memory is first reserved by calling a type of *calloc*()[4] functions in C, thus creating a memory pool. After that, each `new` operation in the thread's run method invokes low-level functions, mainly *exchange_and_add*(), to allocate real Java objects in the memory pool, represented by a memory area object at Java-level.

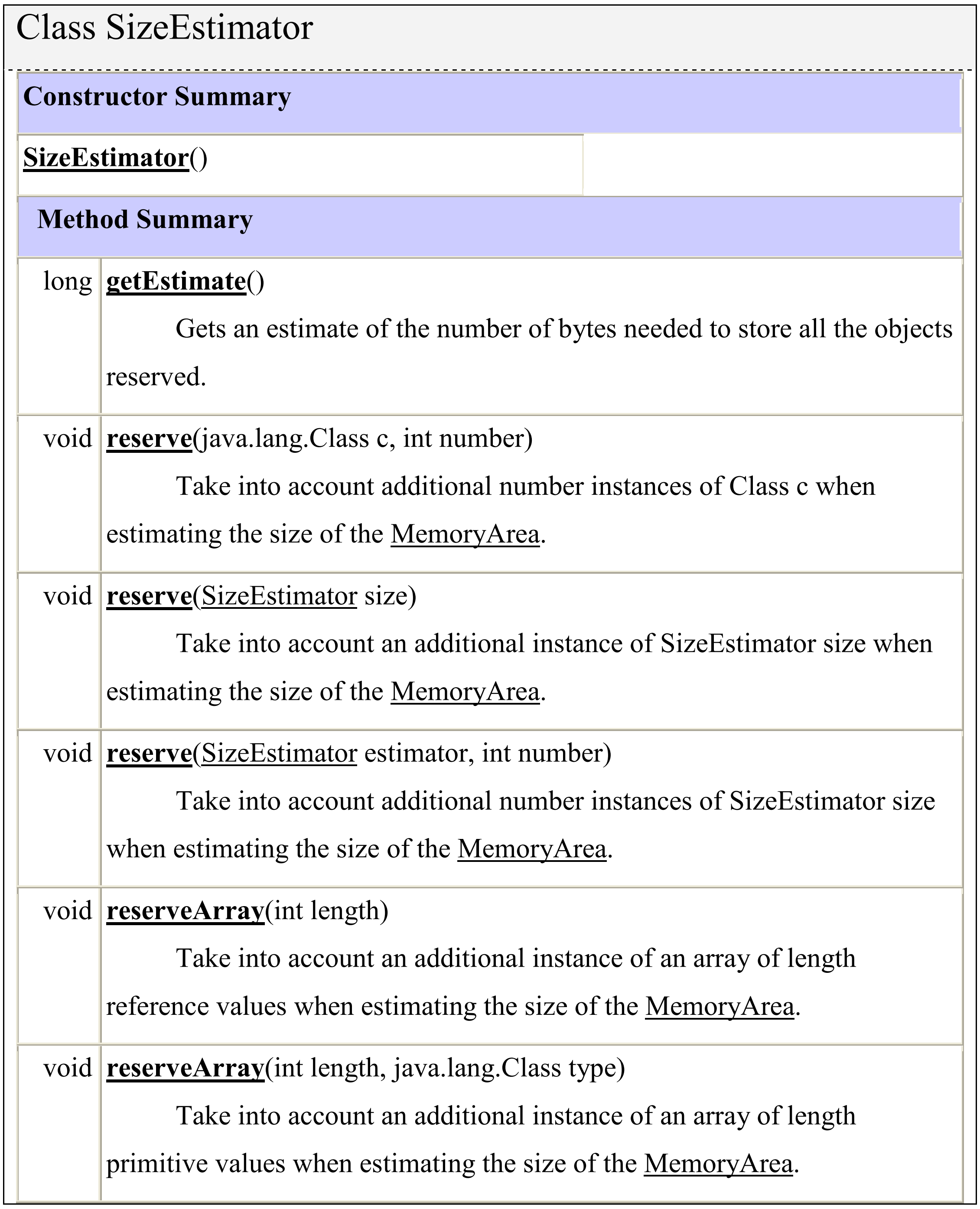
In the object allocation process described above, one can deduce an object's size in reality. However, there can be a misconception here; from a user's view point, two objects that share the same parent class may well have different *collective* sizes at runtime. This is so, since objects that are created and referenced inside an enclosing object can be different even though they are created and allocated in the same area. Such objects are hidden from outside (a user's perspective) and allocated separately from the enclosing object. In other words, when estimating an object's size, it is incorrect to assume that an object's allocated size would suggest the amount of memory it will use at runtime. In actual fact, the *thread-wide* amount of memory that an object will take up at runtime must be calculated as a cumulative sum of its contained objects' sizes required dynamically. Each object's size is determined by adding up the sizes of the following attributes;

- *Primitive type fields,*
- *Reference to the class block, and*
- *References to all contained objects.*

[4] There are subtle differences between calloc() and malloc(); the former assigns required memory at once in a linear space while the latter may not.

The size of a class, including its constant pool and method areas, is not included here because all classes are loaded into immortal memory (or the heap if present) in the initialization phase. They will not affect the size of a scoped memory area in Ravenscar-Java.

For the purpose of estimating an object's size, the RTSJ defines the *SizeEstimator* class, as shown in the figure below, meaning that one does not have to instrument the object allocation code of the virtual machine. However, as mentioned above, the *getEstimate*() method of the class does not return the actual amount of memory that an object of a class and its methods dynamically use, but simply the total size of the class's fields and references. In this sense, the *SizeEstimator* class alone is not readily usable in calculating the worst case amount of memory required for each memory area. Hence, there is a need for an analysis that investigates flow information of Java code and locates objects that are created dynamically.



Class SizeEstimator

Constructor Summary	
SizeEstimator()	

Method Summary	
long	**getEstimate**() Gets an estimate of the number of bytes needed to store all the objects reserved.
void	**reserve**(java.lang.Class c, int number) Take into account additional number instances of Class c when estimating the size of the MemoryArea.
void	**reserve**(SizeEstimator size) Take into account an additional instance of SizeEstimator size when estimating the size of the MemoryArea.
void	**reserve**(SizeEstimator estimator, int number) Take into account additional number instances of SizeEstimator size when estimating the size of the MemoryArea.
void	**reserveArray**(int length) Take into account an additional instance of an array of length reference values when estimating the size of the MemoryArea.
void	**reserveArray**(int length, java.lang.Class type) Take into account an additional instance of an array of length primitive values when estimating the size of the MemoryArea.

Figure 5.4. SizeEstimator [RTSJ2005, Bollella2000a]

The Ravenscar-Java profile places a number of restrictions on the use of memory areas; for instance, access to linear-time memory (*LTMemory*) areas must not be nested and such memory areas cannot be shared between *Schedulable* objects. These restrictions facilitate the development of an algorithm that will inspect each thread's logic to discover all classes it instantiates. By making use of control and data flow information extracted from the code (including maximum loop bounds – annotated by the programmer), the algorithm will be able to calculate how many instances of each class are created by a thread on its dedicated memory area. This information can then be used to produce a tight upper bound of the amount of memory that a thread utilizes at run-time by applying *reserve*() and *getEstimate*() methods of the *SizeEstimator* class at the target platform.

This idea is illustrated in Figure 5.5 below. Note that the maximum loop bounds are annotated for each loop, and objects created inside *callee* methods are also added to a list called *Instantiated Objects Table* (IOT) that keeps track of every object created. We need to consider constructors as well, which are invoked implicitly when a new object is created and additional objects can be created inside them. The number of times that an object is created is also inserted in the IOT node. These values will be used to calculate the exact figure of the whole memory consumption later.

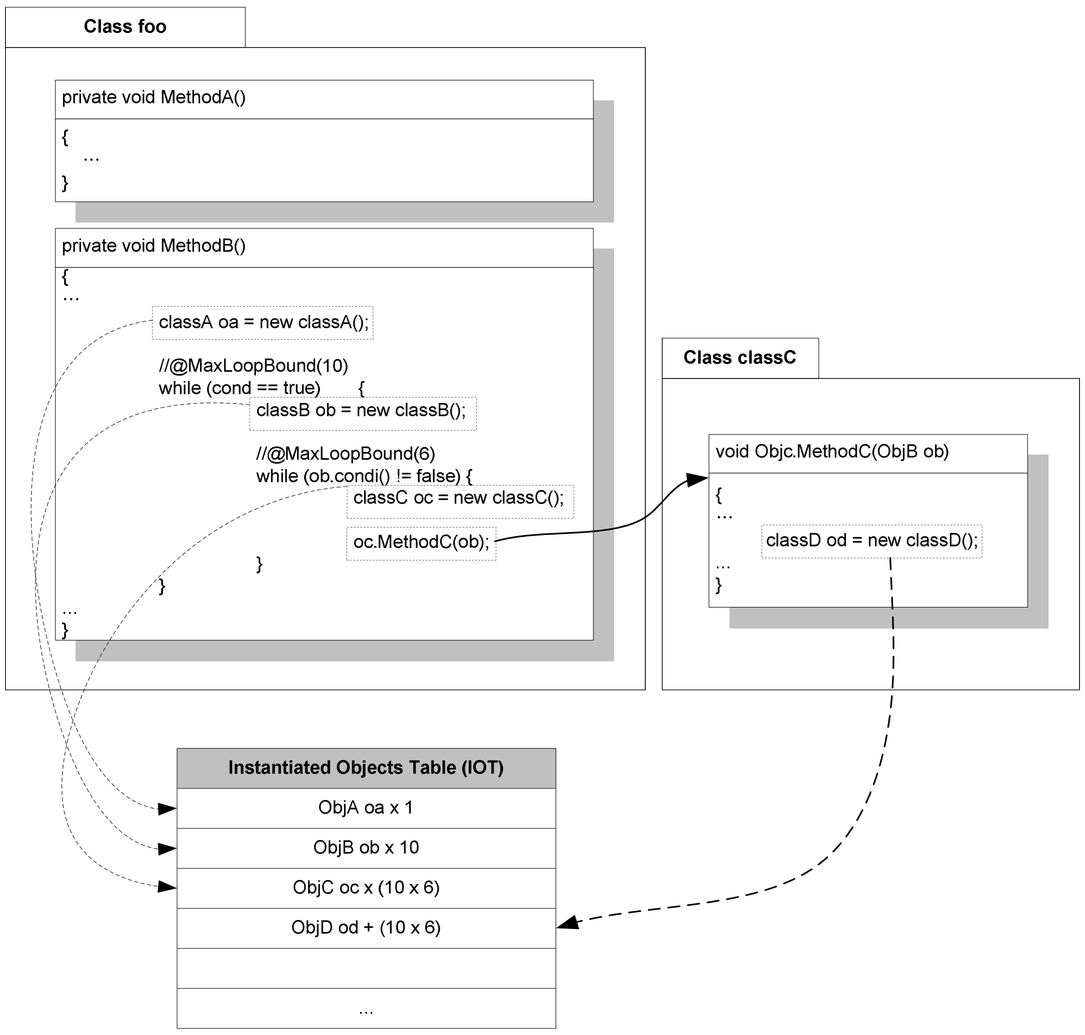


Figure 5.5. Illustration of the Worst Case Memory Consumption analysis

In the example above, class *foo* has two methods. *MethodB* is elaborated in detail to show how new objects created inside a method and the way such instances are documented in an IOT node. A thread-wide analysis has to be carried out by inspecting a thread's execution paths, since different threads may invoke different combinations of methods.

When a thread's logic contains conditional branches, the analysis has to iterate more than once (or *backtrack* to the last branching point), such that every possible path is examined. In this case, there needs to be a collection of IOT nodes for each path and one can observe which collection results in the worst case memory usage for a given piece of logic or thread. The figure below illustrates a more elaborate view of how such a collection of IOT node are built. It shows the conditions on which subsequent IOT nodes are created, which match those of the code.

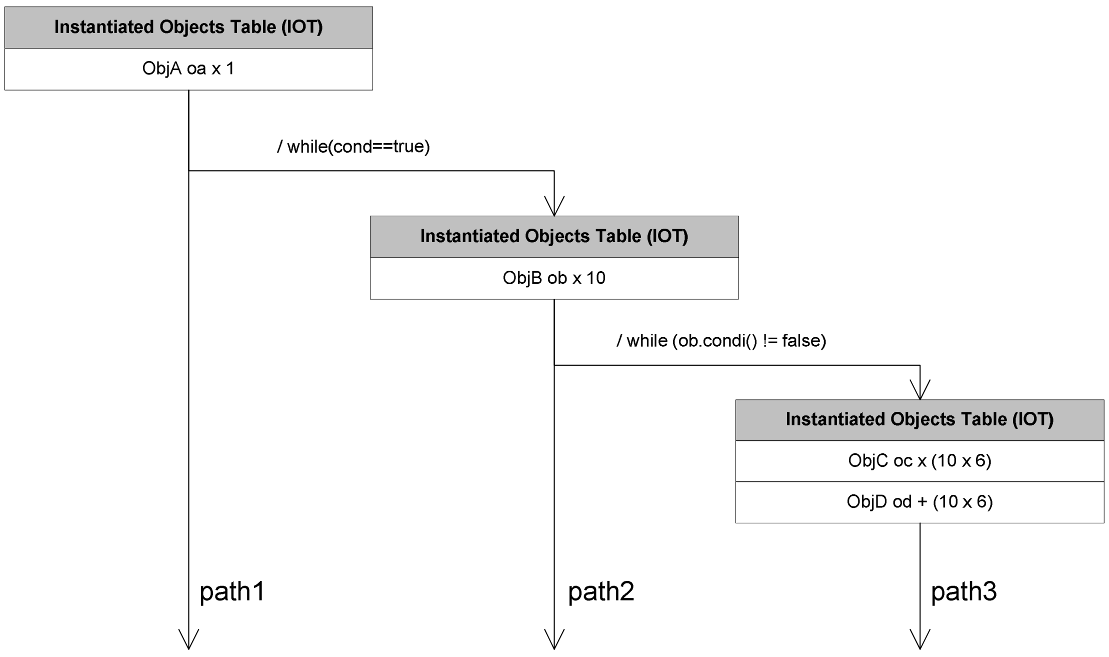

Instantiated Objects Table (IOT)
ObjA oa x 1
/ while(cond==true)
Instantiated Objects Table (IOT)
ObjB ob x 10
/ while (ob.condi() != false)
Instantiated Objects Table (IOT)
ObjC oc x (10 x 6)
ObjD od + (10 x 6)
path1
path2
path3

Figure 5.6. Illustration of IOT paths of *MethodB*

Among the execution paths created for *MethodB* above, *path3* is the worst case since it creates most objects. This is a simple case because there is no new objects in *else* parts of the branches. If, however, there are objects created in else parts, calculation becomes rather complex, as shown in the figure below. One cannot compare the sums of objects' sizes of all paths until the actual size of every object is known. Even if the same class is used to create objects in different paths, we cannot simply assume their sizes would be the same. This is because different constructors and methods may be invoked on different execution paths.

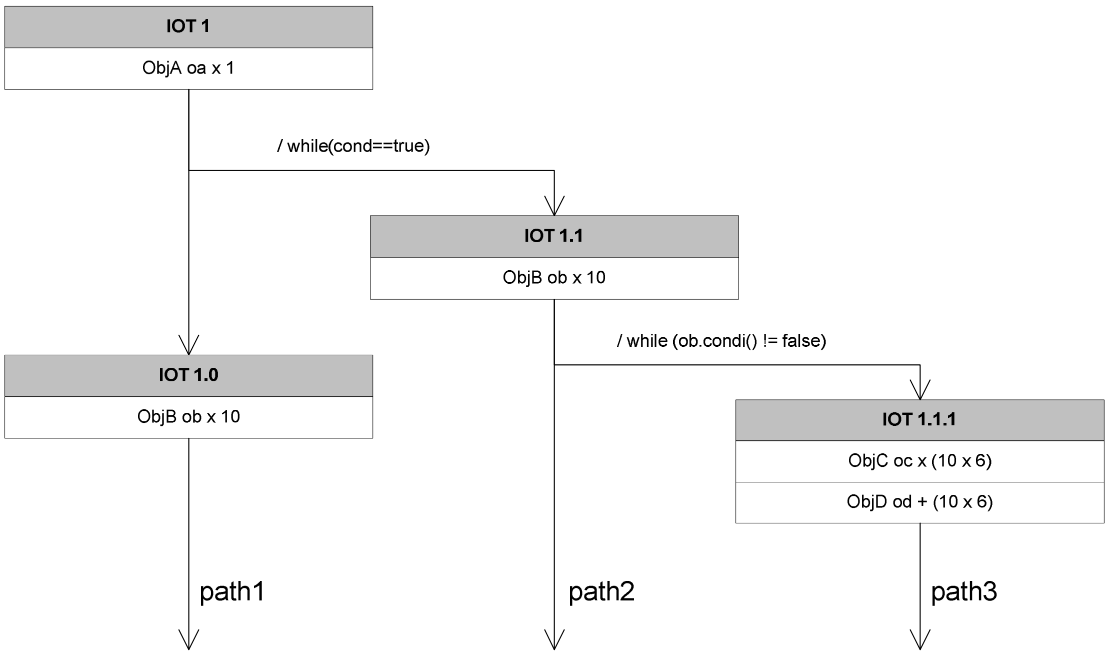


Figure 5.7. Illustration of IOT paths with conditional branches

Notice the way each IOT node is numbered from now. The parent node is numbered as 1 and, depending on subsequent conditional branches, each child IOT node is numbered as 1.0 or 1.1. Simply put, the ones with 0 at the end signify IOT nodes that are created when the previous branching condition is false, and *vice versa*. As mentioned above, even though IOT 1.0 contains the same object as that of IOT 1.1, one should not conclude both will take up the same amount of memory. Each object in IOT nodes must be expanded because they can create objects in their constructors and subsequent method calls (in fact, flow analysis will take care of this, in order to reveal hidden objects not shown above in each path).

Once we have obtained all newly created objects for every path, we can now calculate the actual memory requirements for each of the paths by adding up the sizes of the objects in the IOT nodes, as shown below. *wm* denotes a function of the worst-case memory usage for an execution *path* while *sizeOf* the actual sum of the sizes of every enclosed objects in each IOT node.

wm(path1) = sizeOf (IOT 1) + sizeOf (IOT 1.0)

wm(path2) = sizeOf (IOT 1) + sizeOf (IOT 1.1)

wm(path3) = sizeOf (IOT 1) + sizeOf (IOT 1.1) + sizeOf (IOT 1.1.1)

In the RTSJ, the above translates to the following (assuming that we have an object *wm* and *long* variables *path1*, *path2* and *path3*).

*wm.path1 = getEstimate(ObjA) * 1 + getEstimate(ObjB) * 10*

*wm.path2 = getEstimate(ObjA) * 1 + getEstimate(ObjB) * 10*

*wm.path3 = getEstimate(ObjA) * 1 + getEstimate(ObjB) * 10 +*
    *getEstimate(ObjC) * (10 * 6) + getEstimate(ObjD) * (10 * 6)*

If any of the objects above enclose other objects and instantiate them in the constructor, or call a method explicitly that creates such objects, then the instantiated objects must be taken into account as well. For instance, let us assume that *ObjB* creates further objects (say *aClass1 and aClass2*) in its constructor. *sizeOf (IOT 1.0)* and *sizeOf (IOT 1.1)* are expanded as follows. Depending on the flow information, IOT 1.1 does not create an instance of *aClass1*.

sizeOf (IOT 1.0) = (getEstimate(ObjB) + getEstimate(aClass1) +
    *getEstimate(aClass2)) * 10*

*sizeOf (IOT 1.1) = (getEstimate(ObjB) + getEstimate(aClass2)) * 10*

Our flow analysis has a depth-first search algorithm that traverses the whole method invocation trees and creates a tree of IOT nodes. Upon reaching an end node, it backtracks to the latest branch/conditional point, and begins searching again for ‘*new*’ instructions on a different path and creates a new set of IOT nodes. It turns out that each IOT node is, in fact, a *basic block* of code that contains at least one or more ‘*new*’ instructions. Those with no *new* instructions are simply discarded.

We use the Soot framework [SOOT] to gain access to *bytecode* in a format that allows us to easily explore Java programs. By being able to access each instruction and blocks of code, we can count the number of new operations and backtrack to the latest branch points. The algorithm’s complexity is linear to the size of the input tree. Input to this analysis should be the name of a thread that the analysis will start from. The figure below shows a *baf* output of the class *foo* and MethodB.

```
public class foo extends java.lang.Object
{
     private void MethodB() {
        word this, oa;
        this := @this: foo;

     label0:
        new classA;
     label1:
        dup1.r;
     label2:
        specialinvoke <classA: void <init>()>;
     label3:
        store.r oa;
     label4:
        load.r this;
     label5:
        fieldget <foo: boolean cond>;
     label6:
        push 1;
     label7:
        ifcmpne.b label21;
     label8:
        new classB;
     label9:
        dup1.r;
     label10:
        specialinvoke <classB: void <init>()>;
     label11:
        store.r oa;
     label12:
        load.r oa;
     label13:
        virtualinvoke <classB: boolean condi()>;
     label14:
        ifeq label20;
     label15:
        new classC;
     label16:
        dup1.r;
     label17:
        specialinvoke <classC: void <init>()>;
     label18:
        load.r oa;
     label19:
        virtualinvoke <classC: void MethodC(classB)>;
        goto label12;
     label20:
        goto label4;
     label21:
        return;
    }
...
}
```

Figure 5.8. *Baf* output of class foo

5.3. Java Memory Model and Shared Objects Analysis

As reported in [Pugh1999, Manson2005, Pugh2005] and [Roychoudhury2002], the original Java memory model (JMM) in [Gosling2000] is a weaker model of execution than ones supporting *sequential consistency*. It allows more behaviours than ordered interleaving of the operations of the individual threads. In other words, the JMM permits compilers to optimize heavily and reorder code around[5], so that when run in parallel with another thread, unexpected and incorrect results can be seen (see Figure 5.9. below for an example). Therefore, verification tools that simply examine Java source code or even bytecode are prone to produce false results [Roychoudhury2002]. Because the semantics of the JMM can lead to different implementations, some virtual machines may support sequential consistency, while others may not for performance reasons. This does not match the Java's *write once, run everywhere*[6] philosophy.

Initially, x == y == 0	
Thread1	**Thread2**
1: r2 = x;	3: r1 = y;
2: y = 1;	4: x = 2;
r2 == 2, r1 == 1 violates sequential consistency	

Figure 5.9. Unexpected results of the original JMM [Manson 2005]

Recently, in the release of Java 5, a new memory model that challenges the problem has been incorporated. While it still allows certain compile time optimizations, it is strict enough that, even when optimized, programs will not exhibit the same unintended behaviours shown above. However, many resource-constrained embedded

[5] Typical optimisations including the Common Expression Elimination or Redundant Read Elimination.
[6] Programs may still run everywhere, but possibly with incorrect or unsafe behaviours.

virtual machines, such as Sun's CDC VM [Sun2006], do not implement the new memory model introduced in Java 5. Plus, data races can still occur.

