

Technical University of Denmark



Safety-Critical Java for Embedded Systems

Rios Rivas, Juan Ricardo; Schoeberl, Martin

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

[Link back to DTU Orbit](#)

Citation (APA):
Rios Rivas, J. R., & Schoeberl, M. (2014). Safety-Critical Java for Embedded Systems. Kgs. Lyngby: Technical University of Denmark (DTU). (IMM-PhD-2014; No. 340).

DTU Library

Technical Information Center of Denmark

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Safety-Critical Java for Embedded Systems

Juan Ricardo Rios Rivas



Kongens Lyngby 2014
IMM-PhD-2014-340

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-PhD-2014-340

Summary (English)

Safety-critical systems are real-time systems whose failure can have severe or catastrophic consequences, possibly endangering human life. Many safety-critical systems incorporate embedded computers used to control different tasks. Software running on safety-critical systems needs to be certified before its deployment and the most time-consuming step of this process is the testing and verification phase. Due to the increasing complexity in safety-critical systems there is a need for new technologies that can facilitate testing and verification activities.

The safety-critical specification for Java aims at providing a reduced set of the Java programming language that can be used for systems that need to be certified at the highest levels of criticality. Safety-critical Java (SCJ) restricts how a developer can structure an application by providing a specific programming model and by restricting the set of methods and libraries that can be used. Furthermore, its memory model do not use a garbage-collected heap but scoped memories.

In this thesis we examine the use of the SCJ specification through an implementation in a time-predictable, FPGA-based Java processor. The specification is now in a mature state and with our implementation we have proved its feasibility in an embedded platform. Moreover, we have explored how simple hardware extensions can reduce the execution time of time-critical operations required by the SCJ specification.

The scoped memory model used in SCJ is perhaps one of its most difficult features to use correctly. Therefore, in this work we have also studied practical aspects of its usage by developing scoped memory use patterns and reusable libraries aiming at facilitating the development of complex software systems.

Summary (Danish)

Sikkerhedskritiske systemer er realtidssystemer. Hvis sådanne systemer fejler, kan det have alvorlige eller katastrofale konsekvenser, muligvis livsfarlige konsekvenser. Mange sikkerhedskritiske systemer inkorporerer embeddede computere, som bruges til at kontrollere forskellige opgaver. Software, som kører på sikkerhedskritiske systemer, skal certificeres, før softwaren udrulles, og det mest tidskrævende skridt i denne proces er test- og verifikationsfasen. På grund af den stigende kompleksitet i sikkerhedskritiske systemer er der behov for nye teknologier til at facilitere test- og verifikationsaktiviteter.

Den sikkerhedskritiske specifikation for Java har til formål at levere en reduceret udgave af Java-programmeringssproget, som kan bruges til systemer, der skal kunne certificeres på de højeste kritikalitetsniveauer. Safety Critical Java (SCJ) begrænser, hvordan en udvikler kan strukturere en applikation, ved at levere en specifik programmeringsmodel og ved at begrænse de metoder og biblioteker, som kan anvendes. Derudover benytter hukommelsesmodellen i SCJ ikke en heap, der er garbage-collected, men afgrænsede hukommelser.

I denne afhandling undersøger vi brugen af SCJ-specifikationen gennem en implementering i en tidsforudsigelig, FPGA-baseret Java-processor. Specifikationen er nu på et modent stadie, og med vores implementering har vi demonstreret dens funktionsduelighed i en embedded platform. Desuden har vi undersøgt, hvordan simple hardware-tilføjelser kan reducere udførelsestiden for tidskritiske handlinger, som er påkrævet af SCJ-specifikationen.

Den afgrænsede hukommelsesmodel, som anvendes i SCJ, er måske en af de funktioner, som er allersværest at anvende korrekt. Vi har derfor også kigget på de praktiske aspekter af modellens brug ved at udvikle afgrænset-hukommelse brugsmønstre og genbrugelige biblioteker med henblik på at facilitere udviklingen af komplekse softwaresystemer.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D degree in Computer Science.

The thesis deals with the topic of Java for safety-critical embedded systems. We provide an open-source implementation of the safety-critical Java (SCJ) profile on top of a time-predictable Java processor. In addition, we report in our experience while implementing the profile and analyze the impact of SCJ's memory model on the development of applications and libraries.

The thesis subsumes the work done during the Ph.D course work undertaken from 2011 to 2014.

Juan Ricardo Rios Rivas

Acknowledgements

I would like to thank my supervisor Martin Schoeberl for his advice and support, especially during the last phase of the PhD program, for his very useful comments, suggestions, and insights on my work, and also for the opportunity to work in this project.