The essential problem, however, can be resolved by properly *synchronising* such blocks of code. Synchronization in Java will flush all the data in registers and caches, and compiler optimizations will not move code in or out of synchronized blocks of code (i.e., *synchronized methods* in the profile). Although this comes at a cost of lower performance (depending on how often synchronized methods are used), it is always better to be safe in high integrity systems, rather than having to rely on an underlying memory model. In fact, if a static analysis can trace all shared objects in a program, it will be able to discover only those absolutely necessary synchronizations. The program is not only optimized by removing unnecessary or redundant synchronizations, but also becomes safer in terms of sharing objects.

Here, we only need to discover objects that are unintentionally shared between threads. The shared objects analysis is in two parts: 1) tracing shared objects as well as ensuring they are created and reside in the immortal memory only, and 2) checking whether they are properly synchronized. In the first part, one must make sure shared objects are created in the initialization phase only, because in the mission phase one cannot allocate a new object in the immortal memory. This means any objects created outside immortal memory must not be shared or referenced directly. In reality, we found that *illegally* shared objects can be identified by assignment checks. Failing the assignment checks means that one tries to assign a reference to an object in one memory area to another area that does not belong to the calling thread (or, it is also possible that an object in a short-term memory area is referenced by one in a long-term memory area). Hence, any reference relationships that cross beyond thread boundaries are illegal and unintentionally attempt to share objects, as illustrated below in Figure 5.10. Moreover, memory area objects themselves must be protected from illegal threads. Remember that the profile does not allow threads to obtain references to other memory areas than its own.

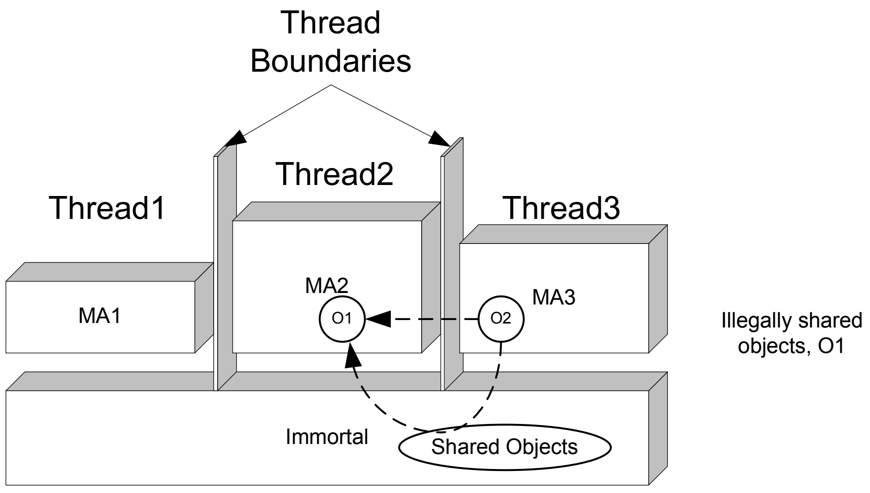


Figure 5.10. Illegally shared objects can be identified by assignment checks

Having located all legally (and illegally) shared objects, the second part checks whether the shared objects are properly synchronized. The analysis must ensure that every *write* access to the identified shared objects is synchronized.

The underlying assumption of this approach is that any reads and writes on a shared object in a method must be enclosed within the same synchronized block (or method in the profile) in order not to have any data races[7]. In other words, any syntactical gap between a read and write that are not covered by a single synchronized block will cause possible data races in a multithreaded environment because either a read or write action can be lost. This is true even when a shared object is indirectly read and updated using a local object because, for example, an interleaving of another thread that may update the shared object can occur in between the indirect read and a

[7] Essentially, data races are the most obvious outcome that the Java memory model could have on any multithreaded Java programs.

(possibly synchronized) write in the method, resulting in a lost write. Thus, any indirect reads and writes should also be treated in a similar manner to direct ones on a shared object. The figure below shows an example case (see the *increaseBy1* method).

```
Shared variable: int Svar
```

```
void increaseBy1 (int val){
  int tmp1;
  int tmp2;

  tmp1 = Svar;
  tmp2 = tmp1 + val;
         //indirect read

  sync (Svar, tmp2);
}

synchronized void sync(int tmp2){
    Svar = tmp2;
    // synchronized write
}
```

```
void increaseBy2 (int val){
  int tmp;
  tmp = sync1();
  sync2(tmp, val);
}

synchronized int sync1(){
    return Svar;
    // synchronized read
}

synchronized void sync2
             (int tmp, int val){
    Svar = tmp + val;
    // synchronized write
}
```

Figure 5.11. Two methods illustrating possible data races

Another similar case is that even when both a read and write are synchronized, there still can be data races if the two blocks are guarded by two different synchronized methods and can be interleaved by other threads in between (see the *increaseBy2* method in the figure). In essence, independent writes that are preceded by a read may potentially generate a condition where data races can occur, which thus should be warned.

An algorithm can be developed that is capable of analysing all such conditions above, thus detecting problematic data races by tracing all shared objects and checking whether they are properly guarded by synchronized methods. As an illustration, the following algorithm in pseudo code has been devised.

```
For each shared variable SVi used by thread T
(in case SVi is a compound object, for each member of the shared
variable)

while (T's thread logic != EOF)
{
  curr_instruction = getNextInstruction(); // go through instructions
  if (curr_instruction == ENTER_MONITOR_METHOD) {
    in_sync_method = true;
    continue;
  } else (curr_instruction == EXIT_MONITOR_METHOD)
  {
    in_sync_method = false;
    if (value_read == true)
    {
      exiting_sync_method_value_read = true;
    } else
    {
      exiting_sync_method_value_read = false;
    }
    continue;
  }

  if (SVi is read || SVi is written) {
    if (in_sync_method == true) {
      if (SVi is read) {
        value_read = true;
        exiting_sync_method_value_read = false;
      } else if (SVi is written) {
        value_written = true;
        value_read = false;
      }
      if (shared_without_sync == true && value_written == true) {
        Raise a warning! // Read&write not in the same sync
      }
      if (exiting_sync_method_value_read && value_written == true) {
        Raise a warning! // Read&write synchronized seperately
      }
  } else {
    if (SVi is read) {
      shared_without_sync = true;
    } else if (SVi is written) {
      if (shared_without_sync == true) {
        Raise a warning! // Used without sync - write
      } else if (value_read == true) {
        Raise a warning! // Read was synchronized, but write is not.
      }
    }
  }
}
```

The Soot framework [SOOT] provides a means to access each instruction in bytecode, and one can identify threads and memory areas. Thus, it is possible to test if objects in *Instantiated Objects Table* (IOT) nodes introduced earlier, are accessed across thread boundaries. The point-to and escape analysis [Choi1999, Salcianu2001, Blanchet2003] can also be used to trace escaping and possibly shared objects, as well as improving overall performance by allocating non-escaping objects in the stack of a method. The Ravenscar-Java profile simplifies this task, since such algorithms will not have to deal with many of the language's complex features, such as the object queue, complex constraints in conditional statements, and synchronized statements. Having passed the conformance test, all Ravenscar-Java programs create shared objects only in the immortal memory and only in the initialization phase.

5.4. Memory Area Access Analysis and Optimisation

This analysis is concerned with eliminating unpredictable runtime overheads caused by dynamic assignment checks. Because of the characteristics of the memory areas defined in the RTSJ, every assignment expression is subject to a dynamic check at runtime, which will determine the legality of such expressions. In other words, an object in a long-term (or parent) memory area cannot not reference an object in a short-term (child) one; this will create a dangling reference when the latter area ceases to exist.

However, such costly checks can be prevented by means of the analysis introduced above, or a pointer and escape analysis [Choi1999, Salcianu2001, Blanchet2003]. Objects that escape from their original memory areas can first be identified using an escape analysis technique, followed by a simple means to resolve which *direction* the object is escaping in the memory area stack of a particular thread (see Figure 5.12 below). If the escaping object is referenced by another object in a longer-term memory area, then the assignment at run-time will fail and an exception will be raised. A cooperating virtual machine will not need such checks.

The shared objects analysis introduced earlier can also perform a similar task. Like the escape analysis, it can identify shared objects, whose scopes are beyond the boundary of a thread and a memory area. Any object that is shared by more than one thread through a scoped memory area means that such an object can be illegally accessed by others. If, however, an object is accessed by objects in more than one memory area in the same thread boundary, then again, one must investigate the direction of such references in the memory area stack. If object references are stored in a longer-term memory area, then assignment errors will be raised.

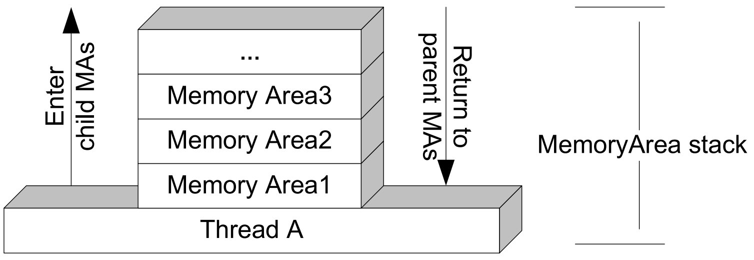


Figure 5.12. Memory Area Stack in the RTSJ

Unlike the RTSJ, however, the profile allows only one scoped memory area for each thread and the immortal memory that shared and permanent objects are allocated. Hence, there are only two memory areas to consider for a thread, and as a bonus, the Single Parent Rule [RTSJ2005] needs not be checked at all[8]. The only way that a reference is passed to a parent memory area is through parameters of a call to an object inside the immortal memory. Assuming that all methods of each immortal object are identified in the previous analysis (see section 5.2 and 5.3), a tool can check

[8] If one chooses to use the Single Nested Scoping feature introduced in Chapter 4, all scoped memory areas are hidden from user code and they are out of concern. This also means the Single Parent Rule still needs not be checked.

whether or not such methods take object references as parameter and store them in a local object. Methods that do not *store* any parameters are considered *assignment safe*, and may be annotated with this information. If all of the methods are proven to be assignment safe, then dynamic checks are no longer required for the enclosing class (thus should be annotated so).

Another check that must be performed is whether or not scoped memory areas are shared between threads. Any means of gaining access to other threads' memory areas is prohibited in the profile. This check is in fact part of the conformance test, in which a list of memory area-to-thread matches is created. If a reference to a memory area object is passed to more than one thread, the program must be flagged illegal.

5.5. The Profile's Impacts on Real-Time Scheduling

The profile's computational model is based on fixed-priority pre-emptive scheduling [Audsley1993, Burns2001]. A thread with the highest priority will always be executed whenever it becomes ready. Only two types of threads are supported; periodic threads (*PeriodicThread* or *NoHeapRealtimeThread* class) for periodic activities, and sporadic event handlers (*BoundAsyncEventHandler* or *SporadicEventHandler* class) for sporadic events with minimum inter-arrival times. It has been well-studied in the literature how to determine the worst-case response times of the two types of scheduling entities, given the worst-case computation & blocking times of them [Burns01, Audsley1993, Liu1973].

Calculating the worst case execution time (WCET) of a thread is challenging with the RTSJ. There are a number of factors that affect this analysis, such as, dynamic class loading, unlimited possibility of creating threads, operations on memory areas, asynchronous event handlers, and asynchronous transfer of control

(ATC). All of these concerns are cast out in the profile, so that well-ordered execution of threads is guaranteed. Only when shared objects are involved, subject threads will be dependent on each other, and the Priority Ceiling Emulation should be used to counteract the effect of unlimited priority inversions.

WCET analysis tools require loops to be annotated with their maximum bounds. As with the memory consumption analysis discussed early, this value will be multiplied by the execution time of such loops. Given more than one paths in the program, the one that results in the longest execution path will always be selected for further analysis. Once WCET values are known, blocking times are also obtainable using analysis techniques in [Burns2001, Audsley1993]. However, other scheduling overheads or *jitters* can be hard to attain by application-level analysis. Such values must be provided by platform vendors.

5.6. Summary

Our profile greatly simplifies program analysis. We have given evidence of potential use of analysis techniques that can benefit from the profile. Because of the simpler computational model and less error-prone features, such analysis techniques can be designed and developed more reliably. Most of them need some form of data- or control-flow analysis, which can be performed on Ravenscar-Java programs that do not contain intricate features. Especially, the Instantiated Objects Table (IOT) developed in this chapter is useful when calculating the worst case memory usage as well as identifying shared objects.

Chapter 6. Evaluation

In this chapter, the profile presented so far will be evaluated in terms of its effectiveness in producing predictable high integrity software. We will first investigate how well the profile fits in the framework of criteria developed in Chapter 3. Next, the expressive power of the profile will be illustrated and evaluated, by means of a realistic case study, an implementation of European Space Agency's Packet Utilization Standard.

6.1. Adherence to the criteria

For the purpose of evaluation, a high integrity language should be approached from at least two different viewpoints: *how well a language satisfies the standards and guidelines* (appropriateness), and *how expressive it is* (expressive power or expressiveness). We already have a well-defined set of criteria derived from various standards and guidelines (see Chapter 3), and a language that does not conform to them is unlikely to be accepted in certification processes. At the same time, a language with a limited set of features that do not reflect on technological advances in software engineering is also not what is desired in industry. The key here is a balance between *useful restrictions* and *expressiveness*.

In this section, an investigation of how well the Ravenscar-Java profile blends in the framework of criteria is carried out. As mentioned early, the framework was developed by reviewing influential high integrity standards and guidelines, so that it echoes most of the requirements on high integrity languages. Expressive power is the topic of the next section.

6.1.1. Assessment against Level 1 Mandatory Requirements

Note that how the original ratings given in Chapter 3 for original Java plus RTSJ have changed. In most of the areas improvements have been made (marked as 'improved' in each subject criterion). A summary will be given at the end.

L1.1. Syntactical / Semantic Requirements

L1.1.1. Type safety / Strong typing rules

As with the original rating given in Chapter 3. However, static type analysis is possible since in the profile

- dynamic class loading is avoided (A1.1. Rule 1),
- dynamic method bindings are minimised (A1.1. Guideline 2), and
- dynamic method bindings are identified with comments (A1.4. Guideline 10).

The above rules and guidelines mean that every class that will be required during the lifetime of an application will be discovered and loaded before the Mission phase. Although not immediately visible, one can deduce all the possible set of corresponding types for any arbitrary type with the help of a class hierarchy and data flow analysers. Annotations are useful here to guide such analysers to gather minimum bounds of types for a specific type matching (likely to be assignments or method calls).

Rating: 1. Strongly typed / Statically analysable (Improved).

L1.1.2. Side effects in expressions / Operator precedence levels / Initial values

Side effects are eliminated by most of the rules and guidelines in *A.3 Programming in the small* (see Appendix A.3). All the possible causes of side effects are factored out because

- all constraints must be static (A3.1. Rule 4),
- compound expressions are removed (A3.1. Rule 6),
- expressions whose values are dependent on the order of evaluations are avoided (A3.1. Rule 7),
- parentheses are used rather than relying on the default order of precedence (A3.1. Guideline 1),
- parentheses are used in bitwise operations, comparisons, and conditions (A3.1. Guideline 2 and 3), and so on.

It is also necessary for variable declarations to include a static initialization expression (A3.1. Rule 5).

Rating: 1. All the above specifics are satisfied (Improved).

L1.1.3. Modularity / Structures

As with the original rating given in Chapter 3. The profile does not affect Java's modularity.

Rating: 1. The language provided rich and precise means of structuring programs, and programs can be maintained in terms of modules or objects.

L1.1.4. Formal semantics / International standards

As with the original rating given in Chapter 3. Formal definitions of the profile may be produced in future, but it is not the main focus of this work.

Rating: 2. The language or a high integrity subset of it can be formally defined (Not improved).

L1.1.5. Well-understood

As with the original rating given in Chapter 3. The Ravenscar-Java profile is a subset of Java and RTSJ, which means any Java programmers are accustomed to the syntax and APIs. However, they must be informed about missing features and libraries, so that appropriate design decisions can be made in advance.

Rating: 1. The language is well understood, and there are many trained developers and designers.

L1.1.6. Support for domain specific or embedded applications

As with the original rating given in Chapter 3. The language itself does not support hardware; but this is done by means of physical memory areas in the RTSJ and in the profile.

Rating: 2. There is a limited support, but external libraries or language extensions can be utilised (Not improved).

L1.1.7. Concurrency / Parallel processing

Because the original Java's concurrency model is liberal, complex programs can be created possibly with deadlocks and unbounded priority inversions. The profile limits some of the concurrency features of Java, i.e., object queues and monitors. In addition, applications must be structured in a way where only two types of active execution entities exist, i.e., periodic and sporadic threads. Interactions between threads are performed via shared objects that must be identified and reside in immortal memory. The two execution phases exist in order to ensure well-ordered execution of the whole application, thus improving predictability of memory utilisation as well as timing analysis.

Although not as expressive as the original language, the profile still supports multi-threading and a way of inter-thread communications with bounded blocking.

Rating: 1. All the above specifics are satisfied.

L1.2. Application of verification techniques / Predictability

L1.2.1. Functional predictability

There are many analysis tools and techniques for Java, and they are all applicable to Ravenscar-Java programs. Such tools now need not consider error-prone features, e.g., object queues. Model-checkers will require much less state space because thread interactions are limited only to exclusively accessing shared objects (for example, the thread-*join* and object queue mechanisms are completely ruled out).

The Java's assertion facility will also help in this area as it can be used to check pre- and post-conditions as well as user specified assertions.

Rating: 1. All techniques in the above specifics or feasible alternatives can be utilised (Improved).

L1.2.2. Temporal predictability / Timing analysis

Garbage collection is cast out, and a number of improvements have been made over the original Java and RTSJ in this area. For instance, the two execution phases ensure well-ordered execution of threads, and in the Mission phase only periodic and sporadic activities are allowed. Priorities cannot be changed once set, and classes are not loaded dynamically. All these limitations will contribute towards obtaining tight bounded execution times.

Rating: 1. Tightly bounded execution time(s) can be obtained (Improved).

L1.2.3. Resource usage analysis

One of the prime resources in any application is memory. In the profile, programmers have control over how each memory area is used. All allocated objects are identifiable with analysis (as shown previously in Chapter 5), and for each execution path, a list of created objects can be obtained and used to calculate the worst-case memory bounds. This analysis is only possible once the target platform is known (i.e., one must know how much memory every object will use in reality). Otherwise, only symbolic analysis is achievable.

Rating: 1. Exact prediction of the above specifics is possible (Improved).

L1.3. Language Processors / Run-time environment / Tools

L1.3.1. Certified language translators / Run-time environments

As with the original rating given in Chapter 3.

Rating: 2. Language translators may contain several known errors or malfunctions that are well documented, but they will not affect the development of high integrity software (Not Improved).

L1.3.2. Run-time support / Environment issues

As with the original rating given in Chapter 3. All external software entities must be produced with high integrity applications in mind. To be used in conjunction with Ravenscar-Java programs, such software should also be written in a way that follows the guidelines and rules given here in Chapter 4 and Appendix A. For instance, a library that invokes a class loader will only be used in the Initialization phase and will not obstruct the way memory areas are used[1]. Worst-case behaviours or bounds in terms of timing and memory utilisation must also be known and documented using annotations.

Depending on applications, different ratings can be given in this area. For all libraries and runtime that are written in Ravenscar-Java, Rating 1 can be given, i.e., *There exists concrete information on the functional and temporal behaviours of all libraries and run-time system*, since static functional and timing analyses are applicable. As a minimum, Rating 2 (i.e., *worst-case analysis is possible*) must be obtainable in order to be used in high integrity systems.

Rating: Variable, dependant on applications.

6.1.2. Assessment against Level 2 Desirable Requirements

Criteria in this category are recommendations. As mentioned before, these are not compulsory conditions, but rather favourable ones in that they help produce more efficient, expressive, and structured software.

[1] In a conventional virtual machine, temporary objects may be created during class loading. For memory sensitive applications, class loaders can be improved in a way that a temporary scoped memory area is used. In other cases, classes can be loaded pre-runtime, like in Jamaica VM [Aicas].

L2.1. Syntactical / Semantic Requirements

L2.1.1. Exception Handling / Failure behaviour

While the profile supports Java's exception mechanism, a number of guidelines are given on the treatment of exceptions. All user exceptions must be identified and documented, and must be handled locally, i.e. within a schedulable object's boundary. Any system-level exceptions, e.g., null-point exceptions and array boundary exceptions, must not be thrown at run-time, which means they are analysed statically.

An exception propagation checker can be developed, taking advantage of the profile's unambiguous computational model.

L2.1.2. Model of Mathematics

The profile orders to use the strict floating-point mode (FP-strict) in applications where precision and portability is important. As in Java, all the integer and floating types are available for mathematical operations. The *java.math* package is also accessible.

L2.1.3. Support for User documentation

As with the original rating given in Chapter 3. Two methods of commenting souce code is offered, as well as the meta-data types, which were introduced in Java 1.5.

L2.1.4. Support for a range of static types including subtypes and enumeration types

As with the original rating given in Chapter 3. Generic and enumeration types are supported from Java 1.5. These are not excluded from the profile because they are statically bound at compilation time.

L2.1.5. Coding style guidelines

Although the profile does not give detailed coding guidelines, a number of rules dictate the way that applications are written. For instance, the two phases of execution will undoubtedly help produce programs into a well-ordered structure, and the rules and guidelines in *A.3. Programming in the small* will reduce syntactical errors (e.g. by avoiding octal constants, and using parentheses appropriately). General coding guidelines for Java apply to Ravenscar-Java programs.

L2.1.6. Support for abstraction and information hiding

As an object oriented programming language, Java supports abstraction and information hiding through the class and interface. The profile does not hinder the use of such types in any way, albeit dynamic binding should be minimised.

L2.1.7. Assertion checking

Java 1.4 introduced the assertion facility, which is useful for checking various properties at run-time. Complex assertions can be created by means of invoking static methods from assertions. The profile permits the use of assertions.

L2.2. Language Processors / Run-time environment / Tools

L2.2.1. Certified (static/dynamic) analysis tools

Certification of tools is only possible when there is a certifying community. At the time of writing, there is no such organisation. However, all available tools for Java can be used for Ravenscar-Java programs due to the fact that the profile is a subset.

Several analysis tools are now being developed, and some means of testing their compatibility should be developed in the near future.

L2.2.2. Interface to other languages

Native methods should be avoided when there is no guarantee about their functional and non-functional behaviours (A3.Guideline 9). The use of such will also hamper portability.

L2.2.3. Code optimisation

As with the original rating given in Chapter 3. The profile keeps silent about code optimisation. Its main focus is on predictability, which often prohibits code optimisation techniques. However, proven techniques may be used for both source and compiled code.

L2.2.4. Code portability

Since the semantics of the language is not extended, a Ravenscar-Java program is executable on any RTSJ compliant virtual machines (yet, for high integrity applications, it is generally obligatory to use a certified run-time system when available). All new classes are implemented using RTSJ APIs.

6.1.3. Conclusions of the Assessment

In Chapter 3, an assessment of Java was conducted using the same criteria used above. The conclusions were that the language is exceptional for general purpose applications, yet incorporates features that hamper not only predictability, but also efficiency of the run-time and analysis tools. Comparing the two findings, we now look at the areas that the profile has improved. That way, we can confirm how close the profile is to the requirements of standards and guidelines for selecting programming languages for use in high integrity systems. Below, a comparison between the results from the two assessments is made (Level 1 Criteria only).

Level 1 Criteria		**Ratings**	
		Java with RTSJ	**Ravenscar-Java**
L1.1.1	Type safety / Strong typing rules	2	1▲
L1.1.2	Side effects in expressions / Operator precedence levels / Initial values	2	1▲
L1.1.3	Modularity / Structures	1	1
L1.1.4	Formal semantics / International standards	2	-
L1.1.5	Well-understood	1	-
L1.1.6	Support for domain specific or embedded applications	2	2
L1.1.7	Concurrency / Parallel processing	2	2
L1.2.1	Functional predictability	2	1▲
L1.2.2	Temporal predictability / Timing analysis	2	1▲
L1.2.3	Resource usage analysis	2	1▲
L1.3.1	Certified language translators / Run-time environments	2	2
L1.3.2	Run-time support / Environment issues	3	variable

Figure 6.1. Results from the assessment – Level 1

As can be seen from the figure above, five areas have improved specifically. The profile has scored the highest ratings in most of the criteria, although in a few areas ratings are left unchanged from the assessment of Java plus the RTSJ. Those that have not improved are not mostly related to language specific issues, for instance, international standards, certified compilers and run-time support. Such areas cannot obtain higher ratings because, at the time of writing, there is no certification process or international efforts to standardise the profile. However, the author expects, with the initiative shown in this thesis, a Java Community Process (JCP) or a similar plan will be set up to play a role in this[2].

It is noteworthy that the highest ratings are obtained by restricting many language features. It may seem contrary in some cases, especially the requirements on concurrency, that although Java's concurrency mechanisms are prohibited, we can still satisfy all the specifics of the criterion. Of course, expressiveness is reduced, but not to an extent that the profile cannot meet the requirements. This also has interplay with other criteria, particularly those regarding predictability issues. Limited, but expressive enough features facilitate simpler timing and functional analyses.

Results for the Level 2 criteria remain almost the same in the case of the Ravenscar-Java profile. This is because in most of the areas Java already possesses a number of beneficial features, for example, exception handling and code portability. This evidence again proves that Java was a sound candidate for programming high integrity software from the beginning.

[2] In fact, Ravenscar-Java has been one of the major inputs into two recent international efforts: EC-supported HIJA (High Integrity Java Applications) project and the Open Group's RTES (Real-Time and Embedded System) Forum, Ireland in 2005.

6.2. Expressive power – case study

As discussed briefly in the previous section, syntactic expressive power of the profile is reduced as a matter of course. This is a natural outcome of sub-setting a language, but the essential question to pose here is whether the profile is expressive enough for implementing complex high integrity software. More specifically, does the profile contain a set of features that are expressive enough to implement high integrity software and are they predictable at the same time? The fact that profile satisfies the criteria may not always mean that it is expressive enough to model and solve real world problems. This section investigates whether the prohibited language features are *essential*[3] or just *syntactic sugar*[4] by means of a realistic case study[5].

It would be natural to assume that a comparison would be required between code in Ravenscar-Java and corresponding code in full Java and RTSJ in order to evaluate the expressive power of the profile. However, it is really up to the developer who writes

6.2.1. Introduction to the Packet Utilization Standard and OBOSS-II

The Packet Utilization Standard (PUS) or ECSS-E-70-41A [ECSS-E-70-41A] was developed as part of the series of the European Cooperation for Space Standardization (ECSS) standards. The aim of the PUS was to reduce the costs and risks of developing and operating on-board and ground systems by standardizing and re-using common predetermined services. It is based on the assumption of the Packet Telecommand

[3] If a language feature is *essential*, it is not a syntactical sugar, such that it cannot be reproduced in any other form.
[4] Eliminable syntactic symbols [M.Felleisen]
[5] It would be natural to assume that a comparison would be required between code in Ravenscar-Java and corresponding code in full Java and RTSJ in order to evaluate the expressive power of the profile. However, it is really up to the developer who writes the code in any many he/she likes it by making full use of the whole features of Java. Also, presenting a normal Java program of this scale that does not conforms to the profile would be a rather tedious task.

Standard and Packet Telemetry Standard [ESA PSS-07-101], in which low-level packet details are specified. Essentially, the PUS addresses the end-to-end transport of telemetries and telecommands between user applications on the ground and application processes on-board [Merri2002, ESA PSS-07-101].

The PUS was shaped by defining generic operations concepts and on-board architectures to produce generic service model, as shown below. A mission needs not include all the services, but only those that are required for the particular mission. Such services can also be specialized by overriding or extending them. In addition to the standard services, mission-specific services are defined in the PUS framework.

No	**Service Name**	**No**	**Service Name**
1	Telecommand Verification	11	On-board Operations Scheduling
2	Device Command Distribution	12	On-board Monitoring
3	Housekeeping and Diagnostic Data Reporting	13	Large Data Transfer
4	Parameter Statistics Reporting	14	Packet Forwarding Control
5	Event Reporting	15	On-board Storage and Retrieval
6	Memory Management	17	Test
8	Function Management	18	On-board Operations Procedure
9	Time Management	19	Event/Action

Figure 6.2. Standard PUS Services [ECSS-E-70-41A][6]

Terma's OBOSS I and II are an implementation of the PUS services. It employs Ada as the base development language and is focused on re-using the services and on-board data handling software for satellites. One of the key components is the Command and Data Handling (CDH) system that is in charge of

[6] Note that the missing numbers in the figure are due to the revisions made through reviews and experiences.

routing all packets between ground stations and applications on-board the satellite. After verification, telecommands are stripped off and forwarded to appropriate applications by the CDH system, while telemetries are encoded into a valid telemetry format and sent down to the ground station through a communications subsystem. This fundamental service is implemented as an instance of the Packet Router package in the Ada code, which contains an event queue of TC or TM packets.

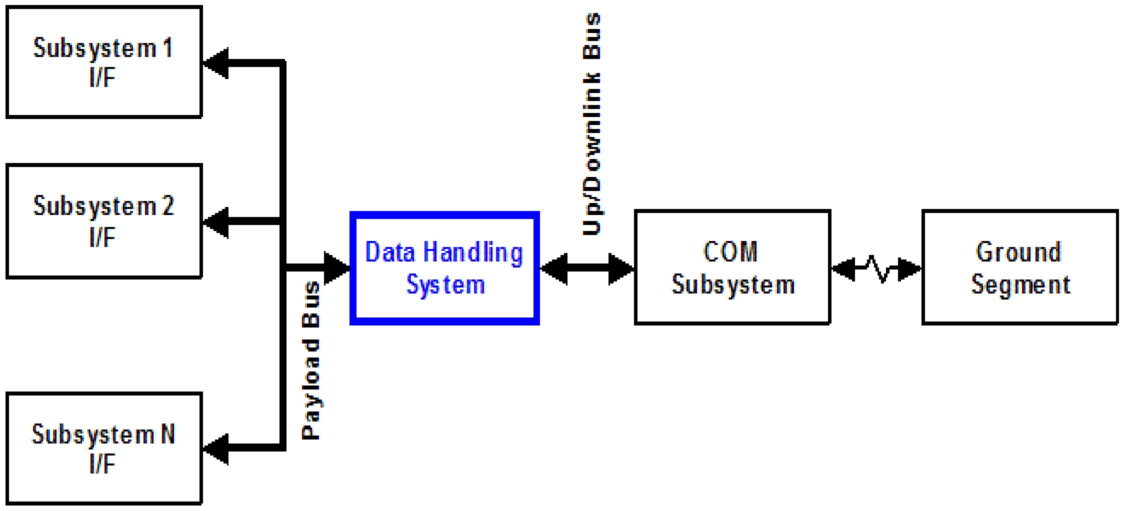


Figure 6.3. Command and Data Handling system (used to be called Data Handling System in the early version of the OBOSS) [Terma2004, ECSS-E-70-41A]

Two types of control structures are defined, namely, sporadic tasks and cyclic tasks. They are used in implementing other basic services, and are re-used for user application processes. Each sporadic task maintains an internal FIFO queue that keeps track of PUS packets. When a TC packet arrives for a particular sporadic task, the packet router forwards or *deposits* the packet in to the queue of the destination task. This call is non-blocking unless the queue is full. The task will be released whenever a packet arrives in the queue, invoking a user-provided function named Sporadic_Operation. Cyclic tasks, on the other hand, are self-contained active entities that are released periodically.

Packets, a stream of bits in reality, are represented as subclasses of PUSPacket, and are categorically defined as TC and TM packets, shown below in Figure 6.4. These are in line with the representations defined in the PUS, Telecommand and Telemtery standards. Each packet contains a packet ID and a destination field (i.e., an application ID) among others, so that the packet router can forward packets to correct destinations.

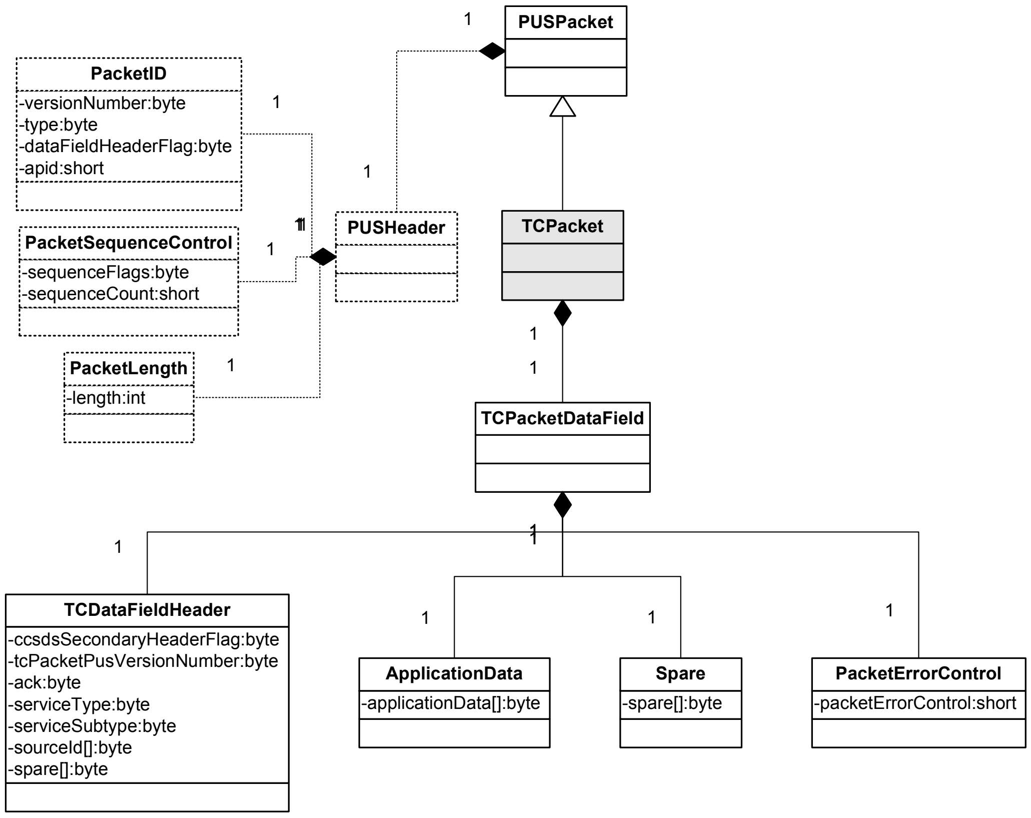


Figure 6.4. Object representation of TCPacket [Terma2004]

Along with the packet structures, all the software entities introduced so far can be classified into four layers (see Figure 6.5 below), i.e.,

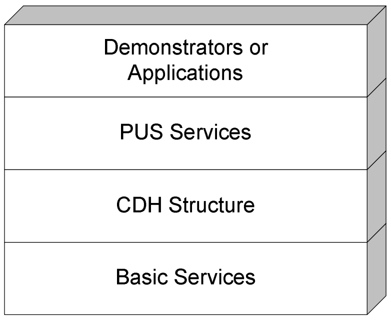


Figure 6.5. Four layers of PUS Services

Layer 1. Basic Services: contains components for a variety of simple services shared among all PUS services. It includes mission specific parameters, containers such as FIFO queues, control structures (cyclic and sporadic tasks), PUS packet representations.

Layer 2. CDH Structure: responsible for routing packets to correct destinations. Packet Router is the key element.

Layer 3. PUS Services: provides a collection of predefined services, which are full or partial implementations of the PUS services, e.g., Telecommand Verification Service, Memory Management, and Onboard monitoring.

Layer 4. Demonstrator or actual applications: A program that illustrates typical use of the services may be supplied with.

Currently, the OBOSS III software supports the following PUS services.

- Telecommand Verification Service
- Device Command Distribution Service
- Housekeeping & Diagnostic Data Reporting Service
- Event Reporting Service
- Memory Management Service

- Function Management Service
- Onboard Scheduling Service
- Onboard Monitoring Service
- Large Data Transfer Service
- Onboard Storage & Retrieval Service
- Event/Action Service

6.2.2. Implementation in Ravenscar-Java

Regarding the PUS architecture consisting of four layers, as mentioned above, it has been decided to implement a select subset of them. Since the OBOSS software and PUS services are large in scale and size, it was impossible to build all the services in a short time span with limited resources. Only some essential components in the *Basic services*, *CDH structure* and *PUS services* are implemented, on which further application-level PUS services can be developed in the future. The goal here is to illustrate that Ravenscar-Java is capable of developing large scale applications, and make improvements on previous studies, for instance, [Hultman and Pedrazzani2004], in which the authors ported parts of the services from Ada to Java (without considering the RTSJ).

A number of design and implementation decisions had to be made. The OBOSS written in Ada was a good starting point and model to follow. However, while Ada95 and Java share many modern programming concepts, such as object-oriented programming, there are a number of differences that must be addressed. Specifically, in Ada it is possible to constrain scalar types and define various record types, whereas in Java there are only two kinds of types, i.e., predefined primitive types and reference types. Hence, some deviations were unavoidable.

In addition, the two control structures in the OBOSS can easily be mapped to periodic and sporadic threads in Ravenscar-Java, although the names of the interfaces are different. For example, for each sporadic task in the OBOSS, there is a common operation, *sporadicOperation*() that will be invoked whenever an event occurs. This is modelled in Ravenscar-Java as a set of an event and event handler, such that whenever an event is fired, the handler's *run* method is invoked. An attempt has been made to keep the method and class names in line with those of the OBOSS. However, this case study is not a direct translation from Ada to Java, and as mentioned early, a number of deviations were inevitable in order to better implement the basic PUS services and requirements within the profile's capability.

There are other non-functional properties that deserve attention, such as memory management, and sharing data (packets) through immortal memory. The later sections of this chapter are devoted to this.

Implemented services: Overview

Regarding the four layers mentioned above, the following figure shows the implemented services in Ravenscar-Java for the purpose of this case study.

Layers	**Implemented services**
Basic Services	PUS packet structures, Non-blocking Queue, Sporadic and Cyclic tasks
CDH structure	Packet Router
PUS services	Telecommand verification service, Test service, Onboard Storage and Retrieval Service
Demonstrator/Applications	CameraController

Figure 6.6. Four layers of services

As a basis for other services, all of the classes shown in Figure 6.4 have been implemented, which are the essential packet structures. Other services are defined assuming those packet classes exist. The packet router will interpret the header part of each packet to decide a destination, i.e., an application process. Every packet is subject to verification, so that corrupt or malicious packets will be ignored by the on board system. Indeed, this task[7] is so essential, we have decided to perform TC verification whenever a telecommand packet arrives. However, internal packets generated by application processes are not subject to validation.

Having received a verified packet, an application thread[8] will read in the packet's data field, which is in a predefined format known to the service handler and caller. Essentially being a byte stream, the field will contain commands with or without data when finally interpreted. Appropriate actions are now taken according to commands received, and acknowledgements may be generated in the form of a telemetry packet and sent back to the originator through the packet router.

[7] We do not mean tasks in the sense of Ada here.

[8] Be it a housekeeping & data reporting service, large data transfer service, or a custom built services like a custom camera controller, etc.

As mentioned above, packets can be generated from any source including applications on board. This means all inter-application communications are also achieved by means of packets put through the packet queue. In order to comply with the Ravenscar-Java profile and to avoid copying packets between threads, a logical implementation may well create a shared queue in immortal memory. A thread that checks for new packets will first read raw packets and place them in the queue in immortal memory, while the packet router will be awaken whenever the queue is not empty, directing packets to correct destinations. In our first attempt, as will be seen later in this section, the destinations are all incorporated inside the packet router, such that only a single thread deals with routing as well as actually performing tasks according to commands through the packet. A more elaborate version will have separate threads (indeed applications) for each service and packet router, meaning that the router will not be delayed by carrying out applications' tasks. The following subsections will discuss our implementation strategies.

Packet classes

Classes shown below in Figure 6.7 are all implemented in Ravenscar-Java. The key issue here is converting raw packets in a byte stream format into *Packet* objects, be it a *TCPacket* or *TMPacket*, because only object representations will be used by application threads. The byte stream is placed in an array of bytes, so that each byte will be read one by one with the index increasing. The length of each packet is indicated before variable application data come in the stream, meaning only required number of bytes will be fetched to create an appropriate *Packet* object. Only after all values are set according to the raw packet data, the object is in a usable form. Backward conversion is also possible; internally generated packets need to be put in the queue in a raw byte stream and may be sent to ground stations through a communication medium. There were no particular difficulties in implementing these classes. See Appendix D for the code.

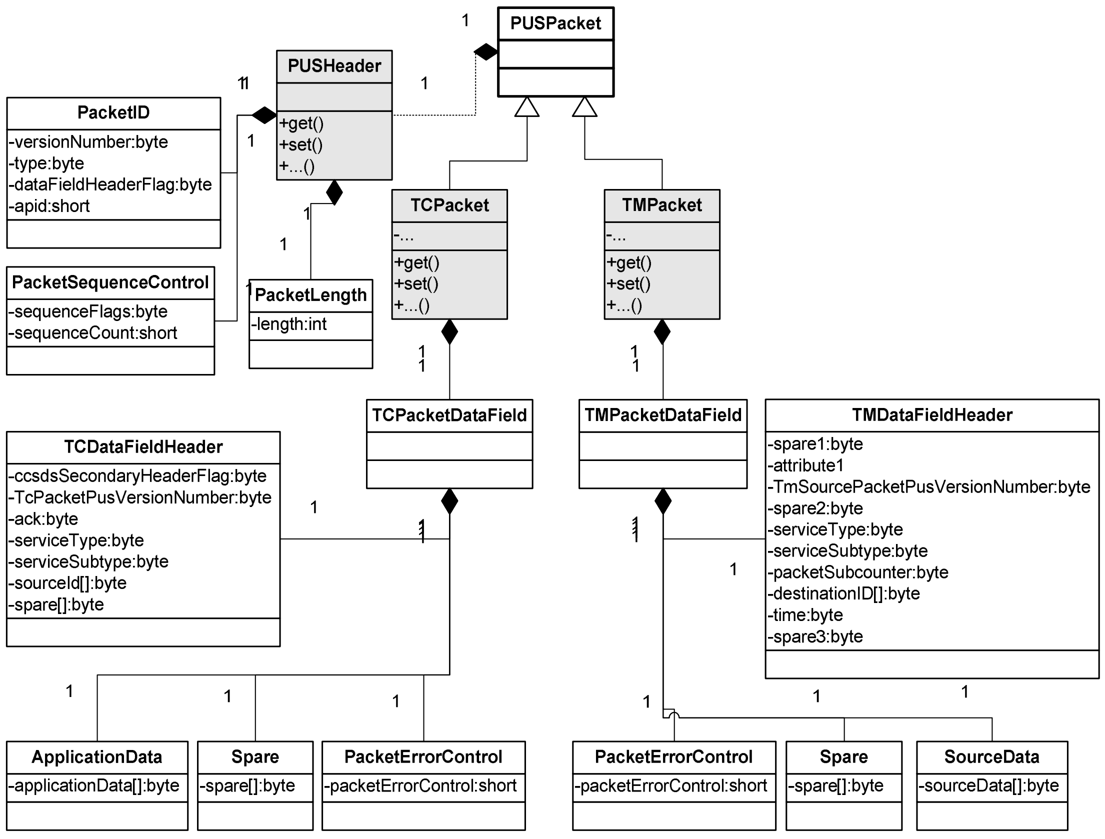


Figure 6.7. PUS related classes implemented in Ravenscar-Java

Packet Queue

Implementing a shared queue appeared to be a challenge in Ravenscar-Java because automated garbage collection is prohibited and all shared data must reside in immortal memory. This meant that elements of a queue must be created and reused with extreme caution in order not to result in memory leaks in the immortal memory. Any garbage created in that area will never be collected. This fact has influenced the design of the shared packet queue.

Figure 6.8 below reveals our proposed design where the packet queue is shared, and resides in immortal memory. Three data structures are defined, i.e., a *pool* of byte stream represented as an array, *element objects* each indexing a segment in the pool, and a *queue object* that maintains element objects in a circular FIFO queue. The pool is used to store all incoming and outgoing packets in byte, while each element object contains a head and tail indexes of one packet stored somewhere in the pool. It also encloses a Boolean flag indicating each packet's availability. This is needed so that packet element objects and their indexed segments can be reused. A queue object keeps all element objects together as a circular linked list. The sizes of the pool and queue are application-dependant, and to be provided in a mission parameters class as *final* constants. A class diagram in UML is presented in Figure 6.9.