I would also like to thank my friends Jorge, Olivier, Hector, and Octavian who became my family in Denmark and made my life a lot more easier and full of enjoyable moments. Also, special thanks goes to the Borger-Winther family who opened their doors to Keyla and me during one of the toughest moments of our stay here in Denmark and for showing us a completely different side of the danish culture and people.

I am extremely grateful to my parents, my brothers, and my beloved Keyla, for their unconditional love and support throughout this journey. Without your support and encouragement this work for sure will not have been completed.

I thank God for taking care of me on every moment of my life, even in the wrong decisions and for showing me how perfect His timing really is.

Contents

Summary (English)	i
Summary (Danish)	iii
Preface	v
Acknowledgements	vii
1 Introduction	1
2 Background and Related Work	5
2.1 Real-time and Safety-critical Systems	6
2.2 Programming Languages for Safety-critical Systems	9
2.3 Java for Real-time Systems	11
2.3.1 Java for High-integrity Real-time Systems	14
2.4 Safety-critical Java Specification (JSR-302)	16
2.4.1 Execution Model	16
2.4.2 Concurrency	17
2.4.3 Memory Model	17
2.4.4 An SCJ HelloWorld Example	19
2.5 Java Optimized Processor JOP	21
2.6 High Integrity Java Profiles on Embedded Systems	24
2.6.1 Ravenscar-Java in the aJ-100 Processor	25
2.6.2 oSCJ	26
2.6.3 SCJ on HVM	27
2.6.4 Predictable Java	28
2.6.5 Cyclic Executive for SCJ on Chip-multiprocessors	29
2.7 Reference Assignment Checks	29
2.8 Scoped Memory Use	31

2.8.1	Patterns	31
2.8.2	Libraries	33
2.9	Testing Real-time Features in Real-time Virtual Machines	34
2.10	Summary	35
3	Safety-critical Java on an Embedded Java Processor	37
3.1	Overview	38
3.1.1	Design Decisions	38
3.1.2	Limitations	40
3.1.3	Building and Running an SCJ Application in JOP	41
3.2	Package Crossing	42
3.3	Concurrency and Scheduling	43
3.3.1	Missions and Mission Sequencer	43
3.3.2	Periodic and Aperiodic Event Handlers	48
3.3.3	Scheduler	52
3.3.4	Multi-core Support	54
3.4	Memory	55
3.5	Scope Checks	57
3.5.1	Referential Integrity and the Scope Stack	58
3.5.2	Detecting Illegal Reference Assignments	60
3.5.3	Evaluation	64
3.5.4	Discussion	65
3.6	Interaction with Devices and External Events	66
3.6.1	Raw Memory	67
3.6.2	Managed Interrupts	70
3.7	Summary	73
4	Scoped Memory Use: Patterns and Reusable Libraries	75
4.1	Use Patterns and Idioms	76
4.1.1	The Basic Pattern	76
4.1.2	Loop Pattern	77
4.1.3	Execute with Primitive Return Value	78
4.1.4	Returning a Newly Allocated Object	79
4.1.5	Scoped Methods	81
4.1.6	Runnable Factory	81
4.1.7	Producer/Consumer	84
4.2	Reusable Libraries: Issues and Solutions	84
4.2.1	Lazy Initialization	84
4.2.2	Dynamic Resizing	85
4.2.3	Objects Used in Mixed Contexts	88
4.2.4	Iterators	89
4.2.5	Loop Bounds	89
4.2.6	Exceptions	90
4.3	Reusable Libraries: Implementation	91

4.3.1	Analysis of Standard Java Class Libraries	93
4.3.2	<code>AbstractStringBuilder</code> and <code>StringBuilder</code>	94
4.3.3	<code>DataInputStream</code>	95
4.3.4	<code>Vector</code> and <code>HashMap</code>	95
4.3.5	Comparison with JCL	97
4.3.6	Testing	99
4.3.7	Discussion	102
4.4	Summary	106
5	Evaluation	107
5.1	Microbenchmarks	108
5.1.1	Accuracy of Periods	108
5.1.2	Linear-time Memory Allocation Time	110
5.1.3	Aperiodic Event Handling	111
5.1.4	Dispatch Latency for Interrupts	117
5.1.5	Context Switch Latency	119
5.1.6	Synchronization	120
5.1.7	Discussion	121
5.2	SCJ's TCK and miniCDj	122
5.3	Summary	125
6	Conclusions	127
6.1	Main Results	127
6.2	Lessons Learned	131
6.3	Future Work	133
	Bibliography	137