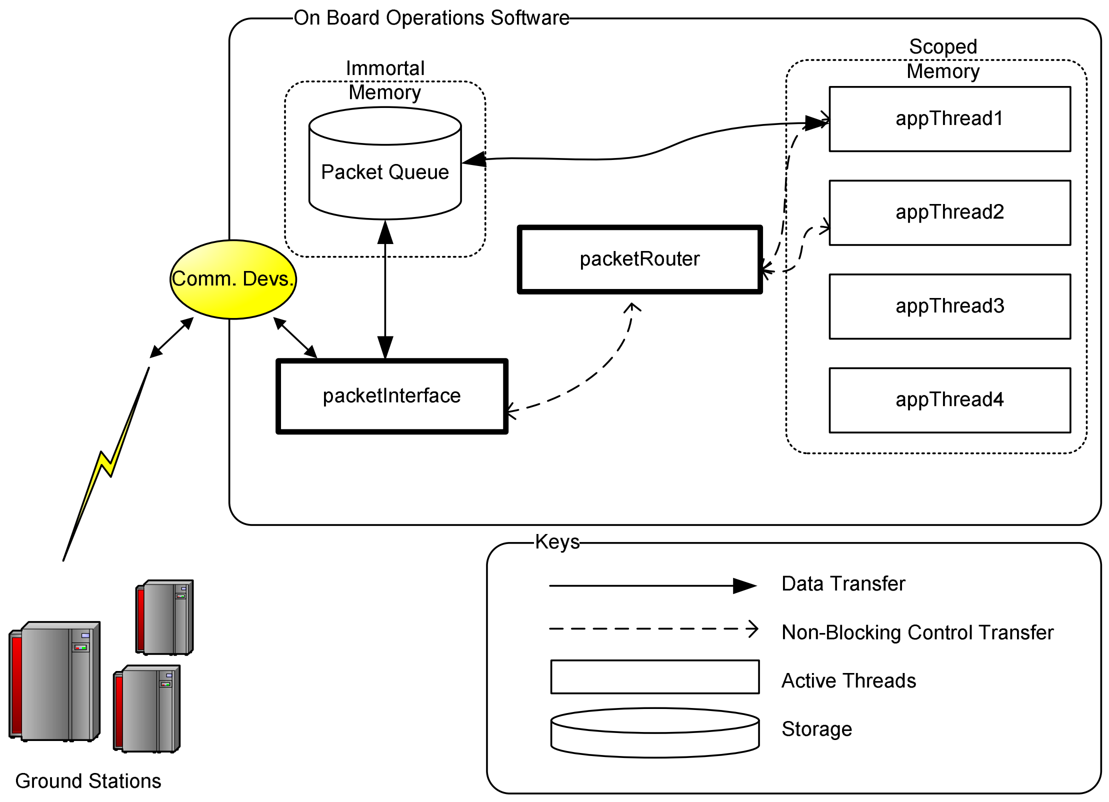


Figure 6.8. Data and control transfer view

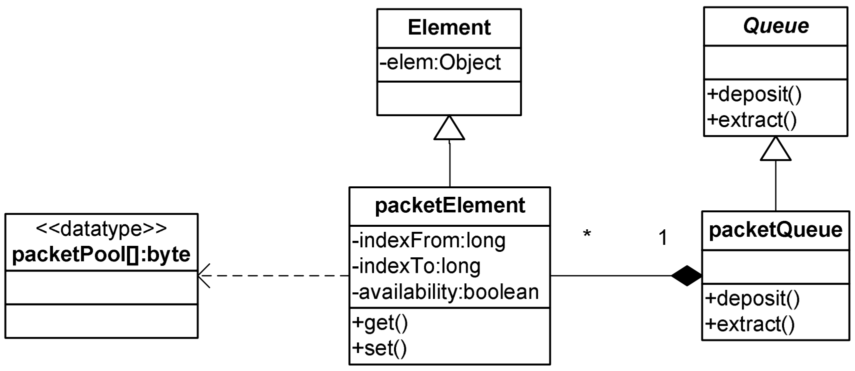


Figure 6.9. Class diagram of *packetQueue* and *packetElement*

A new object of *PUSPacket* (instance of either *TCPacket* or *TMPacket*) will be created every time the *extract*() method of *packetQueue* is invoked. Since that method is only called by an application thread awaken by the packet router, the memory area to be used will always be a scoped one dedicated to that particular thread. That way, after the *PUSPacket* object is created in a scoped memory area for a thread, the element index object, *packetElement* can now be reused by setting the value of availability to true. The byte stream in the pool that was reserved by that particular element of the queue will be freed, so that next time *deposit*() is executed, the method will be able to reuse the pair. The two methods in *packetElement*, i.e., *get*() and *set*(), are used to read and write packets in byte format.

References to the *packetQueue* object and *packetRouter* must be available to application threads. Otherwise, threads cannot gain access to the queue and packet router. Such references may well be passed to the threads in the construction time or through a *setter* method. Threads dependent to another, however, will not need the references. This is a possible situation as an application thread could make use of others in order to accomplish a complex job.

If garbage collection was allowed, this would have been an easy assignment in terms of memory usage. Yet, considering timing analysis and predictability, programs would become extremely difficult to analyse for the reasons we have given earlier. Our approach will not only make programs temporally predictable, but also facilitate a safer memory usage. In fact, one can bound the maximum amount of memory that will be used at run-time at any given instance. Moreover, it is feasible to reuse the queue structure for other applications in the future.

Control Structures

As suggested in the PUS and OBOSS documents, two types control structures are recommended; one that is released periodically and the other released by an event with a minimum inter-arrival time. Ravenscar-Java catches these requirements well already by means of the two derived thread classes, i.e., *PeriodicThread* and *SporadicEventHandler*.

These control structures are used to implement all application threads as well as other critical services like the packet router. The following list shows an example set of threads for an on board system. Of course, the first two are essential components of any system. Depending on missions, various numbers of threads (be it sporadic or periodic) will be created.

Threads/Runnables	**Functions**
packetInterface *<periodic>*	Interfaces to communication hardware to receive and send raw packets
packetRouter *<sporadic>*	Directs packets to destinations and performs TC verification
AppThread 1 *<...>*	Application specific tasks, e.g., controlling an on board camera, large data transfer, housekeeping and diagnostic data reporting, etc.
AppThread 2 *<...>*	
…	

Figure 6.10. Example set of threads

A typical way to implement a periodic thread is to create a *Runnable* object and pass it to the constructor of a *PeriodicThread* object. One can also extend the class directly, but the *run* () method cannot be overridden since the logic provides an automatic periodic release mechanism as defined in the profile. Other additional methods, as well as attributes, can be added to the new extending class, thus opening up the possibility for other threads to interface with this thread.

As in the OBOSS implementation, a packet queue may be attached to every application thread, which is seen as a buffer of events or packets in reality. In this case, the queue will not be shared by any other threads. When a packet for a thread has arrived, the packet router would fetch the packet and put it into the thread's queue. In our implementation, we define a new method, *forwardPacket*() as shown below in Figure 6.11, so that the packet router will call it whenever necessary. The way this method works is illustrated below. The call to *globalQ.extract*() will create a new *PUSPacket* object that is then stored in the internal queue (thus, the new packet object and queue are all in a scoped memory area of the application thread). In this case, the periodic thread may examine the queue during each release, and respond appropriately. Sporadic event handlers will also have the same method, but after that method is invoked by the packet router, associated events should be fired. All calls to *forwardPacket*() will be non-blocking. If the queue is full, an exception will be raised immediately and the caller (in this instance, the packet router thread) must deal with such unexpected outcomes. Such cases should benefit from a better analysis of packet arrival rates, queue sizes, and schedulability.

Another added method is *getInternalPacketQueue*(), which is useful when the associated *Runnable* object needs to gain access to the internal queue, i.e., when extracting a packet from the queue. The internal queue is instantiated in the constructor with properties passed as parameters, as shown in the code below (see the last two parameters). The two methods introduced just now are also added to the *PUSSporadicEventHandler* class.

Application threads often send acknowledgement or telemetry packets to the ground stations or other threads via the packet router. It is also possible that in one application there exist more than one packet router (which would be a rare case, but one could implement a router dealing with all incoming packets, the other outgoing packets). For this purpose, the *setPacketRouter*() method has been added to both *PUSPeriodicThread* and *PUSSporadicEventHandler* classes.

```
class PUSPeriodicThread extends PeriodicThread {
    PUSPeriodicThread (PriorityParameters priority,
                       PeriodicParameters period,
                       Runnable logic,
                       long[] memSizes,
                       long poolSizeForQueue,
                       int elementNumberForQueue) {
    …
    // Create a Packet Pool (byte array) and Queue
    }
    …
    public void forwardPacket(Queue globalQ) {
        internalQ.deposit( globalQ.extract() );
    }
    public final Queue getInternalPacketQueue () {
        return internalQ;
    }
    public final void setPacketRouter(PUSSporadicEventHandler pr) {
        currentPacketRouter = pr;
    }
    private Queue internalQ;
    long poolSizeForQueue; // If set to zero, this thread do not use pool.
    int elementNumberForQueue; // Again, if set to zero, no queue to use.
    PUSSporadicEventHandler currentPacketRouter;
}
```

<***PUSPeriodicThread*** class >

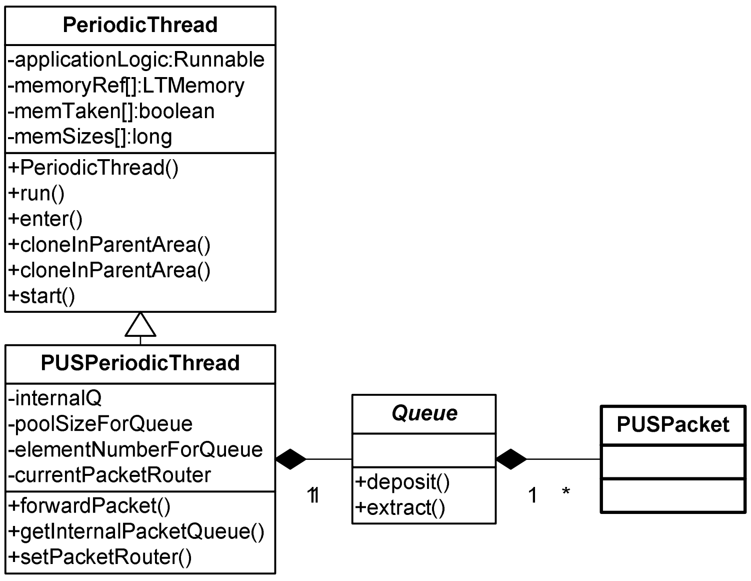


Figure 6.11. PUSPeriodicThread class extends PeriodicThread

As expected, there have been no particular difficulties in implementing the PUS control structures in Ravenscar-Java. The next subsection exposes the CDH structures that take advantage of the control structures defined here.

Packet Interface and Packet Router

Crucial instances of the two types of threads introduced above are the *packetInterface* and *packetRouter* classes. They implement *Runnable*, such that they will be passed to a *PeriodicThread* and *SporadicEventHandler* object respectively at construction time. The periodic thread, which takes a *packetInterface* object, will constantly monitor arrival of packets. Once it detects one it will place the packet in a shared queue and fire an event to wake the sporadic event handler that takes care of the packet. After a common TC verification, the event handler, *packetRouter* will perform a specific task depending on the command in the packet. Different tasks have different memory requirements, which could vary significantly.

The call from *packetInterface* to *packetRouter* is non-blocking (using the event firing mechanism), such that the on board software can response immediately to requests from ground stations. By extracting a packet from the queue whenever an event is fired, the *packetRouter* will perform a set of predefined tasks, possibly taking into account application parameters or data. Listed below is a possible implementation of the two threads that interact through a shared queue, without nested scoping of memory areas.

```
public class packetInterface implements Runnable {
	private packetQueue myQ;
	private SporadicEvent se;

	/** Set up a queue: the queue may be shared by all threads
	 *  so it must be in Immortal memory
	 */
	packetInterface (packetQueue q, SporadicEvent se) {
		myQ = q;
		this.se = se;
	};

	// Check if a new packet has arrived
	private boolean newPacketArrived() {//…};

	/** Read in packets (stream) from hardware
	 *  Convert the stream into an object
	 *  Each Packet consists of a representing object and byte stream
	 */
	private PUSPacket readPacket() {//…}

	// This run method will be executed periodically
	public void run () {
		if (newPacketArrived() == true) {
			// Get the packet and store it in the queue
			// deposit will copy the content into its internal array
			aQ.deposit(readPacket());
			// Fire an event
			se.fire();
		}
	}
}
```

<***packetInterface*** class>

The essential task of this thread is, as mentioned early, to check on newly arrived packets into the on board system. Upon the detection of a new arrival, the thread will read in the packet from the communication unit and store it in the queue in the application-level immortal memory. Next, an event for the packet router is fired, which eventually triggers the *run*() method to execute. The code below is a cut down version of the *packetRouter* class. Note that application threads are not yet incorporated for the sake of simplicity. Instead this thread itself deals with all requests in the *switch-case* statement.

```
public class packetRouter implements Runnable {
	private packetQueue myQ;
	// Initialize the packet router with a queue
	packetRouter(packetQueue myQ) {
		this.myQ = myQ;
	}
	public void run () {
		// 'extract' creates a new local copy of the packet object, and
		// set the original packet object in the queue and memory free
		currentPacket = myQ.extract();
		if (TCVerification(currentPacket) == true)
		{…}// if TC verification fails, set up an error code
		switch (currentPacket.getDestination()) {
			case CAMERA_CONTROLLER:
				// Take out the application data
				cameraCommand cc =
					currentPacket.getApplicationData();
				// Create & invoke objects for a variety of jobs
				OnBoardCamera.performCommands(cc);
				break;
			case ONBOARD_STORAGE_RETRIEVAL:
				    …
				break;
			case MEMORY_MANAGER:
				    …
				break;
				// Etc.
				// Different applications may have different
				// requirements on memory
			default:
				// Do something if the packet is not destined
				// to any entity on board.
		}
	}
	private boolean TCVerification(PUSPacket pk)
	{// TC verification logic:
		byte packetInByte[] = pk.getPacketInByte();
		  …
	}
}
```

<***packetRouter*** class>

As an event-triggered thread, this thread will extract a packet from the queue whenever an event is fired. This is followed by some verification process in case packets are corrupted or invalid. Depending on the destination, the switch-case statement will execute code for the requested job.

This simplified model can be extended, such that rather than the packet router executing all the code itself, additional sporadic or periodic threads may well be added which will execute the code concurrently (refer to Figure 6.8 above). That way, the packet router will be freed from application specific jobs and could easily be extended for future missions. Programs will become more structured, and timing and schedulability analyses should result in a more optimistic result.

Memory Management with Single Nested Scoping

<u>Concerns with Memory Usage</u>

Memory usage rate for a *packetInterface* object will remain constant in every release if received packets are of the same size. However, we cannot cast out completely cases in which packets of variable sizes[9] are allowed. A *packetRouter* object, on the other hand, may consume memory in a different fashion. Conditional on the target of each packet, *packetRouter* may create a variety of objects, requiring different amounts of memory in turn. Moreover, there are opportunities in the code that a certain amount of memory can be reused, i.e., memory used for *TCVerification*() could be reclaimed and reused if nested scoping is allowed. The worst case memory usage is determined by the maximum collected size of objects created in a scope dynamically. That way, if memory areas are reused, the worst case memory usage figure will decrease at the

[9] As shown early, to handle packets with variable sizes, a circular queue that operates on a byte array in immortal memory is a valid option. A queue of objects that keep head and tail indexes within the byte array would also be necessary to keep track of available and used parts of the byte array. Such arrays will not suffer from fragmentation because of the characteristic of the FIFO queue. Immortal memory is used since packets must be shared or copied between two or more threads.

expense of slightly amplified computation time for finalizing objects and re-entering areas. In many cases, this justifies its use in resource sensitive critical systems.

Case with Single Nested Scoping

This time, let us consider the case with the single nested scoping proposed in Chapter 4. The code for *packetInterface* may remain the same since its use of memory is not particularly diverse or demanding among releases. However, *packetRouter* can be improved by reusing, for example, a memory area dedicated to telecommand verification for other given jobs later at some point of the code. Overall, the code should remain almost the same, but the parts that need to be executed in a new memory area must be implemented as *Runnable*s. That means we should now have extra *setter* methods (including constructors), if required, in the *Runnable* objects to pass parameters or data to the new context. An example is shown below: one of the constructors takes a packet as parameter, and there is also a setter method.

```
/*
 * This Runnable is to be executed in a new memory area
 */
class TCVerification implements Runnable {
        PUSPacket p;
        // Constructor
        TCVerification () {
        }
        TCVerification (PUSPacket pk) {
                p = pk;
        }
        public void setPacket(PUSPacket pk) {
                p = pk;
        }
        public void run() {
                if ((p == null) || (verifyTCPacket(p) == false)) {
                // If verification fails, do something here
                }
        }

        public boolean verifyTCPacket(PUSPacket pk) {
                // Logic for verification
        }
}
```

<***TCVerification*** class>

Before entering a memory area, an object of *TCVerification* must be created and a packet be passed to one of the constructors or *setPacket*(). The following syntax is required for a *Runnable* to enter a new memory area (notice the underlined code). *Thread.currentThread*() is a static method call that returns the reference to the current thread. Information regarding memory areas is completely hidden from application programmers and is managed internally. Hence, single parent rule checks are entirely unnecessary while programs become more structured and safer. The *enter*() method takes a *Runnable* object and the required size of memory.

```
…
TCVerification tcv = new TCVerification (pk);
PeriodicThread ct = (PeriodicThread)Thread.currentThread();
ct.enter(tcv, 1000);
…
```

<Entering a new memory area>

Note that if there is a need for data to be copied back to a calling context, one should consider using a *call back* object, which should also be passed to the *Runnable* before it enters a memory area. With the call back object at hand, the *Runnable* can clone an object in a parent area (by using one of the *cloneInParent*() methods), and then call a setter method of the call back object to pass the reference to the cloned object in to a parent area.

The application services implemented inside the *switch-case* statement in *packetRouter* can now reuse the memory area that was once used by *TCVerification* (see the code below). Any subsequent request of memory area with a smaller or the same size has a potential to be allocated in the same area. However, if a thread is maintaining more than one memory area, and their sizes are different, then the closet in size to the required amount of memory will be selected and used. The next time a new memory area is requested whilst running inside the previous area, the thread will have fewer choices that fit into the request.

```
…
CameraController cc = new CameraController (pk);
PeriodicThread ct = (PeriodicThread)Thread.currentThread();
ct.enter(cc, 1000);
…
```

<Entering a new memory area>

When to use Single Nested Scoping?

Each thread has a single dedicated memory area in the profile. One would have to reserve memory space large enough for every object created in the *Mission phase* by the thread during each release. While loops and method invocations can result in a huge number of temporary objects that are simply waste of space later, all the garbage created in each release will not be collected until the next release. A limited form of dynamic memory allocation will facilitate more efficient memory usage in the profile while maintaining the predictability of the current model.

More specifically, recursions[10] and loops with new statements inside and/or method calls should be re-structured into a *Runnable* object and executed in a different memory area. In other words, anything programming logic that is considered to consume an excessive amount of memory should be placed in a separate area.

Another issue is that the information to be exchanged between the calling and *callee* contexts should be small, such that 'object cloning and copying process' will take less time. This also means the two logics are functionally modularised, or *loosely coupled*. In short, a trade-off is essential between the maximum dynamic memory consumption allowed, and added computation time for managing memory areas and finalizing objects.

Limitations of the approach to Single Nested Scoping

There is an overhead incurred by the selection algorithm, used inside *PeriodicThread* to choose the best fit memory area. In order to make it predictable, it operates on a couple of internal arrays, and takes a constant time to search for the best fit. If there are an enormous number of memory areas (say 10,000, although not realistic), this

[10] Currently Ravenscar-Java profile does not allow method recursion.

algorithm will suffer. A more efficient logic can be developed that takes advantage of more advanced sorting and searching algorithms.

To the outside of a thread, memory areas are only known by their sizes. However, there may be a certain cases where a specific memory area must be used in a particular situation. For example, there could be a fast on-chip memory that should be used only by a demanding logic. Some way to designate a memory area to a specific *Runnable* could be an advantage. One possible idea is that memory areas be indexed, so that they are requested by not just sizes but also by a direct index as well.

Integrating Application Threads

Putting the packet interface and router together with application threads, a realistic multithreading program is constructed. It may also utilise multiple memory areas in order to make programs more flexible and predictable at the same time on memory usage. A plus is that the worst case memory usage can be bound as well. Here, a scenario is presented that involves an on board camera controller, and the Storage and Retrieval service (which is a standard PUS service).

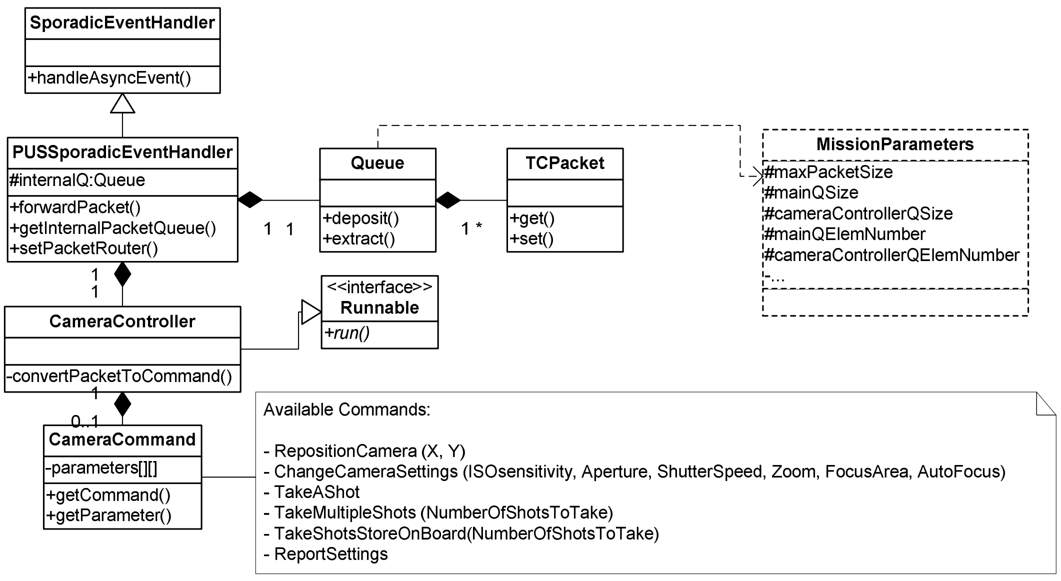


Figure 6.12. Camera Controller class

Figure 6.12 shows a class diagram of *CameraController*, which implements *Runnable* and passed to a *PUSSporadicEventHandler* object at construction time (let us call it *camController*). This event handler is associated with an event object that will be fired by the packet router whenever a packet destined to the camera controller arrives. Just before firing the event, however, the router will call *forwardPacket*() of the event handler, so that the new packet will have been copied into the internal queue of the handler by the time it is woken up by the event.

Once the event is fired, the *Runnable's run*() method (i.e. that of *CameraController*) or *handleAsyncEvent*() method will be executed. In this scenario, rather than directly extending the event handler class, we have decided to create a *Runnable* object (i.e., *CameraController*) and pass it to the constructor of the *PUSSporadicEventHandler* class. Hence, the *run*() method is now executed and will need to gain access to the internal queue created as part of the construction process of the handler. That way, the camera controller can obtain commands sent from ground stations, possibly with parameters, and act appropriately. For this scenario, the following commands are defined for the camera controller. These will be encoded into the packets sent to the camera controller, and there will be a specialised class for representing such commands.

Commands	Parameters	Functions
`RepositionCamera`	`X, Y`	Reposition the camera.
`ChangeSettings`	`ISOsensitivity, Aperture, ShutterSpeed, Zoom, FocusArea, AutoFocus`	Change camera settings, which will be applied from the next shot taken.
`TakeAShot`		Take a shot from the satellite.
`TakeMultipleShots`	`NumberOfShotsToTake`	Take multiple shots.
`TakeShotsStoreOnBoard`	`NumberOfShotsToTake`	Take shots and store in on board memory. May be retrieved later.
`ReportSettings`		Report the current settings to the ground station.

Figure 6.13. Available commands for *CameraController*

The *run*() method of *CameraController* is organised as in the following code. Note that the *switch-case* statement handles commands, and take appropriate actions. Consider especially commands such as *TAKE_A_SHOT*, *TAKE_MULTIPLE_SHOTS*, and *TAKE_SHOTS_STORE_ONBOARD*, which are predefined in the *MissionParameter* class.

```
/**
 * @author Jagun Kwon
 * Real-Time Systems Research Group
 * Department of Computer Science
 * University of York, York, UK
 * Last updated on: 21 October 2005
 */

import EarthObservationMission.MissionParameter;
import EarthObservationMission.CameraControl.*;
```

```
class CameraController implements Runnable {
       CameraController () {
              // Set up the camera with default values
       }
       public void run() {
              // Get the ref. to the current thread
              PeriodicThread ct = (PeriodicThread)Thread.currentThread();
              Queue internalQ = ct.getInternalPacketQueue();
              PUSSporadicEventHandler pkRouter = ct.getPacketRouter();

              // Extract the packet stored in the internal queue and
              // Convert the packet into a camera comment object
              PUSPacket pk = internalQ.extract();
              CameraCommand cmd = new CameraCommand(pk);

              // TMPacket to acknowledge
              CamControllerTMPacket ack;

              switch (cmd.getCommand()) {
                     case REPOSITION_CAMERA:
                            // Code Omitted
                            break;
                     case CHANGE_CAMERA_SETTINGS:
                            // Code Omitted
                            break;

                     case TAKE_A_SHOT:
                            // This is the logic that actually takes pictures
                            // by interacting with the camera hardware
                            Shooter sh = new Shooter();

                            // The result will be stored here
                            callBackObj cb = new CallBackObj();
                            sh.setCallBackObj(cb);

                            ct.enter(sh, MissionParameter.MAX_PIC_SIZE);

                     // Picture taken is passed through the call back object
                     // and stored in this memory area (cloned)
                            ack = new CamControllerTMPPacket(cb.get());

                     // a new TM packet is created with the picture taken
                     // as data
                            ack.setDestination
                                          (MissionParameter.GROUND_STATION);
```

```
                // Packet router will now take care of the packet
                pkRouter.forwardPacket(ack);
                break;

         case TAKE_MULTIPLE_SHOTS:
                // First parameter indicates the number of shots
                // to be taken
                Shooter sh = new Shooter(cmd.getParameter(1));

                // Enters a new MA and take more than one picture
                for (int i=0; i<cmd.getParameter(1); i++)
                {
                // Second parameter gives the memory size limit
                       ct.enter(sh, cmd.getParameter(2));

                // Picture taken is passed through the
                // call back object and stored in this memory area
                // (cloned)
                       ack = new CamControllerTMPPacket(cb.get());

                // a new TM packet is created with the picture
                // taken as data
                       ack.setDestination
                                (MissionParameter.GROUND_STATION);
                // Packet router will now take care of the packet
                       pkRouter.forwardPacket(ack);
                }
                break;

         case TAKE_SHOTS_STORE_ONBOARD:
                // First parameter indicates the number of shots
                // to be taken
                Shooter sh = new Shooter(cmd.getParameter(1));

                // Enters a new MA and take more than one picture
                for (int i=0; i<cmd.getParameter(1); i++)
                {
                // Second parameter gives the memory size limit
                       ct.enter(sh, cmd.getParameter(2));

                // Picture taken is passed through the
                // call back object and stored in this memory area
                // (cloned)
                       ack = new CamControllerTMPPacket(cb.get());
                       // a new TM packet is created with
                       // the picture taken as data
                       ack.setDestination
                      (MissionParameter.ONBOARD_STORAGE_RETRIEVAL);
```

```
                                        // Packet router will now take care of
                                        // the packet
                                        pkRouter.forwardPacket(ack);
                                }
                                break;

                        case REPORT_CURRENT_SETTINGS:
                                // Read the current settings from the hardware
                                // …
                                ack = new CamControllerTMPPacket();
                                ack.setDestination
                                                (MissionParameter.GROUND_STATION);
                                pkRouter.forwardPacket(ack);
                                break;
                        default:
                }
        }
}
```

<*CameraController* class>

Multiple shots can be taken and transmitted to the ground station, while there is an optional command that prescribes the application thread to store pictures in the on board storage and retrieval service. In this case, the TM packet generated is passed on to the On Board Storage and Retrieval Service through the packet router. The storage service will check periodically whether a new packet has arrived, and store packets' data in its own pool with appropriate identification. It will also have a command for generating a new packet and return some previously stored to anywhere. This service is only partially implemented, and has a similar structure to that of *CameraController*.

Notice the use of scoped memory areas within the code for *TAKE_MULTIPLE_SHOTS* and *TAKE_SHOTS_STORE_ONBOARD* commands. Whenever a picture is taken a certain amount of memory is required to store the picture, such that, if nested scoping is not allowed, the thread would consume a total size of the sum of all pictures taken at once. In order words, the worst case memory usage is dependent on the number of shots taken. However, with our proposed approach, the worst case will be the size of only one shot plus memory required during entering and exiting memory areas, and cloning a picture object into parent area.

6.2.3. Summary of the case study

In this case study, we have illustrated that a realistic and large system can be implemented in Ravenscar-Java. The profile catches the requirements of high integrity real-time systems well, especially through the ready-made control structures (i.e., periodic and sporadic threads), and shared immortal memory, and the analysable single nested scoping approach. These confirm the expressive power of the profile is sufficient for high integrity applications with similar complexity levels.

Nevertheless, being a subset of the full language and RTSJ, there were a few challenges to meet, and deviations were inevitable. For example, without a garbage collector, implementing a shared array that resides in immortal memory could cause problems such as race conditions. This is so particularly when there are multiple threads that have access to the array. However, a practical solution was presented; the packet queue class in the case study controls the byte array, and access to it is only possible through the synchronized methods in the class. Therefore, we no longer need to rely on an automatic garbage collection mechanism.

We have also shown a way to pass objects between different memory areas by making use of call back objects, and *cloneInArea*() methods. Although all the concerns and extra efforts would not be required if a garbage collection mechanism was allowed, the reasons for not opting for it are documented in the early part of this thesis, i.e., to make programs more predictable and analysable in terms of timing and schedulability as well as memory usage.

The case study may well serve as a starting point where further services will be implemented, either mission specific services or standard services of the PUS. Based on the control structures, new application threads can be created, and packet router's destination fields added or modified. Everything else should remain the same (except for mission parameters, such as the size of queue and pool). All the classes can be reused by extending them, but must be re-analysed to check extended classes' memory usage and timing requirements.

6.3. Summary

This chapter is dedicated to evaluating the profile from two perspectives, i.e., how it meets the requirements of high integrity standards and guidelines, and how a realistic application can be developed by means of a case study. First, the results from the assessment are convincing since the profile scored the highest possible ratings in most of the areas, apart from a few exceptions on certification issues, and dependency on platform or implementation of virtual machines.

Second, the case study showed that the expressive power of the profile is sufficient in developing applications with a similar or even higher level of complexity. It is intuitive to assume the profile as a limiting factor to the development of complex software. However, we have confirmed the profile's potential as a base development language for high integrity applications, where an appropriate balance between expressiveness and predictability/analysability is the key target to achieve. Next chapter will summarise the work in full and draw a few conclusions. Future works, mostly regarding the limitations of the approach presented here, will also be given.

Chapter 7. Conclusions and Future Work

High integrity systems are the ones that must not fail. They play key roles in our society, for instance, in transportation and tele-banking. By all means, we strive to engineer the best possible approaches, so that the possibility of failure is kept to a minimum. Over the past few decades, such systems have become more and more complex, and the software technology is employed to better manage the complexity of application domains, reduce production cost, and increase flexibility in design and implementation. The choice of programming languages that will be used to implement the software is thus an important factor to any successful development and maintenance of modern high integrity systems. There are many considerations to make; one being whether or not an expertise or user-base is readily available.

In this thesis, Java with the Real-Time Specification for Java has been examined. After gathering language selection criteria from influential standards and guidelines, we have developed an amalgamated framework of 23 assessment criteria that also reflect on advances in modern programming languages in Chapter 3. Object orientation and concurrency were considered essential in order to facilitate software reuse and modelling of realistic interactions between applications and their environment. The categorised criteria were then used to evaluate how close Java comes to the ideal requirements from various standards. The results were mixed; in certain areas, like strong typing, support for concurrency and well-understoodness, Java scored high. In areas that require analysis, it failed to do so mainly due to the complex features and overheads associated with the run-time. Side effects can also occur in expressions and be quietly discarded.

The Ravenscar-Java profile was motivated by the assessment. By sub-setting the full language, we attempted to overcome the shortcomings revealed in the assessment, yet still maintain the advantages Java and RTSJ have to offer. The profile is not just a collection of APIs, but also includes new rules and guidelines specific to Java and RTSJ. Such rules and guidelines were inspired by the Nuclear Regulatory Commission's guidelines [NUREG/CR-6463]. The profile is different from previous works in a number of ways. First, it is based on requirements raised by certification standards, which were surveyed and incorporated into our framework of criteria. Second, it is positive about concurrency, yet restricts the overly expressive and complex model in Java and RTSJ. With the ordered computational model and new classes added, static schedulability and WCET analyses now become possible, and concurrency related errors, such as deadlocks and race conditions, are cast out. Third, the profile also deals with sequential features of the language. Generic programming guidelines are also given with appropriate rationales, such as how to avoid side effects in expressions.

The extensions to the profile address a new dynamic memory allocation scheme and annotations for documenting the developer's intention and program proof. In the original profile, since each thread can only have one scoped memory area, enough memory for the worst case must be reserved for each thread. The memory usage estimation task could result in a very pessimistic value due to, for example, temporary objects created in loops and libraries. To increase flexibility and re-use memory, the Single Nested Scoping approach is proposed. It not only allows flexible allocation of memory areas within a thread, but also eliminates some of the costly run-time checks, such as the single parent rule. It also provides a safe way to copy objects between different memory areas, thus preventing assignment violations.

The second extension is the annotations for the safe use of methods, maximum loop bounds, prohibited classes and methods in the profile. The `@ScopeSafe` annotation is used to document a method's behaviours in terms of assignments. The validity of such annotations can be checked by analysing each method's logic, after which the entire thread-wide analysis of inter-methods and inter-memory areas relationships can be carried out. Furthermore, maximum loop bounds are vital in the worst-case execution time and worst-case memory usage analysis. They are documented to provide analysis tools with more accurate information, and reduce analysis complexity because such tools do not need to deduce the loop bounds themselves. All prohibited classes and methods are also marked with an annotation, so that they cannot be used directly by application threads. For instance, prohibited entities include the Java's original *Thread* class and most of its methods, which must not be used and extended directly in Ravenscar-Java programs.

Having presented the profile and its extensions, the impact that the profile has on analysis techniques was investigated. In other words, we discussed how analysis techniques could benefit from our approach. Especially, a deeper understanding was gained on what is meant by checking conformance. The profile with the rules and guidelines was developed without the knowledge of how they could be enforced, in particular, mechanically. All the rules were categorised into five groups and required analysis efforts and strategies for each group were discussed. Several of the guidelines are, however, difficult to check because they are either general programming recommendations or computationally too demanding.

Other analysis techniques considered are memory usage analysis, shared object analysis, and memory area access analysis. It was shown how to calculate the worst case memory consumption of each thread and how large its own memory area should be. In doing so, we developed a new tree structure that consists of multiple Instantiated Objects Table (IOT) nodes, which presents newly created objects in each basic block. By traversing the tree with branches and IOT nodes, one can obtain the worst-case path by adding the cumulative sizes of objects on every path and comparing them. The tree structure is also beneficial in detecting shared objects as well as locating unnecessary assignment checks. Of course, other analysis techniques (such as, the escape analysis) can be used as well. Moreover, without synchronized statements, locks on arbitrary objects cannot be obtained.

In terms of performance, the profile does not penalise the user to an extent that it would be unreasonable to consider the profile in high performance systems. Assuming that a synchronized method and block are used in the same context, one of the test results revealed that there is only a marginal difference in the number of instructions executed. When synchronized statements are replaced by synchronized methods, the performance figure drops only slightly. On the experimental platform, the number of times a synchronized method is executed in a second is 326447.34 with 10 threads, just 0.03881% short of that of synchronized blocks.

The profile is then assessed from two different perspectives, i.e., how well it meets the requirements against the criteria developed in Chapter 3, and how expressive it is in developing realistic applications. First, the results from the assessment are convincing since the profile scored the highest possible ratings in most of the areas, apart from a few exceptions on certification issues, and dependency on platform or implementation of virtual machines. It may seem contrary in some cases, especially the requirements on concurrency, that although Java's expressive concurrency mechanisms are prohibited, we can still satisfy all the specifics of the criterion. Expressiveness is certainly reduced, but not to an extent that the profile cannot meet the requirements. This also has interplay with other criteria, particularly those regarding predictability issues. Limited, but expressive enough features facilitate simpler timing and functional analyses.

Second, the case study showed that the expressive power of the profile is sufficient in developing applications with a similar or even higher level of complexity. It is intuitive to assume the profile as a limiting factor to the development of complex software. However, we have confirmed the profile's potential as a base development language for high integrity applications, where an appropriate balance between expressiveness and predictability/analysability is the key target to achieve.

In conclusion, the work presented in this thesis gives a firm indication that a subset of Java and the RTSJ augmented with static analysis techniques is a valid approach to the development of high integrity software. The resulting programs become predictable and analysable while run-time overheads are kept low. Additionally, the profile simplifies program analysis to a great extent. We have given evidence of potential use of analysis techniques that can benefit from the profile. Because of the simpler computational model and less error-prone features, such analysis techniques can be designed and developed more reliably.

7.1. Future work

Model Checking for Ravenscar-Java programs

As mentioned in Chapter 2, there are a number of successful model checkers. They typically deal with models of software. However, because of the simplified computational model, a model checker specific to the profile could be developed. State space that such model checkers will need to handle will be a great deal smaller than that of full Java. For example, the JavaPathFinder [JPF2005] could be modified for this purpose. With annotations, model checking can also be modularised, and, once checked, a class never needs to be verified again.

High Integrity RMI

The profile keeps silent about networking. Nevertheless, there are situations where one is required to communicate with other nodes in a high integrity system. A real-time RMI framework may be developed and is a good candidate for such applications.

High Integrity Java Isolates

There is also an on-going development on supporting multiple applications on a single virtual machine by means of Java Isolates [JSR121]. Applications can have a different criticality, and scheduling them on one virtual machine poses interesting problems, especially when one needs to exchange information with one another. Hao and Wellings address this issue in [Cai2004, Cai2005], but further improvements would be possible.

Memory Area Usage Optimization

If there is a precedence relationship between threads, memory can be reused between those threads, assuming memory areas are actually reserved when ‘enter’ is invoked rather than when a memory area’s constructor is invoked (this is how it is done in RTSJ reference implementations - for example, the FLEX VM [Beebee2001, Beebee2001b]). Currently the maximum memory requirement for any Ravenscar-Java applications is the sum of the sizes of all memory areas plus that of immortal memory reserved (trade off between memory footprint and allocation time).

7.2. Summary

This thesis has examined Java and the RTSJ in the context of high integrity real-time systems, and presented the Ravenscar-Java profile. It has also discussed issues related to the correctness of code, conformance, memory analysis, and a realistic case study was given that implements part of the Packet Utilisation Standard. A few future works are outlined above, which may play an important role in the success of the profile in the future.

Appendix A. Ravenscar-Java Rules and Guidelines

A complete list of rules and guidelines are provided here. *Rules* are to be interpreted as strong instructions that must be obeyed, whereas *guidelines* are highly recommended, but not compulsory, practice. They are grouped into three areas, i.e., programming in the large, concurrent real-time programming, and programming in the small. Each of the groups is further categorised, so that it is discovered that which area a certain group of rules are aimed at improving. For instance, Rule 1 to 4 in Programming in the large (A.1) are focused on improving reliability, while Guideline 3 to 7 are on robustness.

Each item is denoted with *M*, *T*, or *C*; *M* means that the particular rule/guideline is concerned with predictability of memory utilisation, *T* with predictability of timing, and *C* with predictability of control flow. Certain rules and guidelines are related to more than one area, while a few exceptions cannot naturally fit in any of the three categories.

A.1. Programming in the large

A.1.1. Reliability

Rule 1. Avoid dynamic class loading in the mission phase (M, T, C)

Additional class loading at runtime is seen as overheads to both the virtual machine and the application. Accurate memory and timing analyses are thus impossible and dependent on the location and size of classes, as well as the implemented loading and linking mechanisms.

In order to prevent dynamic class loading, either the virtual machine has to preload all classes that the application utilises, and/or the application must not be permitted to load any class in the mission phase by restricting the use of the following classes and their subclasses (i.e. user defined class loaders). This can be achieved by employing a class hierarchy analyser.

- **java.lang.ClassLoader**
- **java.lang.Class** (*forName*() methods of this class)
- **java.net.URL.ClassLoader**
- **java.security.SecureClassLoader**

Rule 2. All user-defined classes must include constructors that initialise all internal variables and objects (C, M)

Java automatically allocates initial values to variables, but programmers must not depend on those as they can be mistakenly used or misinterpreted. Such initial values can also differ from system to system. This rule is equally applied to reference types.

Guideline 1. Use only necessary and analysable classes in the class library for the application domain (C, M, T)

To keep the complexity and memory requirement of the application to the minimum, not only should we use absolutely necessary classes in the library, but also the behaviours of such classes must be statically analysable in terms of temporal and functional characteristics.

Guideline 2. Minimise dynamic method binding (T, C, M)

Dynamic method binding makes it complex to perform various flow analyses and to obtain the worst-case execution time of a thread. Although there may be only a few choices or branches of methods, the runtime overheads incurred by the virtual machine will be hard to predict and, thus undesirable. Accurate memory requirement analysis can also be difficult when different methods have different memory utilisation. Therefore, programmers are encouraged not to excessively *override* and *overload* methods with ones that have significantly differing logics and overheads, as this can result in a pessimistic timing analysis. Where the logics are significantly different, the programmer should avoid dynamic dispatching by ensuring that class hierarchies are not passed as parameters to methods.

This guideline equally applies to utilising interface types; the virtual machine needs to resolve a method reference every time it encounters an interface method call at runtime by searching through an interface method table for the method reference since the organisation of the table may vary from class to class that implements the same interface [Venners1999].

Along the same line, *monomorphic* method invocations are greatly recommended wherever possible, in place of *polymorphic* invocations. Code optimisation tools may be used to assist this task.

Rule 3. Do not use or override **java.lang.Thread** to create (non real-time) threads (T, M)

Threads must not be created by instantiating or overriding **java.lang.Thread** class because it provides possibly unsafe asynchronous operations, as well as an inaccurate timing and priority model that are inconsistent with the Real-Time Specification for Java [Bollella2000a]. It is also impossible to explicitly specify memory requirements for such regular threads. Instead, the real-time thread classes of the specification must be used for all real-time and even non real-time threads (in case of non real-time threads, they must be given a low priority than critical ones and may not invoke **waitForNextPeriod()** method). Ideally, however, applications should make use of the **Initializer**, **PeriodicThread**, and **SporadicEventHandler** classes defined in the profile. Refer to *A.2. Concurrent Real-Time Programming* for a more detailed explanation.

Rule 4. Do not utilise Java classes to schedule threads (T, C, M)

Programmers must not make use of the pure Java classes that can be used to schedule threads with an incompatible timing and priority model. Such classes, for example, **java.util.Timer**, **java.util.TimerTask**, and **java.util.Calendar**, ought to be replaced by appropriate counterparts of the Real-Time Specification for Java [Bollella2000a], which will be discussed in *A.2. Concurrent Real-Time Programming*.

A.1.2. Robustness

• Controlling use of exception handling

Guideline 3. Minimise propagation of exceptions (C, T, M)

Guideline 4. Localise handling of predefined exceptions (C, T, M)

Guideline 5. Handle all user-defined exceptions (C, T, M)

Guideline 6. Clearly express and document all user-defined exceptions

(All related to each other)

In Java, when exceptions are not handled locally (i.e. within a *try-catch* block in a method), their enclosing methods will be terminated and returned to the calling method(s). This terminating-and-returning process will continue until an appropriate handler is found. This not only hampers program analysability and adds overheads at runtime, but also could lead to an entire system failure if no proper handler can be located. Hence, every possible effort has to be made to eliminate any uncaught exceptions, i.e. unchecked exceptions and errors.

The *finally* clause may be added to a *try-catch* block, which will always execute before control transfers to a new destination (unless System.exit() method is invoked in the *try* block). It can be used to prevent the propagation of any uncaught exceptions at an outer level of the program or all application threads (i.e. in the initialisation phase) by explicitly transferring control from the finally clause itself to a safe destination, thus abandoning any pending (and possibly disastrous) control transfers that could halt the whole system. A safe destination, which may be part of the initialisation phase, may attempt to restart the application threads that have failed due to an uncaught exception or error, or replace them with threads that have different logics (i.e. N-version programming [Burns2001]).

• Checking input and output

Guideline 7. Methods for input and output should be written defensively (C)

It is a common practice to write a program such that it checks whether or not all input and output values from it are within a legal or specified range. This may prevent some unwanted programming errors. However, this job may be left to program verification tools possibly tailored to a specific application.

A.1.3. Traceability

• Readability

See Readability in *A.1.4. Maintainability*.

• Controlling use of native functions and compiled libraries

See Guideline 1 in Predictability of control flow above.

A.1.4. Maintainability

• Readability

Guideline 8. Comment on the purpose, scope, and date of creation for each object

Guideline 9. Comment on the purpose, and exceptions raising and handling for each method

Guideline 10. Identify dynamic method binding with comments

• **Portability**

Any Java program that supports this profile should be executable on a virtual machine that implements the Real-Time Specification for Java [Bollella2000a]. This, however, does not imply that such virtual machines will always succeed in providing an accurate, robust and cost-effective runtime base because they may not have been developed with high integrity applications in mind.

A.2. Concurrent Real-Time Programming

A.2.1. Reliability

Rule 1. Avoid the use of any garbage collection mechanism and heap memory (M, T)

It has long been argued that the runtime behaviour of the implementation-dependent garbage collector is difficult to predict in terms of its resource (including CPU time) and memory utilisation [Bollella2000a, Venners1999]. Although there have been some works to improve the situation as in [Henriksson1998, Kim1999], it is still challenging to put them into practice. If, however, a predictable garbage collector becomes available, a cautious decision should be made by a reliable organisation after evaluating its usage in high integrity real-time systems.

Without a garbage collector and the use of heap memory area, programmers are able to utilise only *immortal*, *scoped* and *physical* memory areas defined in the Real-Time Specification for Java [Bollella2000a] to allocate objects. The initialisation phase will use immortal memory by default, and object creation in that memory area is allowed only in the initialisation phase (Refer to Rule 2 below).

This rule also renders the use of **java.lang.ref** class obsolete, which allows Java programs to interact with the garbage collector.

Rule 2. Object creation in an *immortal* memory area must be allowed only in the initialisation phase (T, M)

By definition, objects in an immortal memory area cannot be freed or moved, and all threads in an application share the memory area [Bollella2000a]. Hence, in an attempt to prevent memory run-out and possible programming errors, this rule is enforced. Object creation during the mission phase should make use of linear-time scoped memory areas (i.e. **LTMemory** class).

Rule 3. Do not create or instantiate schedulable objects in the mission phase (T, M)

Creation or instantiation of schedulable objects, i.e. threads and events, will cause the underlying virtual machine to allocate new memory space and handle a new set of information, which will delay the execution of other threads for an indefinite time. This will hamper low-level memory and timing analyses. Therefore, all schedulable objects must be created in the non-time-critical initialisation phase.