List of Figures

2.1	Relation of domain-specific second tier guidelines to IEC-61508 safety standard	7
2.2	Example of an SCJ <code>HelloWorld</code> program	20
2.3	Application build tool-chain of JOP	22
2.4	Object and array layout in JOP	23
3.1	Overview of SCJ's concurrency classes implemented in JOP . . .	39
3.2	Overview of SCJ's memory classes implemented in JOP	39
3.3	An example of how class-private information is shared using Sun's <code>SharedSecrets</code> class	44
3.4	An example of how package-protected information is shared using our singleton delegator	45
3.5	Mission change timing for SCJ L0 and L1 applications	47
3.6	An extract of the SCJ's <code>PeriodicEventHandler</code> class implementation in JOP.	49
3.7	An extract of the SCJ's <code>AperiodicEventHandler</code> class implementation in JOP.	51
3.8	Location of JOP's scheduling run-time structures within SCJ's memory areas	54
3.9	Overview of the scoped memory layout in JOP.	56
3.10	Scoped memory hierarchy and a possible scope nesting level assignment	59
3.11	Example of three implementations of raw memory accessor objects. .	69
3.12	An example of using the <code>RawMemory</code> API to provide low level access to an IO device.	70
3.13	Example of an I2C bus controller implemented using raw memory. .	71
3.14	An extract of the <code>ManagedInterruptServiceRoutine</code> class implementation in JOP.	72

4.1	The loop pattern.	78
4.2	Execute with primitive return value.	79
4.3	Example of the use of <code>executeInOuterArea(Runnable logic)</code> to create objects in outer nested scopes.	80
4.4	Scoped method with parameters and a return object.	82
4.5	Runnable factory.	83
4.6	Effects of using shared objects from different scopes	87
4.7	Example of a method modified to run in a nested private scope. .	96
4.8	Class hierarchy of the modified <code>Vector</code> class.	98
5.1	Precision of periods	108
5.2	Linear time memory allocation time	110
5.3	Aperiodic load, events serviced, and deadlines missed for the 69% periodic load	114
5.4	Aperiodic load, events serviced, and deadlines missed for the 88% periodic load	115
5.5	Aperiodic load and events serviced with multiple AEHs	116
5.6	Expected execution pattern of the task set used to test the correct PCE implementation	120

List of Tables

3.1	Valid referenc assignments in SCJ	60
3.2	Execution time of the three bytecodes implementing scope checks	64
3.3	Sensor application execution time including the three versions of the scope checks.	65
3.4	N-Body simulation application execution time including the three versions of the scope checks.	66
4.1	List of library classes allowed by SCJ	92
4.2	Comparison of the implemented classes with JDK’s implementation	100
4.3	Number of modified methods and additional methods in the mod- ified classes	101
5.1	Measured PEH start time deviations	109
5.2	Measured scoped memory allocation times	111
5.3	Task sets used for the aperiodic event handling tests	113
5.4	Number of interrupts serviced when measuring the interrupt dis- patch latency.	118
5.5	Measured interrupt dispatch latency.	119
5.6	Measured context switch latency times	119
5.7	Task set for synchronization test	120
5.8	Measured timing values of synchronization task set	121
5.9	TCK tests implementation and successfully passed	123
5.10	miniCDj benchmark execution time measurements	125

CHAPTER 1

Introduction

Real-time systems are those systems whose correct operation depends not only on producing valid computation results but also in delivering those results within specified deadlines. The correct operation of a real-time system can be characterized by the ability of the system to meet those deadlines.

Real-time systems can be classified according to how useful the result of a computation is in the boundaries of its deadline. If the value of a result past its deadline is zero, then the system is said to be either *firm* or *hard* real-time. Hard real time systems are further classified as *mission-critical*, if a failure of the system makes impossible the completion of the system's goals but human life will not be endangered; and *safety-critical*, if a failure of the system may endanger human life.

Modern safety-critical systems are heavily dependent on embedded computers or networks of computers with specialized peripherals [66]. As demands for integrating more functionality in safety-critical systems increase, complexity also increases thereby demanding new technologies and programming languages to ease its development, maintenance, and certification process.

Java, as a programming language and as a large collection of standard libraries, has proven to be a useful tool to increase programmer efficiency both for the development and maintenance of software [47]. The benefits provided by Java

come from its high level object-oriented programming features and the use of an automatic garbage collector (GC).

In recent years, the use of Java for systems with real-time constraints has been enabled through the definition of the real-time specification for Java (RTSJ) [24] which specifies how real-time behavior can be achieved with Java. The RTSJ tightens loosely defined aspects of the Java language and enhances other several areas such as memory management through a strategy that avoids the use of an automatic GC. Avoidance of GC interference is particularly useful on applications with hard real-time requirements which are difficult to achieve with current GC technologies [95]. Without a GC, memory allocations are performed in specific regions called scoped memories and objects are collectively deallocated at scope exit.