Rule 4. The size of an **LTMemory** area shall not be extended (M, T)

(Also related to Rule 5 below)

In the RTSJ, the **LTMemory** (or linear time scoped memory) class takes two parameters, one for the initial size, and the other for the maximum size in byte. The two sizes must always be the same in this profile, because any additional memory allocation at runtime may be seen as overheads to the virtual machine, and may not be necessary because of static memory analysis.

Rule 5. Access to **LTMemory** areas must not be nested (M, T, C)

The RTSJ allows nested entry and exit of scoped memory areas in the form of a stack and/or *cacti*. This can be inefficient and error-prone because the virtual machine needs to check dynamically whether scopes are properly nested, the single-parent rule [Bollella2000a] is observed, and object reference relationships are correct. Such runtime checks are not desirable, and may have ambiguous time or memory requirements. With this restriction, however, static analysis can readily ensure that the assignment rules (for assigning references to objects within different memory areas) are correctly obeyed. That is, an object in the immortal memory/heap must not be able to obtain a reference to an object within a scoped memory area to prevent any dangling references. As explained in Chapter 4, one thread is allowed only one scoped memory area (except in the case of the proposed extension) and access to immortal memory.

Rule 6. LTMemory areas must not be shared between **Schedulable** objects (M, T, C)

As with the Rule 5 above, it is difficult to cost-effectively validate if a given scoped memory area is exploited correctly when different schedulable objects share it. Ideally, one thread should have only one dedicated **LTMemory** area to it, or should use the immortal memory area. With this rule enforced, additional overheads of dynamic memory access checking are eliminated, and the virtual machine design and implementation can be significantly simplified.

Rule 7. Create all memory area objects during initialisation phase (M, T)

All memory areas must be created only in the initialisation phase, in order to prevent any runtime overheads for allocating a new memory area.

Rule 8. *Finalizers* must not block (T, M, C)

Generally, finalizers of objects are invoked when the virtual machine detects that there is no more reference to the objects. In the context of the scoped memory area, this process should occur when a memory scope is escaped (i.e. the *reference count* becomes zero), and all the finalizers of the objects in the scope should be invoked. Finalizers of objects allocated in the immortal memory area will only be invoked when the whole application terminates, or there is no runnable *non-demon* thread.

The overheads of finalizers must be taken into account when performing schedulability analysis, and the virtual machine can take some time to free up used memory areas. On the whole, finalizers should be as compact as possible and must not block (for example, by invoking the *sleep()* method).

Rule 9. Asynchronous transfer of control (ATC) and any thread aborting mechanisms are disallowed (C, T, M)

These features result in high runtime overheads, and obscure static timing and flow analyses. All abnormal conditions that may necessitate the use of ATC must be identified at design stages and prevented by means of off-line analysis and design.

Rule 10. Do not use *wait*, *notify*, and *notifyAll* methods (T, C)

This rule eliminates the need for the whole object queue management in the virtual machine, resulting in more efficient and deadlock-free programs.

Rule 11. Use the *synchronized* method construct that implements either the priority inheritance protocol or ceiling emulation (M, C, T)

The *synchronized* method construct provides mutually exclusive access to shared resources or objects, and programmers are always encouraged to use it to avoid data races (see also A.3. Rule 9). However, excessive use of it may well result in a poor response time, implying that high priority threads that become ready to run may have to wait until lower ones finish their *synchronized* methods. Therefore, in order to prevent unbounded priority inversions and deadlocks, the priority inheritance protocol (by default) or priority ceiling emulation must be implemented in the runtime system and explicitly used for all objects with synchronized methods.

Rule 12. Use only **NoHeapRealtimeThread** class to create periodic threads (T, M)

In the absence of a garbage collector and heap memory area, the **NoHeapRealtimeThread** class has naturally to be a default framework for modelling periodic threads, which may typically utilise a linear-time scoped memory area. Moreover, only the **waitForNextPeriod** method of that class must be used to delay associated threads because other delay statements (e.g. *sleep*) in Java almost certainly cause difficulties in timing and control flow analyses, and are not compatible with the Real-Time Specification for Java. Preferably, programmers should make use of the **PeriodicThread** class defined in Section 4 of this paper, which automates the timely execution of a given **Runnable** logic.

Rule 13. Use only **BoundAsyncEventHandler** class to model sporadic and event-triggered activities (T, M)

An instance of the **BoundAsyncEventHandler** class is bound to a dedicated thread permanently, and this way of handling sporadic events eases timing analysis. All event handlers must be initialised and set up with one event each before the mission phase starts. Once this task is complete, the application must not attempt to rebind the handlers with other event(s), since it will make timing analysis unfeasible. Again, the **SporadicEventHandler** class, defined in Section 4, should preferably be used.

Rule 14. Do not use processing groups, overrun and deadline-miss handlers (T, M)

The RTSJ allows applications to define processing groups, and have overrun and miss handlers associated with real-time threads. Yet, these are likely to be overheads, as they require runtime support for the scheduler to determine the feasibility of the temporal scope of a processing group. Timing analysis must be statically performed before despatching high integrity software, thus making processing groups and the two sorts of handlers unnecessary.

Guideline 1. Concurrent software design should be as simple as possible (T, M, C)

There should be no more threads and synchronisations than necessary, so that predictable programs will be produced with low performance penalties. Once an application has entered its mission phase, no thread may be created and despatched.

A.2.2. Robustness

See Rule 9 in Predictability of control flow above.

A.2.3. Traceability

No specific rules and guidelines.

A.2.4. Maintainability

• Readability

Guideline 2. Identify threads with comments (T)

Guideline 3. Identify memory objects with comments (M)

A.3. Programming in the small

A.3.1. Reliability

Rule 1. Avoid method recursion (C, T, M)

Recursive method calls (including mutually recursive calls) can dramatically consume available memory space at runtime, and an erroneous termination condition can cause unbounded recursion. However, this rule may be relaxed if the memory consumption for each method and termination conditions can be formally verified.

Rule 2. Do not use *continue* and *break* statements in loops (C)

The *continue* and *break* statements can be used to jump out of a loop in an uncontrolled manner, which makes static analysis difficult to perform.

Rule 3. Use brackets for every branch in *if-else* statements (C)

The *if-else* statements can have a branch that has a single statement, and such branches do not need brackets. But it can be confusing and lead to programming errors.

Rule 4. All constraints, such as one used in a *for* loop, must be static (C, T)

This facilitates the prediction and analysis of memory and time requirements of loops prior to program execution. If, however, constraints can change during the course of the program, then at least a tight upper bound must be easily deducible.

Rule 5. Variable declarations must include a static initialisation expression (C)

Java automatically allocates initial values to variables, but programmers must not depend on those as they can be mistakenly used or misinterpreted. Such initial values can also differ from system to system. This rule is equally applied to reference types.

Rule 6. Eliminate compound expressions in parameter passing to methods (C)

Expressions that are used as part of parameters for method calls can easily cause side effects and misinterpretation, leading to unintended behaviours of the program. These particularly include ones with the increment and decrement operators (i.e. ++ and --), which depending on the syntactic position can produce different results. Relational expressions should not appear.

Rule 7. Avoid expressions whose values are dependent on the order of evaluations (C)

In relational operations, the evaluation of the right-hand expression of a logical operator (such as the logical *AND* (&&)) is decided by the truth-value of the left-hand operand. In other words, only if the left-hand expression is considered to be true, will the right-hand one be evaluated. Consequently, it is not recommended for a right-hand expression to contain any operators that can have an influence on the intermediate result of any object or variable, like the assignment operator.

Guideline 1. Use parentheses rather than rely on the default order of precedence (C)

Guideline 2. Use parentheses in bitwise operators (C)

Guideline 3. Use parentheses in comparisons and conditions (C)

Parentheses should be used wherever the meaning of an expression can be vague and needs to be clarified. This will prevent any misinterpretation by programmers.

Guideline 4. Use only one *return* statement per method, preferably at the end of each method (C, T)

Multiple return statements can make flow and timing analyses difficult or pessimistic to perform.

Guideline 5. Define *defaults* in *switch-case* statements (C)

It is a good programming practice to explicitly state that a *switch-case* statement performs either some given operations or default operations in case there is no condition satisfied.

- **Predictability of mathematical or logical result**

Rule 8. Use the strict floating-point mode (FP-strict) instead of the FP-default mode

The FP-default, introduced in Java 1.2, allows a virtual machine to utilise supported floating-point hardware to speed up its execution, and may store intermediate data in the hardware specific format. However, this way of representing intermediate data is dependent on underlying hardware, and thus hinders portability and even accuracy. Therefore, the original IEEE 754 formats of Java should be used. Nevertheless, this rule may be relaxed if the required precision for a particular application is not important.

Rule 9. Statements that access shared resources or objects must be guarded by synchronized methods (C, M, T)

Data races can occur if a shared object or variable is accessed by more than one thread, and at least one of them updates the object. The original *synchronized* method statement must be used to avoid race conditions, and it is generally the job of the programmer (or a tool) to ensure such statements are safely used and compact enough not to seriously affect the response time of other threads (see also A.2. Rule 11). However, synchronized *blocks* are disallowed in the profile, as one can lock any arbitrary objects accessible.

Rule 10. Do not use octal constants

Octal constants can be confused with other (decimal) numbers, since any number beginning with a zero will be interpreted as an octal constant by the compiler.

Guideline 6. Remember that integers are truncated when divided

The results of integer divisions are always truncated in Java without any warning such that the precision of the values will be reduced. Floating-point types should be used to prevent any integer truncation.

Guideline 7. Ensure that arithmetic operations produce a result that can be correctly represented

The ranges of values for each type must be considered, and only appropriate values and variables of correct types should be used. Overflow and underflow will never be caught or warned by the compiler, and values may be widened if different types of values and variables are used in expressions.

Guideline 8. Shift operators must be used with caution

The unsigned right shift operator, i.e. >>>, can result in an unexpected value when applied to integer types. That is, Java integer types are all signed and this operator will fill the high-order bits of an integer with zeros, thus possibly changing the sign of that integer value to positive. The left shift operator can also alter the sign of a value.

A.3.2. Robustness

None.

A.3.3. Traceability

• Controlling use of built-in functions and compiled libraries

Guideline 9. Minimise the use of native methods (especially without source code) (C, T, M)

The use of native methods will certainly hamper portability, and such methods may not have been constructed in the same manner that most other high integrity software is built. In other words, they may inconsistently handle errors, input and output data, and not follow programming rules developed by a governing body or a profile such as this.

A.3.4. Maintainability

• Readability

Guideline 10. Blocks should be bounded with brackets

Guideline 11. Minimise use of literals

Appendix B. Ravenscar-Java Profile API Specification

Java classes that do not appear here are assumed to be unchanged from the original, and can be used in association with the profile.

B.1. Classes related to Execution Phase

ravenscar.Initializer

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

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

B.2. Classes related to Memory Utilisation

ravenscar.MemoryArea

```
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();
}
```

ravenscar.ImmortalMemory

```
package ravenscar;

public final class ImmortalMemory extends MemoryArea

{

  public static ImmortalMemory instance();

}
```

ravenscar.ScopedMemory

```
package ravenscar;

public abstract class ScopedMemory extends MemoryArea

{

  public ScopedMemory(long size);

  public ScopedMemory(SizeEstimator size);

  public void enter();

  public int getReferenceCount();

  }
```

ravenscar.LTMemory

```
package ravenscar;

public class LTMemory extends ScopedMemory

{

  public LTMemory(long size);

  public LTMemory(SizeEstimator size);

}
```

B.3. Classes related to Threading and Scheduling

ravenscar.Schedulable

```
package ravenscar;

public interface Schedulable extends java.lang.Runnable
{

}
```

ravenscar.Scheduler

```
package ravenscar;

public abstract class Scheduler
{
}
```

ravenscar.PriorityScheduler

```
package ravenscar;

public class PriorityScheduler extends Scheduler
{
  public int getMaxPriority();
  public int getMinPriority();
}
```

ravenscar.ReleaseParameters

```
package ravenscar;

public class ReleaseParameters

{
  protected ReleaseParameters();
}
```

ravenscar.PeriodicParameters

```
package ravenscar;

public class PeriodicParameters extends ReleaseParameters
{
  public PeriodicParameters(AbsoluteTime startTime,
                            RelativeTime period);
  protected AbsoluteTime getStartTime();
  protected RelativeTime getPeriod();
}
```

ravenscar.SporadicParameters

```
package ravenscar;

public class SporadicParameters extends ReleaseParameters
{
  public SporadicParameters(RelativeTime minInterarrival);
  protected RelativeTime getMinInterarrival();
}
```

java.lang.Thread

```
package java.lang;
@RavenscarProhibited
public class Thread implements Runnable
{
  Thread();
  Thread(String name);

  void start();
}
```

ravenscar.RealtimeThread

```
package ravenscar;
@RavenscarProhibited
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();
}
```

ravenscar.NoHeapRealtimeThread

```
package ravenscar;
public class NoHeapRealtimeThread extends RealtimeThread
{
   NoHeapRealtimeThread(PriorityParameters pp,
         MemoryArea ma);
   NoHeapRealtimeThread(PriorityParameters pp,
         PeriodicParameters p, MemoryArea ma);

   void start();
}
```

ravenscar.PeriodicThread

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

  public void run();
  public void start();
}
```

ravenscar.AsyncEventHandler

```
package ravenscar;
@RavenscarProhibited
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();
}
```

ravenscar.BoundAsyncEventHandler

```
package ravenscar;
@RavenscarProhibited
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();
}
```

ravenscar.SporadicEventHandler

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

ravenscar.AsyncEvent

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

ravenscar.SporadicEvent

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

ravenscar.SporadicInterrupt

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

B.4. Classes related to Representation of Time

ravenscar.HighResolutionTime

```
package ravenscar;
public class HighResolutionTime
{
}
```

ravenscar.AbsoluteTime

```
package ravenscar;
public class AbsoluteTime
{
}
```

ravenscar.RelativeTime

```
package ravenscar;
public class RelativeTime
{
}
```

Appendix C. Implementation Details of PeriodicThread and Illustration on the use of Call-back Objects

C.1. Implementation of PeriodicThread

```
/**
 * @author Jagun Kwon
 * Real-Time Systems Research Group,
 * Department of Computer Science,
 * University of York, UK
 */
package RavenscarJavaProfile;
import javax.realtime.*;

public class PeriodicThread extends NoHeapRealtimeThread {

        // Private classes
        private class cloneCallBack {
                Object clonedObject;
                void set (Object o){
                        clonedObject = o;
                }
                Object get () {
                        return clonedObject;
                }
        }
        private class cloneObject implements Runnable {
                cloneCallBack cloned;
                MACloneable toClone;
                cloneObject (cloneCallBack cb, MACloneable o){
                        cloned = cb;
                        toClone = o;
                }

                public final void run() {
                        cloned.set(toClone.clone());
                }
        }

        // Constructor
        public PeriodicThread(PriorityParameters priority,
                        PeriodicParameters period,
                        Runnable logic,
                        long[] memSizes) {

                super (priority, period, null);

                applicationLogic = logic;
                this.memSizes = memSizes;

                // Array of boolean, its size being the length of memSizes
                this.memTaken = new boolean[memSizes.length];

                memoryRef = new LTMemory[memSizes.length];

                // Create memory areas
                for (int i=0; i<memSizes.length; i++) {
                        memoryRef[i] = new LTMemory(memSizes[i]);
                        memTaken[i] = false;
                }
        }
```

```
public final void run() {
        boolean noProblems = true;
        while (noProblems) {
        // The first memory area will be used by the current thread
                memTaken[0] = true;
                memoryRef[0].enter(applicationLogic);

                noProblems = waitForNextPeriod();

        //      noProblems = waitForNextRelease();
        }
        // A deadline has been missed and no handler has been specified.
        // In which case, we need to decide the default recovery
        // strategy for SCJ.
        System.out.println("*** A Deadline has been missed!");
}

public final static void enter(Runnable logic, long memSize) {
        long smallest = Long.MAX_VALUE;
        int indexToTake = -1;

        // Number of iterations = memSizes.length, i.e. always constant
        for (int i=0; i<this.memSizes.length; i++) {
                if (this.memTaken[i]==false) {
                //Find the most suitable(smallest) available memory area
                        if ((memSize <= this.memSizes[i]) &&
                                        (this.memSizes[i] < smallest)) {
                                smallest = this.memSizes[i];
                                indexToTake = i;
                        }
                }
        }

        if (indexToTake == -1){
                // No suitable MAs fould!!
                // There are elegant ways to exit, but for now, exit()!
                System.out.println("Warning: No suitable MAs found!");
                System.out.println("This program must stop!");
                System.exit(-1);
        } else {
                // System.out.println("PeriodicThread: Suitable MA found");
                // System.out.println("PeriodicThread: Entering a new MA");

                memoryRef[indexToTake].enter(logic);
        }

}
```

```
	public final static Object cloneInParentArea(MACloneable o) {
		// Create an object in the just outer MA and return the ref.
		return cloneInParentArea(o, 1);
	}

	public final static Object cloneInParentArea(MACloneable o,
								int index){
		// index must be between 0 (current MA) and memRef.length

		cloneCallBack toReturn = new cloneCallBack();
		// these new objects are created in the current context,
	       // i.e., where this method is called

		MemoryArea outerMA;

		if (getMemoryAreaStackDepth() <= index)      // Index is not correct
		{
			System.out.println("PeriodicThread:Index is incorrect");
			// In this case, simply enter the MA with index 1,
			// which is the first MA created for this thread
			// in the Initialization phase.
			outerMA = getOuterMemoryArea(1);     // 1 = First scoped MA
			System.out.println("PeriodicThread:outerMA
			     is set to the first scoped MA");
		}
		else {
			outerMA = getOuterMemoryArea(getMemoryAreaStackDepth()
			     -(1+index));
		}

		outerMA.executeInArea(new cloneObject(toReturn, o));

		return toReturn.get();
	}

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

	private Runnable applicationLogic;
	private LTMemory[] memoryRef;
	private boolean[] memTaken;
	private long[] memSizes;
}
```

C.2. Illustration on the use of Call-back Objects

The program below uses three memory areas, and has one periodic thread. The order of invocations is

Initializer(*Immortal*) → **PeriodicThread**:**T1**(*ScopedMA*) → **PeriodicThread**:**T2**(*ScopeedMA*)

T2 creates an object (called *data* in the program) that will be used by T1 after T2 finishes. Hence, T2 calls *cloneInParentArea*() to clone the object in the lower MA, then exits the area. T1 creates and uses a callback object to keep the reference to the new object. The reference must be kept consistent throughout the program (see the *get*() and *set*() methods). Annotations and their check would be a great advantage.

```
/**
 * @author Jagun Kwon
 */
import RavenscarJavaProfile.*;
import javax.realtime.PriorityParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.AbsoluteTime;
import javax.realtime.RelativeTime;

public class SNSExample1 extends Initializer {

        public void run() {

                T1 logicT1 = new T1();

                PeriodicThread thread1 = new PeriodicThread(
                                new PriorityParameters(MAX_PRIORITY),
                                new PeriodicParameters(
                                        null, //new AbsoluteTime(0,0),
                                           // Start must be null
                                        new RelativeTime(2000, 0), // Period of 2s.
                                        null, //new RelativeTime(100, 0), // Cost
                                        null, //new RelativeTime(200, 0),//Deadline
                                        null, //new AsyncEventHandler(),
                                        null), //new AsyncEventHandler()),
                                logicT1,
                                new long[]{20000, 1000, 400, 10000});
                thread1.start();
                System.out.println("Initializer Finished!");
        }

        public static void main(String[] args) {
                SNSExample1 init = new SNSExample1();
                init.start();
        }
}
```

```
/**
 * @author Jagun Kwon
 */

import hardRealTimeProfile.*;

public class T1 implements Runnable {


	public T1(){
		// This object itself is created in the Immortal Memory!
		// But enters a new Scoped MA selected by PeriodicThread,
		// which means every object created here in the constructor
		// will be created in the Immortal Memory Area.
		System.out.println("T1: T1's constructor executed.");
	}

	public void run() {	// This will be run in MemoryAreaIndex(1)
				// Immortal MA's MemoryAreaIndex(0)
		T2 t2;
		callBack cb;
		t2 = new T2();
		System.out.println("T1: t2 has just been created.");

		// This call back object will be used to pass object references
		// whose objects are created in a new MA, and to be cloned into
		// a lower or parent MA.
		cb = new callBack();
		t2.setCallBackObject(cb);

		PeriodicThread ct = (PeriodicThread)Thread.currentThread();
		System.out.println("T1: Entering into T2.");
		ct.enter(t2, 10000);

		System.out.println("T1: Just Returned from T2.");

		//t2.getCallBackObject();
		Object result = cb.get(); // returned obj is created in this MA
		System.out.println("T1: Got the reference to result.");

		// Print out the result that was created in T2
		result.toString();

		System.out.println("Hoooooray! :)");
		System.out.println("");
	}

}
```

```
/**
 * @author Jagun Kwon
 */
import hardRealTimeProfile.*;

public class T2 implements Runnable {

        callBack cb;

        public T2(){
                cb = null;
        }

        public void setCallBackObject (callBack c)
        {
                cb = c;
        }

        public void run() {
                System.out.println("T2: T2's run method executed.");

                // resultToReturn is what we want to return to the lower MA!
                data resultToReturn = new data();

                // So we clone it in the Parent MA, as in the following
                PeriodicThread ct = (PeriodicThread)Thread.currentThread();
                Object r = ct.cloneInParentArea(resultToReturn,1);
                // the index parameter in cloneInParentArea, if given, is the
                // index from the current MA. So 0=the current,
               // 1=direct parent MA...

                System.out.println("T2: Just cloned data to the parent area!");

                // Now r is the reference to resultToReturn that is cloned
                // into the outer MA!
                // In order to return the reference, we use the callback object
                // cb, as seen below.
                cb.set(r);
        }

}
```

```
class callBack
{
        Object o;

        callBack()
        {
                o = null;
        }

        public void set(Object o)
        {
                this.o = o;
        }

        public Object get()
        {
                return o;
        }
}
```

```
package hardRealTimeProfile;

public interface MACloneable extends Cloneable {
	public Object clone();
}
```

```
import hardRealTimeProfile.*;

class data implements MACloneable
{
	String result;

	data() {
		result = new String("data: This is the result!!!");
	}

	data(String o) {
		this.result = o;
	}

	// clone: Create every object that this data class uses!
	public data clone() {
		return new data(new String(result));
	}

	public String toString()
	{
		System.out.println(result);
		return result;
	}
}
```

Appendix D. Cast Study: List of Code

D.1. Classes for the PUS Packet structure

```
//
// PUSPacket class
//

public class PUSPacket {
        private PUSHeader packetHeader;

        public PUSPacket(){
                packetHeader = null;
        }

        public PUSPacket(PUSHeader ph)
        {
                packetHeader = ph;
        }

        public PUSHeader getPacketHeader() {
                return packetHeader;
        }

        public boolean isTMPacket() {
                return (this instanceof TMPacket);
        }

        public boolean isTCPacket() {
                return (this instanceof TCPacket);
        }
}
```

```
//
// PUSHeader class: PUS Packet Header
//

public class PUSHeader {
        private PacketID packetId;
        private PacketSequenceControl packetSequenceControl;
        private PacketLength packetLength;

        public PUSHeader () {
        }

        public PUSHeader(PacketID pid){
                packetId = pid;
        }

        public PUSHeader(PacketID pid, PacketSequenceControl psc, PacketLength pl) {
                packetId = pid;
                packetSequenceControl = psc;
                packetLength = pl;
        }

        public PUSHeader(byte vn, byte t, byte dfhf, short apid, byte sf,
                        short sc, short l) {
                packetId = new PacketID(vn, t, dfhf, apid);
                packetSequenceControl = new PacketSequenceControl(sf, sc);
                packetLength = new PacketLength(l);
        }

        public synchronized void setPacketSequenceControl(PacketSequenceControl psc) {
                packetSequenceControl = psc;
        }

        public synchronized void setPacketLength(PacketLength pl) {
                packetLength = pl;
        }

        public synchronized short getApid() {
                return packetId.getApid();
        }

        public synchronized short getSequenceCount() {
                return packetSequenceControl.getSequenceCount();
        }

        // Returns the external packet header in byte format
        public synchronized byte[] get() {
                byte[] out = new byte[6];

                for (int i=0; i < out.length; i++)
                        out[i] = 0;
                out[0] = (byte)(packetId.get() >> 8);
                out[1] = (byte)(packetId.get());
                out[2] = (byte)(packetSequenceControl.get() >> 8);
                out[3] = (byte)(packetSequenceControl.get());
                out[4] = (byte)(packetLength.get() >> 8);
                out[5] = (byte)(packetLength.get());

                return out;
        }
}
```

```
public class PacketID {
	private byte versionNumber = 0;
	private byte type;
	private byte dataFieldHeaderFlag;
	private short apid;

	// The size in terms of number of bytes of the PacketID
	public static final byte BYTE_SIZE = 2;
	public static final byte BIT_SIZE = 16;

	public PacketID() {
	}

	public PacketID(byte vn, byte t, byte dfhf, short a) {
		if (vn <= 7) versionNumber = vn;
		if ((t == 0) || (t == 1)) type = t;
		if ((dfhf == 0) || (dfhf == 1)) dataFieldHeaderFlag = dfhf;
		if (a <= 2047) apid = a;
	}

	public PacketID(byte t, short a) {
		versionNumber = 0;
		if ((t == 0) || (t == 1)) type = t;
		dataFieldHeaderFlag = 1;
		if (a <= 2047) apid = a;
	}

	public synchronized byte getType() {
		return type;
	}

	public synchronized short getApid() {
		return apid;
	}

	public synchronized short get() {
		short out = 0;
		out = versionNumber;
		out = (short)((out << 1) | type);
		out = (short)((out << 1) | dataFieldHeaderFlag);
		out = (short)((out << 11) | apid);

		return out;
	}
}
```

```
public class PacketSequenceControl {
        private byte sequenceFlags;
        private short sequenceCount;

        // The size of the packet sequence control in byte
        public static final byte BYTE_SIZE = 2;
        public static final byte BIT_SIZE = 16;

        public PacketSequenceControl() {
        }

        public PacketSequenceControl(byte sf, short sc) {
                if (sf <= 3) sequenceFlags = sf;
                if ((sc >= 0) && (sc < 16384)) sequenceCount = sc;
        }

        public PacketSequenceControl(short psc) {
                sequenceFlags = 3;

                if ((psc >= 0) && (psc < 16384)) sequenceCount = psc;
                else System.out.println("\nThe sequence count is just 14 bits long!\n");
                // proper error handling should be here
        }

        public synchronized short getSequenceCount() {
                return sequenceCount;
        }

        public synchronized short get() {
                short out = 0;
                out = sequenceFlags;
                out = (short)((out << 14) | sequenceCount);
                return out;
        }
}
```

```
public class PacketLength {
        private int length;

        public PacketLength() {
        }

        public PacketLength(int l) {
                if (l <= 65535) length = l;
        }

        public synchronized int get() {
                return length;
        }
}
```

```
// Partially taken from the JPUS project and OBOSS
// Jagun Kwon

public class TCPacket extends PUSPacket {

        // The packet data field for this telecommand
        private TCPacketDataField tcPacketDataField;

        public TCPacket() {
        }

        public TCPacket(short apid, short l) {
                super(new PUSHeader((byte)0, (byte)1,  (byte)1, apid,
                        (byte)3, (short)0, l));
                tcPacketDataField = new TCPacketDataField();
        }

        public TCPacket(short apid, short l, TCPacketDataField tcpdf) {
                super(new PUSHeader((byte)0, (byte)1, (byte)1, apid, (byte)3,
                        (short)0, l));
                tcPacketDataField = tcpdf;
        }

        public TCPacket(short apid, short pktsqccnt, short l,
                                        TCPacketDataField tcpdf) {
                super(new PUSHeader(new PacketID((byte)1, apid),
                                new PacketSequenceControl(pktsqccnt),
                                new PacketLength(l)));

                tcPacketDataField = tcpdf;
        }

        public synchronized PUSHeader getPacketHeader() {
                return super.getPacketHeader();
        }

        public synchronized TCPacketDataField getPacketDataField() {
                return tcPacketDataField;
        }


        // Telecommand verification is performed by Packet router
        // in my implementation


        public synchronized byte[] get() {
                byte[] out = new byte[super.getPacketHeader().get().length +
                        tcPacketDataField.get().length];

                for (int i=0; i<super.getPacketHeader().get().length; i++)
                        out[i] = super.getPacketHeader().get()[i];
                for (int i=0; i<tcPacketDataField.get().length; i++)
                        out[i+super.getPacketHeader().get().length] =
                                tcPacketDataField.get()[i];

                return out;
        }
}
```

```
// TCPacketDataField.java
// Part of the code taken from the JPUS project
// Jagun Kwon

public class TCPacketDataField {
	private TCDataFieldHeader tcDataFieldHeader;

	private ApplicationData applicationData;

	private Spare spare;

	private PacketErrorControl packetErrorControl;

	public TCPacketDataField() {
	}

	public TCPacketDataField(TCDataFieldHeader tcdfh) {
		tcDataFieldHeader = tcdfh;
	}

	public TCPacketDataField(TCDataFieldHeader tcdfh,
				ApplicationData app,
				Spare sp,
				PacketErrorControl pec) {
		this.tcDataFieldHeader = tcdfh;
		this.applicationData = app;
		this.spare = sp;
		this.packetErrorControl = pec;
	}

	public synchronized void setApplicationData(ApplicationData app) {
		this.applicationData = app;
	}

	public synchronized void setSpare(Spare sp) {
		this.spare = sp;
	}

	public synchronized void setPacketErrorControl(PacketErrorControl pec){
		this.packetErrorControl = pec;
	}

	public synchronized ApplicationData getApplicationData() {
		return applicationData;
	}

	public synchronized byte getAck() {
		return tcDataFieldHeader.getAck();
	}

	public synchronized byte getServiceType() {
		return tcDataFieldHeader.getServiceType();
	}

	public synchronized byte getServiceSubType() {
		return tcDataFieldHeader.getServiceSubtype();
	}

	public synchronized byte[] getSourceID() {
		return tcDataFieldHeader.getSourceID();
	}

	public synchronized byte[] get() {
		byte out[] = new byte[tcDataFieldHeader.get().length +
					applicationData.get().length +
						spare.getSpare().length +
					packetErrorControl.BIT_SIZE];

		for (int i=0; i< tcDataFieldHeader.get().length; i++)
			out[i] = tcDataFieldHeader.get()[i];
```

```
                for (int i=0; i< applicationData.get().length; i++)
                        out[i+tcDataFieldHeader.get().length] = applicationData.get()[i];
                for (int i=0; i< spare.getSpare().length; i++)
                        out[i+tcDataFieldHeader.get().length+applicationData.get().length]
                        = spare.getSpare()[i];

                out[tcDataFieldHeader.get().length +
                        applicationData.get().length +
                        spare.getSpare().length] =
                        (byte)(packetErrorControl.getPacketErrorControl() >> 8);

                out[tcDataFieldHeader.get().length +
                        applicationData.get().length +
                        spare.getSpare().length + 1] =
                        (byte)(packetErrorControl.getPacketErrorControl());

                return out;
        }
}
```

```
// Part of the code is taken from the JPUS project and OBOSS II
// Jagun Kwon

public class TCDataFieldHeader {
	private byte ccsdsSecondaryHeaderFlag;
	private byte tcPacketPusVersionNumber;
	private byte ack;
	private byte serviceType;
	private byte serviceSubtype;
	private byte[] sourceId;
	private byte[] spare;

	public TCDataFieldHeader() {
		ccsdsSecondaryHeaderFlag = 0;
		tcPacketPusVersionNumber = 1;
	}

	public TCDataFieldHeader(byte cshf, byte tppv, byte a, byte st,
		byte ss, byte[] si, byte[] s) {
		if (cshf == 0) ccsdsSecondaryHeaderFlag = cshf;
		if (tppv <= 7) tcPacketPusVersionNumber = tppv;
		if (a <= 15) ack = a;
		if (st <= 255) serviceType = st;
		if (ss <= 255) serviceSubtype = ss;
		sourceId = si;
		spare = s;
	}

	public TCDataFieldHeader (byte a, byte st, byte ss, byte[] si) {
		ccsdsSecondaryHeaderFlag = 0;
		tcPacketPusVersionNumber = 1;
		if (a <= 15) ack = a;
		if (st <= 255) serviceType = st;
		if (ss <= 255) serviceSubtype = ss;
		sourceId = si;
	}

	public TCDataFieldHeader(byte[] dfh) {
		ack = (byte)(dfh[0] & 0x0f);
		tcPacketPusVersionNumber = (byte)((dfh[0] & 0x70) >> 4);
		ccsdsSecondaryHeaderFlag = (byte)((dfh[0] & 0x80) >> 7);
		serviceType = dfh[1];
		serviceSubtype = dfh[2];
	}

	// Returns the extyernal TC data field header
	public synchronized byte[] get() {
		int i;
		byte[] out = new byte[3 + sourceId.length + spare.length];

		for (i=0; i<out.length; i++) out[i] = 0;
		out[0] = ccsdsSecondaryHeaderFlag;
		out[0] = (byte)((out[0] << 3) | tcPacketPusVersionNumber);
		out[0] = (byte)((out[0] << 4) | ack);
		out[1] = serviceType;
		out[2] = serviceSubtype;

		for (i=0; i< sourceId.length; i++) out[3+i] = sourceId[i];
		for (i=0; i< spare.length; i++)
			out[3 + sourceId.length + i] = spare[i];
		return out;
	}

	public synchronized byte getAck() {
		return ack;
	}

	public synchronized byte getServiceType() {
		return serviceType;
	}
	public synchronized byte getServiceSubtype() {
		return serviceSubtype;
	}

	public synchronized byte[] getSourceID() {
		return sourceId;
	}
}
```

```
public class ApplicationData {
	private byte[] applicationData;

	public ApplicationData() {
		applicationData = null;
	}

	public ApplicationData(byte[] apd) {
		applicationData = apd;
	}

	public synchronized byte[] get() {
		return applicationData;
	}
}
```

```
public class Spare {
	private byte[] spare;

	public Spare() {
		spare = null;
	}

	public Spare(short spareLength) {
		spare = new byte[spareLength];

		for (int i=0; i<spareLength; i++)
			this.spare[i] = 0;
	}

	public synchronized byte[] getSpare() {
		return spare;
	}
}
```

```
public class PacketErrorControl {
	private short packetErrorControl;

	public static final byte BYTE_SIZE = 2;

	public static final byte BIT_SIZE = 16;

	public PacketErrorControl() {
		packetErrorControl = 0;
	}

	public PacketErrorControl(short pec) {
		packetErrorControl = pec;
	}

	public synchronized void setPacketErrorControl(short errorControl) {
		this.packetErrorControl = errorControl;
	}

	public synchronized short getPacketErrorControl() {
		return packetErrorControl;
	}
}
```

```
// Partially taken from the JPUS project and OBOSS
// Jagun Kwon

public class TMPacket extends PUSPacket {

        // The packet data field for this telecommand
        private TMPacketDataField tmPacketDataField;

        public TMPacket() {
        }

        public TMPacket(TMDataFieldHeader tmdfh) {
                super(new PUSHeader());
                tmPacketDataField = new TMPacketDataField(tmdfh);
        }

        public TMPacket(short apid, TMDataFieldHeader tmdfh) {
                super(new PUSHeader(new PacketID((byte)0, apid)));
                tmPacketDataField = new TMPacketDataField(tmdfh);
        }

        public TMPacket(short apid, short l, TMPacketDataField tmpdf) {
                super(new PUSHeader((byte)0, (byte)0, (byte)1, apid, (byte)3,
                        (short)0, l));
                tmPacketDataField = tmpdf;
        }

        public TMPacket(short apid, short l) {
                super(new PUSHeader((byte)0, (byte)0, (byte)1, apid, (byte)3,
                        (short)0, l));
                tmPacketDataField = new TMPacketDataField();
        }

        public TMPacket(short apid, byte[] dId, ServiceName sn, short length) {
                super(new PUSHeader((byte)0, (byte)0, (byte)1, apid, (byte)3,
                        (short)0, length));
                byte temp[] = {0};

                tmPacketDataField = new TMPacketDataField((byte)0, (byte)0,
                                (byte)0, sn.getType(), sn.getSubtype(),
                                (byte)0, dId, temp, temp, temp,
                                (short)0, (short)0);
        }

        public synchronized PUSHeader getPacketHeader() {
                return super.getPacketHeader();
        }

        public synchronized TMDataFieldHeader getDataFieldHeader() {
                return tmPacketDataField.getTmDataFieldHeader();
        }

        public synchronized byte[] get() {
                byte[] out = new byte[super.getPacketHeader().get().length +
                        tmPacketDataField.get().length];

                for (int i=0; i<super.getPacketHeader().get().length; i++)
                        out[i] = super.getPacketHeader().get()[i];
                for (int i=0; i<tmPacketDataField.get().length; i++)
                        out[i+super.getPacketHeader().get().length] =
                                tmPacketDataField.get()[i];

                return out;
        }
}
```

```java
// TMPacketDataField.java
// Part of the code taken from the JPUS project
// Jagun Kwon

public class TMPacketDataField {
	private TMDataFieldHeader tmDataFieldHeader;

	private SourceData sourceData;

	private Spare spare;

	private PacketErrorControl packetErrorControl;

	public TMPacketDataField() {
	}

	public TMPacketDataField(TMDataFieldHeader tdfh) {
		tmDataFieldHeader = tdfh;
	}

	public TMPacketDataField(TMDataFieldHeader tdfh,
			SourceData src, Spare spr,
			PacketErrorControl pec) {
		tmDataFieldHeader = tdfh;
		sourceData = src;
		spare = spr;
		packetErrorControl = pec;
	}

	public TMPacketDataField(byte s1, byte tmsppusvn, byte s2, byte st,
			byte ss, byte ps, byte[] did, byte[] t, byte[] s3,
			byte[] sd, short s, short pec) {
		tmDataFieldHeader = new TMDataFieldHeader(s1, tmsppusvn, s2,
				st, ss, ps, did, t, s3);
		sourceData = new SourceData(sd);
		spare = new Spare(s);
		packetErrorControl = new PacketErrorControl(pec);
	}

	public synchronized TMDataFieldHeader getTmDataFieldHeader() {
		return tmDataFieldHeader;
	}

	public synchronized byte[] get() {
		byte out[] = new byte[tmDataFieldHeader.get().length +
					sourceData.get().length +
					spare.getSpare().length +
					packetErrorControl.BIT_SIZE];

		for (int i=0; i< tmDataFieldHeader.get().length; i++)
			out[i] = tmDataFieldHeader.get()[i];
		for (int i=0; i< sourceData.get().length; i++)
			out[i+tmDataFieldHeader.get().length] =
				sourceData.get()[i];
		for (int i=0; i< spare.getSpare().length; i++)
			out[i+tmDataFieldHeader.get().length +
				sourceData.get().length] = spare.getSpare()[i];

		out[spare.getSpare().length] =
			(byte)(packetErrorControl.getPacketErrorControl() >> 8);

		out[spare.getSpare().length + 1] =
			(byte)(packetErrorControl.getPacketErrorControl());

		return out;
	}
}
```

```
// Part of the code is taken from the JPUS project and OBOSS II
// Jagun Kwon

//import EarthObservationMission.*;

public class TMDataFieldHeader {
	private byte spare1;
	private byte TmSourcePacketPusVersionNumber;
	private byte spare2;
	private byte serviceType;
	private byte serviceSubtype;
	private byte packetSubcounter;
	private byte[] destinationID;
	private byte[] time;
	private byte[] spare3;

	public TMDataFieldHeader() {
		spare1 = 0;
		TmSourcePacketPusVersionNumber = 1;
		spare2 = 0;
	}

	public TMDataFieldHeader(byte st, byte ss, byte ps, byte[] did) {
		this();
		if (st <= 255) serviceType = st;
		if (ss <= 255) serviceSubtype = ss;
		if (ps <= 255) packetSubcounter = ps;
		destinationID = did;
		time = setTime();
	}

	public TMDataFieldHeader (byte s1, byte tmsppusvn, byte s2,
			byte st, byte ss, byte ps, byte[] did,
			byte[] t, byte[] s3) {
		if (s1 <= 1) spare1 = s1;
		if (tmsppusvn <= 7) TmSourcePacketPusVersionNumber = tmsppusvn;
		if (s1 <= 15) spare2 = s2;
		if (st <= 255) serviceType = st;
		if (ss <= 255) serviceSubtype = ss;
		if (ps <= 255) packetSubcounter = ps;
		destinationID = did;
		time = t;
		spare3 = s3;
	}

	public TMDataFieldHeader (byte s1, byte tmsppusvn, byte s2,
			byte st, byte ss, byte ps, byte[] did) {
		if (s1 <= 1) spare1 = s1;
		if (tmsppusvn <= 7) TmSourcePacketPusVersionNumber = tmsppusvn;
		if (s1 <= 15) spare2 = s2;
		if (st <= 255) serviceType = st;
		if (ss <= 255) serviceSubtype = ss;
		if (ps <= 255) packetSubcounter = ps;
		destinationID = did;
	}

	public TMDataFieldHeader (byte[] dfh) {
		spare2 = (byte)(dfh[0] & 0x0f);
		TmSourcePacketPusVersionNumber = (byte)((dfh[0] & 0x70) >> 4);
		spare1 = (byte)((dfh[0] & 0x80) >> 7);
		serviceType = dfh[1];
		serviceSubtype = dfh[2];
		packetSubcounter = dfh[3];
	}

	public synchronized byte[] setTime() {
//		SatelliteClock clock = new SatelliteClock();
//		return clock.getTime();
		// To test
		return new byte[] {100};
```

```
        }

        // Returns the extyernal TC data field header
        public synchronized byte[] get() {
                int i;
                byte[] out = new byte[6 + time.length + spare3.length];

                for (i=0; i<out.length; i++) out[i] = 0;
                out[0] = spare1;
                out[0] = (byte)((out[0] << 3) | TmSourcePacketPusVersionNumber);
                out[0] = (byte)((out[0] << 4) | spare2);
                out[1] = serviceType;
                out[2] = serviceSubtype;
                out[3] = packetSubcounter;

                for (i=0; i< destinationID.length; i++)
                        out[4+i] = destinationID[i];
                for (i=0; i< time.length; i++)
                        out[4 + destinationID.length + i] = time[i];
                for (i=0; i< spare3.length; i++)
                        out[4 + destinationID.length + time.length + i] =
                                spare3[i];
                return out;
        }
}
```

```
public class SourceData {
        private byte[] sourceData;

        public SourceData() {
                sourceData = null;
        }

        public SourceData(byte[] sd) {
                sourceData = sd;
        }

        public synchronized byte[] get() {
                return sourceData;
        }
}
```

D.2. Classes for the Queue structure

```
// packetQueue.java
// by Jagun Kwon
//
// Real-Time Systems Research Group
// Department of Computer Science
// University of York
//

//import MissionParameters;

public class packetQueue {
        private byte [] packetPool;   // Pool for byte stream
        private packetElement [] packetElementPool;  // Pool for elem objects
        private long indToPkPool;     // Index to packetPool
        private long indToPkElemPool; // Index to packetElementPool
        private long poolSize;

        private Queue myQ;

        public packetQueue (int Qsize, long poolSize) {
                packetElementPool = new packetElement[Qsize];
                myQ = new Queue(packetElementPool);

//              if (poolSize <= globals.MAX_POOL_SIZE)
                packetPool = new byte[poolSize];
```

```
		for (int i=0; i<myQ.length; i++) {
			myQ[i].indexFrom = 0;
			myQ[i].indexTo = 0;
			myQ[i].availability = true;
			// First, let them all available to store packets
		}

		indToPkPool = 0;
		indToPkElemPool = 0;
		this.poolSize = poolSize;
	}

	public synchronized void deposit(PUSPacket elem) {
		if ((elem != null) &&
		   (packetElementPool[indToPkElemPool].isAvailable()) {

			for(int i=0;i<elem.get().length; i++)
				packetPool[indToPkPool + i] = elem.get()[i];

			packetElementPool[indToPkElemPool].makeUnavailable();
			indToPkPool += elem.get().length;

			myQ.deposit(packetElementPool[indToPkElemPool++]);
		}
		else {
			System.out.println("***  packetQueue:deposit:parameter  is  either
null or there is no available element in the element queue.\n");
			System.out.println("***  Increase  the  size  of  the  queue  in  the
latter case.");
			// May throw an exception here.
		}
	}

	public synchronized PUSPacket extract() {
		// Extract an element from the queue
		// Create a new PUSPacket object for applications to store
		// in their MA and use
		short tmpOutShort1;
		short tmpOutShort2;
		short tmpOutShort3;

		// Used to store apid
		tmpOutShort1 = ((packetElement)myQ.extract()[3] << 8) +
			(packetElement)myQ.extract()[4];

		// Used to store sequenceCount
		tmpOutShort2 = ((packetElement)myQ.extract()[6] << 8) +
			(packetElement)myQ.extract()[7];

		// Used to store packet length
		tmpOutShort2 = ((packetElement)myQ.extract()[8] << 8) +
			(packetElement)myQ.extract()[9];

		// Create PUSHeader object (to create a PUSPacket eventually)
		PUSHeader pkhd = new PacketID((packetElement)myQ.extract()[0],
			(packetElement)myQ.extract()[1],
			(packetElement)myQ.extract()[2],
			tmpOutShort1,
			(packetElement)myQ.extract()[5],
			tmpOutShort2,
			tmpOutShort3);

		// This packet could either be a TC packet or TM packet.
		// So depending on what it is, create an appropriate object
		// and return

		// *** How do we know from the header if a packet is a TC or TM?
		// if () {
		//	convertIntoPUSPacket(pkhd, );
		// }
```

```
                // else convertIntoPUSPacket(pkhd, );
        }

        private PUSPacket convertIntoPUSPacket(PUSHeader ph,
                TCDataFieldPacket tc) {
                // combine the two parts together
        }

        private PUSPacket convertIntoPUSPacket(PUSHeader ph,
                TMDataFieldPacket tc) {
                // combine the two parts together
        }
}
```

```
public class packetElement extends Element {

        private long indexFrom;
        private long indexTo;
        protected boolean availability;

        public packetElement() {
                indexFrom = 0;
                indexTo = 0;
                availability = true;
        }

        public synchronized byte[] get() {
                if (super.elem != null)
                        return super.elem;
                else return null;
        }

        public synchronized void set(byte[] elem) {
                if (availability != false)
                        super.elem = elem;
        }

        public synchronized boolean isAvailable() {
                return availability;
        }

        public synchronized void makeAvailable() {
                availability = true;
        }

        public synchronized void makeUnavailable() {
                availability = false;
        }
}
```

```
// Queue.java
//
// by Jagun Kwon
// Last Updated: 16/02/04 - added QueueEmptyException to Extract method

//package Basic_Services;

public class Queue
{
	Element[] myQueue;
	int insertIndex, extractIndex = 0;
	int currentSize = 0;
	int queueSize = globals.MAIN_QUEUE_SIZE;	// default queue size
	boolean barrier = false;
	boolean response = false;

	Element tmpElem;

	// Constructors
	public Queue (int Qsize) {
		if (Qsize > 0)
		{ queueSize = Qsize; }

		myQueue = new Element[queueSize];
	}

	public Queue (Element[] elem) throws QueueEmptyException {
		if ((elem.length > 0) && (elem != null)) {
			queueSize = elem.length;
			myQueue = elem;
		}
		else throw new QueueEmptyException();
	}

	public synchronized boolean deposit(Element Elem) {
		if (currentSize < queueSize)
		{
			myQueue[insertIndex] = Elem;
			insertIndex = (insertIndex + 1) % queueSize;
			currentSize = currentSize + 1;

			barrier = true;
			response = true;
		}
		else
		{
			response = false;
		}
		return response;
	}

	public synchronized Element extract() throws QueueEmptyException {
		if (!barrier)  // Throw the following exception
		               // when a thread attempts to extract
		               // an element from the queue that is
		               // empty at the time of request
		{throw new QueueEmptyException();}

		tmpElem = myQueue[extractIndex];
		extractIndex = (extractIndex + 1) % queueSize;

		currentSize = currentSize - 1;
		if (currentSize == 0)
		{
			barrier = false;
		}
		return tmpElem;
	}

	public boolean empty() {
		return (currentSize==0) ? true : false;
```

```
        }

        public boolean full() {
               return (currentSize==queueSize) ? true : false;
        }
}
```

```
// QueueEmptyException.java
// Exception class
//
// Jagun Kwon

// Exception indicating that a thread has attempted to extract an element from
// a queue that has no element at the time of the request.

//package Basic_Services;

class QueueEmptyException extends PUSException
{
        public QueueEmptyException() {super();}
        public QueueEmptyException(String s) {super(s);}
}
```

```
// Element.java
//
// by Jagun Kwon

//package Basic_Services;

public class Element
{
        Object elem;
}
```

Bibliography

[Alves-Foss1999a] J. Alves-Foss and D. Frincke, Formal Grammar for Java, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[Alves-Foss1999b] J. Alves-Foss and F. S. Lam, Dynamic Denotational Semantics of Java, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[Amme2001] W. Amme, N. Dalton, M. Franz, and J. Von Ronne, SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form, Accepted for the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation 2001.

[Appel1999] 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/, accessed in January 2001.

[Audsley1993] 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.

[Azevedo1999] A. Azevedo, A. Nicolau, and J. Hummel, Java Annotation-Aware Just-In-Time (AJIT) Compilation System, ACM 1999 Java Grande Conference, 1999.

[Bandera2005] Bandera Project, http://bandera.projects.cis.ksu.edu/, last accessed June 2005.

[Barnes1998] J. Barnes, High Integrity Ada: the SPARK approach, Addison Wesley, 1997.

[Barnes2003] J. Barnes, High Integrity Software: the SPARK approach to Safety and Security, Addison Wesley, 2003

[Bate2000] I. Bate, G. Bernat, G. Murphy, P. Puschner, Low-level analysis of a portable WCET analysis framework, 6th IEEE Real-Time Computing Systems and Applications (RTCSA), 2000.

[Beebee2001] W. S. Beebee Jr., Region-based Memory Management for Real-Time Java, Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2001.

[Beebee2001b] W. S. Beebee Jr., An Implementation of Scoped Memory for Real-Time Java, Proceedings of the 1st ACM Conference on Embedded Software (EMSOFT), 2001.

[Bentley1999] S. Bentley, The Utilisation of the Java Language in Safety Critical System Development, MSc dissertation, Department of Computer Science, University of York, 1999.

[Bernat2000] 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.

[Blanchet2003] B. Blanchet, Escape Analysis for Java: Theory and Practice, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 25, Issue 6, November 2003.

[Bollella2000a] G. Bollella, et al, The Real-Time Specification for Java, Addison-Wesley, 2000.

[Bollella2000b] G. Bollella and J. Gosling, The Real-Time Specification for Java, IEEE Computer, Vol. 33, No. 6, June 2000.

[Bowen1998] J. P. Bowen and M. G. Hinchey, High Integrity System Specification and Design, Springer-Verlag London, 1998.

[Boyapati2003] C. Boyapati, et al, Ownership types for safe region-based memory management in real-time Java, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, CA, USA, 2003.

[Brat2000a] G. Brat, K. Havelund, S. Park, and W. Visser, Java PathFinder – Second Generation of a Java Model Checker, In Proceedings of Post-CAV Workshop on Advances in Verification, Chicago, July 2000.

[Brat2000b] G. Brat, K. Havelund, S. Park, and W. Visser, Model Checking Programs, Proceedings of the IEEE International Conference on Automated Software Engineering (ASE), Sep. 2000.

[Brown1999] D. F. Brown and D. A. Watt, JAS: a Java Action Semantics, Proceedings of the 2nd International Workshop on Action Semantics (ed. Mosses, P.D., and Watt, D.A.), BRICS NS-99-3, University of Aarhus, Denmark, 1999.

[Brosgol2002] 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.

[Bruening1999] D.L.Bruening, Systematic Testing of Multithreaded Java Programs, Master's Thesis, EECS Department, Massachusetts Institute of Technology, May 21, 1999.

[Bruening2000] D.L. Bruening, J. Chapin, Systematic Testing of Multithreaded Programs, MIT/LCS Technical Memo, LCS-TM-607, MIT, USA, 2000.

[Burns1998] A. Burns, B. Dobbing, and G. Romanski, The Ravenscar Tasking Profile for High Integrity Real-Time Programs, Proceedings of Ada-Europe 98, LNCS, Vol. 1411, pages 263-275, Berlin Heidelberg, Germany, Springer-Verlag 1998.

[Burns2001] A. Burns, and A. Wellings, Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX, 3rd Edition, Addison Wesley, 2001.

[Cai2003] H. Cai, A. Wellings, Towards a High Integrity Real-Time Java Virtual Machine, Department of Computer Science, University of York, York, U.K. 2003.

[Cai2004] H. Cai, A. Wellings, A Real-time Isolate Specification for Ravenscar-Java, Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Vienna, Austria, May 12-14, 2004.

[Cai2005] H. Cai, A Virtual Machine for High Integrity Real-Time Systems, Ph.D thesis, Department of Computer Science, University of York, U.K., 2005.

[Carré1990] B. A. Carré, T. J. Jennings, F. J. Maclennan, P. F. Farrow, and J. R. Garnsworthy, SPARK – The SPADE Ada Kernel, 3rd ed, Program Validation Limited, 1990.

[Chapman1994] R. Chapman, A. Burns and A.J. Wellings, Integrated Program Proof and Timing Analysis of SPARK Ada, Proceedings of the ACM workshop on language, compiler and tool support for real-time systems, ACM Press, Walt Disney World, Florida, USA, June 1994.

[Checkstyle] Eclipse Checkstyle Plug-in, http://sourceforge.net/projects/eclipse-cs.

[Choi1999] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, Escape Analysis for Java, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.

[Corbett2000] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn L.H. Zheng, Bandera: Extracting Finite-state Models from Java Source Code, Proceedings of the 22nd International Conference on Software Engineering, June, 2000.

[Craigen1995] D. Craigen, M. Saaltink and S. Michell, Ada 95 Trustworthiness Study; A Framework for Analysis, ORA Canada, 29 November 1995.

[Cullyer1991] W. J. Cullyer, S. J. Goodenough, and B. A. Wichmann, The Choice of Computer Languages for use in Safety-Critical Systems, Software Engineering Journal, March 1991.

[Cytron1991] R. Cytron et al, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Transaction on Programming Languages and Systems (TOPLAS), Vol. 13, Issue 4, October 1991.

[Demartini1998] C. Demartini, R. Iosif, and R. Sisto, Modeling and Validation of Java Multithreading Applications using SPIN, Proceedings of the 4th Workshop on Automata Theoretic Verification with the SPIN model checker, pp 5-19, Paris, France, November 1998.

[Dibble2002] P. Dibble, Personal communications, October 2002.

[Dobbing2001] B. Dobbing, The Ravenscar Profile for High Integrity Java Programs?, ACM Ada Letters, Vol. 21, Issue. 1, March 2001.

[Drossopoulou1999] 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.

[Eclipse] Eclipse.org, http://www.eclipse.org/.

[ECSS-E-70-41A] Ground systems and operations –Telemetry and Telecommand Packet Utilization, ECSS-E-70-41A, Issue 1, 2003.

[ESA PSS-07-101] European Space Agency, Packet Utilisation Standard, ESA PSS-07-101 Issue 1, May 1994.

[Evans1994] D. Evans, J. Guttag, J. Horning, and Y.M.Tan, LCLint: A Tool for Using Specifications to Check Code, Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, 1994.

[FindBugs] FindBugs, IBM, http://www-128.ibm.com/developerworks/java/library/j-findbug1/.

[Flanagan2002] C. Flanagan, K.R.M. Leino, M. Lillibridge, G.Nelson, J.B.Saxe, R.Stata, Extended Static Checking for Java, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, 2002.

[Gong1999] Li Gong, Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation, Addison-Wesley, 1999.

[Gosling2000] J. Gosling, B. Joy, G. Steele, and G. Bracha, The Java Language Specification, 2nd Edition, Addison Wesley, 2000.

[Hartel2001] 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.

[Hatton1995] L. Hatton, Safer C: Developing for High-Integrity and Safety-Critical Systems

[Havelund1999a] K. Havelund, J. Skakkebaek, Applying Model Checking in JAVA Verification, LNCS 1680, pp 216-231, Springer-Verlag September 1999.

[Havelund1999b] K. Havelund and T. Pressburger, Model Checking Java Programs Using Java PathFinder, to appear in International Journal on Software Tools for Technology Transfer, 1999.

[Havelund1999c] K. Havelund, JavaPathfinder, A Translator from Java to Promela, LNCS 1680. Springer-Verlag, pp 152, September1999.

[Hetcht2002] 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, also available at http://fermi.sohar.com/J1030/index.htm, last accessed in January 2002.

[Henriksson1998] R. Henriksson, Scheduling Garbage Collection in Embedded Systems, Ph.D thesis, Department of Computer Science, Lund University, 1998.

[Holzmann1997] G.J. Holzmann, The Model Checker SPIN, IEEE Transactions on Software Engineering, 23(5): 279-295, 1997.

[Holzmann2000] G.J. Holzmann, E.Najm, and A. Serhrouchni, SPIN Model Checking: An Introduction, Internal Journal on Software Tools for Technology Transfer (STT), vol. 2, num. 4, Springer LINK, 2000

[Holzmann2002] G.J. Holzmann, Software Analysis and Model Checking, Proceedings of the 14th International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science 2404, Springer, 2002.

[Hu2003] E. Y-S. Hu, J. Kwon and A. Wellings, XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment, Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), 2003.

[Hutcheon1992] 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.

[Ippolito1995] L. M. Ippolito and D. R. Wallace, A Study on Hazard Analysis in High Integrity Software Standards and Guidelines, Gaithersburg, MD: U.S. Dept. of Commerce, National Institute of Standards and Technology, NISTIR 5589, http://hissa.ncsl.nist.gov/index-pubs.html, published in 1995.

[Isaksen1997] U. Isaksen, J.P. Bowen, and N. Nissanke, System and Software Safety in Critical Systems, RUCS Technical Report, RUCS/97/TR/062/A, The University of Reading, Department of Computer Science, Reading, UK. 1997.

[ISO/IEC 9899:1990] Programming Languages – C, ISO/IEC 9899:1990, ISO/IEC, 1990.

[ISO/IEC DTR 15942] Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems, ISO/IEC DTR 15942, ISO/IEC WG9, 1999.

[JConsortium2000] J Consortium, International J Consortium Specification: Real-Time Core Extensions, Revision 1.0.14, www.jconsortium.org, September 2000.

[JML] The Java Modeling Language, http://www.cs.iastate.edu/~leavens/JML/.

[JPF2005] Java PathFinder, http://ti.arc.nasa.gov/ase/jpf/index.html, last accessed in Nov. 2005.

[JSR121] JSR 121: Application Isolation API Specification, http://www.jcp.org/en/jsr/detail?id=121 .

[JSR133] JSR 133: Java Memory Model and Thread Specification Revision, http://jcp.org/en/jsr/detail?id=133 .

[JSR175] JSR 175: A Metadata Facility for the JavaTM Programming Language, http://www.jcp.org/en/jsr/detail?id=175 .

[JSR176] JSR 176: J2SETM 5.0 (Tiger) Release Contents, http://www.jcp.org/en/jsr/detail?id=176 .

[Kernighan1988] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall Inc., 1988.

[Kim1999] T. Kim, N. Chang, N. Kim, H. Shin, Scheduling Garbage Collector for Embedded Real-Time Systems, In Proceedings of the LCTES '99, ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, 1999.

[Kozen1999] D. Kozen, Language Based Security, Technical Report TR99-1751, Cornell University, 1999.

[Kwon2002a] J. Kwon, A. Wellings, S. King, Ravenscar-Java: A High Integrity Profile for Real-Time Java, Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, 2002.

[Kwon2002b] J. Kwon, A. Wellings, and S. King, Assessment of the Java Programming Language for Use in High Integrity Systems, 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.

[Kwon2003a] J. Kwon, A. Wellings and S. King, Predictable Memory Utilization in the Ravenscar-Java Profile, published in the Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Hakodate, Hokkaido, Japan, 2003.

[Kwon2003b] J. Kwon, A. Wellings and S. King, Assessment of the Java Programming Language for Use in High Integrity Systems, published in ACM SIGPLAN Notices, April, 2003.

[Kwon2004] J. Kwon, A. Wellings, Memory Management base on Method Invocation in RTSJ, published in Lecture Notes in Computer Science (LNCS) 3292, Proceedings of the OTM 2004 Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES) Oct. 2004.

[Kwon2005] J. Kwon, A. Wellings and S. King, Ravenscar-Java: A High Integrity Profile for Real-Time Java, published in the Journal of Concurrency and Computation: Practice and Experience, John Wiley & Sons Ltd (2005; 17:681-713) Feb. 2005.

[Laprie1992] J.C. Laprie (Ed.), Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer-Verlag, 1992.

[Larsen1997a] K.G. Larsen, P. Petterson, and W. Yi, Uppaal: Status & Developments, Proceedings of the 9th International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science, Springer, 1997.

[Larsen1997b] K.G. Larsen, P. Pettersson, W. Yi, UPPAAL in a Nutshell, International Journal on Software Tools for Technology Transfer, vol. 1, num. 1-2, pages 134-152, 1997.

[Larsen2001] K.G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, J. Romijn, As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata, Proceedings of the 13th International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science, Springer, 2001.

[Leino2000] 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. http://www.research.compaq.com/SRC/esc/papers.html.

[Leveson1986] N. G. Leveson, Software Safety: Why, What, and How, Computing Surveys, Vol. 18, No. 2, ACM, June 1986.

[Leveson1991] N. G. Leveson, Software Safety in Embedded Computer Systems, Communications of the ACM, Vol. 34, No. 2, February 1991.

[Leveson1995] N. G. Leveson. Safeware: System Safety and Computers, Addison-Wesley, 1995.

[Lindholm1999] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Second Edition, Addison-Wesley 1999.

[Liu1973] C. Liu and J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment, Journal of ACM, 20(1), 46-61, 1973.

[Manson2005] J. Manson, W. Pugh, and S.V. Adve, The Java Memory Model, Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL), Jan. 2005.

[Merri2002] M. Merri, B. Melton, S. Valera, A. Parkes, The ECCS Packet Utilization Standard and its Support Tool, AIAA International Conference on Space Operations, http://www.aiaa.org/Spaceops2002Archive/papers/SpaceOps02-P-T5-06.pdf, 2002.

[MIRA1998] The Motor Industry Software Reliability Association, Guidelines for the use of the C language in vehicle based software, The Motor Industry Research Association (MIRA), 1998.

[MISRA1994] The Motor Industry Software Reliability Association, Development Guidelines for Vehicle Based Software, ISBN 0952415607, MIRA Ltd., November 1994.

[MISRA1995a] The Motor Industry Software Reliability Association, Report 1: Diagnostics and Integrated Vehicle Systems, MIRA Ltd., February 1995.

[MISRA1995b] The Motor Industry Software Reliability Association, Report 2: Integrity, MIRA Ltd., February 1995.

[MISRA1995c] The Motor Industry Software Reliability Association, Report 3: Noise, EMC and Real-Time, MIRA Ltd., February 1995.

[MISRA1995d] The Motor Industry Software Reliability Association, Report 4: Software in Control Systems, MIRA Ltd., February 1995.

[MISRA1995e] The Motor Industry Software Reliability Association, Report 5: Software Metrics, MIRA Ltd., February 1995.

[MISRA1995f] The Motor Industry Software Reliability Association, Report 6: Verification and Validation, MIRA Ltd., February 1995.

[MISRA1995g] The Motor Industry Software Reliability Association, Report 7: Subcontracting of Automotive Software, MIRA Ltd., February 1995.

[MISRA1995h] The Motor Industry Software Reliability Association, Report 8: Human Factors in Software Development, MIRA Ltd., February 1995.

[NUREG/CR-6463] H. Hetcht, M. Hecht, S. Graff, et at, Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997, also available at http://fermi.sohar.com/J1030/index.htm, last accessed in 2004.

[Parnas1990] 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.

[PERENNIAL2001] JETS, A Perennial Validation Suite for the Java language, http://www.peren.com/pages/jets_set.htm, last accessed in December 2001.

[Pizlo2004] F. Pizlo, et al, Real-Time Java Scoped Memory: Design Patterns and Semantics, Proceedings of the 7th IEEE Intl., Symposium on Object-Oriented Real-Time Distribute Computing (ISORC'04), Vienna, Austria, May 2004.

[Pugh1999] W. Pugh, Fixing the Java Memory Model, Proceedings of Java Grande Conference, 1999.

[Pugh2005] W. Pugh, The Java Memory Model, http://www.cs.umd.edu/~pugh/java/memoryModel/ .

[Puschner2001a] 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.

[Puschner2001b] P. Puschner, G. Bernat, WCET Analysis of Reusable Portable Code, Proceedings of the 13th Euromicro International Conference on Real-Time Systems, 2001.

[Roychoudhury2002] A. Roychoudhury and T. Mitra, Specifying Multithreaded Java Semantics for Program Verification, Proceedings of the International Conference on Software Engineering – ICSE, 2002.

[RTSJ2005] The Real-Time Specification for Java, http://www.rtsj.org.

[Saaltink1997a] M. Saaltink and S. Michell, Ada 95 Trustworthiness Study; Analysis of Ada 95 for Critical Systems, V2.0, ORA Canada, 27 March 1997.

[Saaltink1997b] M. Saaltink and S. Michell, Ada 95 Trustworthiness Study; Guidance on the Use of Ada95 in the Development of High Integrity Systems, V2.0, ORA Canada, 27 March 1997.

[Salcianu2001] A. Salcianu and M. Rinard, Pointer and Escape Analysis for Multithreaded Programs, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2001.

[Savage1997] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A Dynamic Data Race Detector for Multithreaded Programs, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[Schoeberl2004] M. Schoeberl Restrictions of Java for Embedded Real-Time Systems, Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Vienna, Austria, May 12-14, 2004.

[Siebert2002] F. Siebert, Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages, aicas GmbH, available at http://www.aicas.com/books.html, 2002.

[Sommerville2000] I. Sommerville, Software Engineering, 6th Edition, Addison Wesley, 2000.

[SOOT] Soot: a Java Optimization Framework, http://www.sable.mcgill.ca/soot/.

[Stoller2000] S. Stoller, Model-checking multi-threaded distributed Java programs, Technical Report 536, Computer Science Dept., Indiana University, 2000.

[Storey1996] N. Storey, Safety-Critical Computer Systems, Addison-Wesley, 1996.

[Stroustrup1997] B. Stroustrup, The C++ Programming Language, 3rd Edition, Addison Wesley, 1997.

[Sun1999] Sun Microsystems, Code Conventions for the Java Programming Language, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html, written in April 1999, last accessed in December 2004.

[Sun2003] Sun Microsystems, JSR 175: A Metadata Facility for the Java Programming Language, available at http://www.jcp.org/en/jsr/detail?id=175.

[Sun2006] Sun Microsystems, The Connected Device Configuration HotSpot Implementation, http://java.sun.com/j2me/docs/cdc_hotspotds.pdf, last accessed March 2006.

[Terma2004] Terma, Onboard Operations Support Software (OBOSS III), http://spd-web.terma.com/Projects/OBOSS/Home_Page/, released in 2004.

[TimeSys2002] TimeSys™, Products and Services: Real-Time Java, available at http://www.timesys.com/rtj/index.html, last accessed in January 2002.

[UKMoD1991] U.K. Ministry of Defence, The Procurement of Safety Critical Software in Defence Equipment, INTERIM Defence Standard 00-55 (PART 1: REQUIREMENTS)/Issue 1, 5 April 1991.

[USDoD1978] U.S. Department of Defence, Requirements for High Order Computer Programming Languages "STEELMAN", U.S. Department of Defence, 1978.

[USDoD1990] U.S. Department of Defence, Ada 9X Requirements, Office of the Under Secretary for Defence Applications, Washington, D.C., December 1990.

[Venners1999] B. Venners, Inside the Java Virtual Machine, 2nd Edition, McGraw Hill, 1999.

[Watt2000] D. A. Watt and D. F. Brown, Formalising the Dynamic Semantics of Java, In Proceedings of the Third International Workshop on Action Semantics (AS2000), Recife, Brazil, May 2000.

[Wellings2004a] A. Wellings, G. Bollella, P. Dibble, D. Holmes, Cost Enforcement and Deadline Monitoring in the Real-Time Specification for Java, Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Vienna, Austria, May 12-14, 2004.

[Wellings2004b] A. Wellings, Concurrent and Real-Time Programming in Java, Wiley, 2004.

[Wheeler1997] D.A. Wheeler, Ada, C, C++, and Java vs. the Steelman, ACM Ada Letters, Vol. 17, Issue 4, July 1997.

[Wichmann1992] Software in Safety Related Systems, NPL, Wiley, 1992.

[Zhao2004] Tian Zhao; Noble, J.; Vitek, J, Scoped types for real-time Java, Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International 5-8 Dec. 2004 Page(s):241 – 251.