The RTSJ is however too large for resource-constrained embedded systems and too complex for safety-critical systems where a rigorous process of certification must be followed. In contrast, the safety-critical Java (SCJ) profile, being developed under the Java specification request 302 (JSR-302), provides a smaller, tightly defined subset of the Java programming language intended for applications that need to be certified. SCJ uses a programming model that limits how a developer can structure an application. SCJ's programming model is more restricted in terms of concurrency, memory, synchronization, input and output, clocks and timers, and exception processing [77]. Three levels of compliance with different degrees of complexity are defined, namely Level 0 (L0), Level 1 (L1), and Level 2 (L2), L2 being the most complex.

In this thesis we examine the use of the safety-critical Java profile for embedded systems through an implementation on a time-predictable Java processor. We also report on our experiences while implementing the profile, explore hardware extensions for time-critical operations, and examine the impact of the scoped memory model of SCJ on the development of applications and reusable libraries for safety-critical Java. In more detail, our contributions are:

- *Implementation of safety-critical Java on a Java processor:*

As the SCJ specification matures, implementations are emerging and the need to evaluate those implementations in a systematic way is becoming important. To the best of our knowledge, currently there are only two implementations of the SCJ profile that targets embedded systems: (1) the oSCJ [96] implementation running on the Open Virtual Machine [15], and (2) the Hardware-near Virtual Machine (HVM) [121] implementation.

In this work, we provide an open-source implementation and evaluation of SCJ in the context of the Java Optimized Processor (JOP) [113]. We

also present some implementation issues and solutions that arise from the Java package crossing of different classes and from SCJ's scoped memory model.

Although the implementation is targeted for JOP, it can also be executed on a standard PC using JOP's software simulator. Therefore the implementation can also be used to evaluate experimental SCJ features or proposed changes that are not part of the specification.

- *Hardware extensions for scope checks:*

Without a GC, application developers have to be aware of where objects are allocated, thus increasing the risk of leaving dangling references. Referential integrity has to be guaranteed by following a set of rules that have to be enforced on every reference assignment which is a process that adds an extra overhead to the execution time of an application.

Given the simplified memory model of SCJ, we examine how a single scope nesting level can be used to check the legality of every reference assignment. We also show that with simple hardware extensions we can check reference assignments without the overhead of a software-based solution and improve the execution time of applications with frequent cross-scope references. This proposal was implemented and tested on JOP.

- *Exploration of the expressiveness of SCJ memory model:*

Explicit scoping requires care from programmers when dealing with temporary objects, passing scope-allocated objects as arguments to methods, and returning scope-allocated objects from methods. Moving data between scopes require a creative way of using the available SCJ memory API features. This API contains methods used to change the allocation context and safely create the returned objects.

The expressive power and ease of use of the SCJ memory model is explored. A collection of memory use patterns for the development of SCJ applications is presented. These patterns have different levels of complexity depending on the situations where they can be applied. The main focus while developing the patterns was on how to safely pass arguments and return objects between private memories.

- *Libraries for safety-critical Java:*

SCJ's memory model complicates the use of the standard Java class libraries (JCL) in application development because the libraries were developed under the assumptions of having a GC and unrestricted references between objects. Therefore it is important to find ways to modify standard libraries, so they work well with the SCJ memory model.

We inspect some of the most common programming patterns and idioms present in OpenJDK6's standard JCL. We identify patterns and idioms

which are problematic on SCJ applications and we present different ways to mitigate their impact. In addition, we create and test a total of five scope-safe classes representative of three of the most commonly used libraries: `java.util`, `java.lang`, and `java.io`. The focus was to minimize changes and modifications with respect to the original implementation.

This thesis is organized as follows: Chapter 2 provides the necessary background on safety-critical systems from a certification point of view and related work on Java for embedded real-time and safety-critical systems. Chapter 3 describes the details of the implementation of the safety-critical Java profile in the context of a Java processor. Chapter 4 provides a study on SCJ's memory model from a users perspective and is divided in two sections: (1) scoped memory use patterns and (2) reusable libraries for safety-critical Java. Chapter 5 presents an evaluation of our SCJ's implementation where we divided the tests in two categories: (1) performance and timeliness and (2) compliance to the SCJ specification. We conclude in Chapter 6 with a perspective and future work.

Publications

- Juan Rios and M. Schoeberl. Reusable Libraries for Safety-Critical Java. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 188–197, June 2014
- Juan Rios and M. Schoeberl. An Evaluation of Safety-Critical Java on a Java Processor. In *Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), 2014 IEEE/IFIP 10th Workshop on*, pages 276 – 283, June 2014
- Juan Ricardo Rios, Kelvin Nilsen, and Martin Schoeberl. Patterns for safety-critical Java memory usage. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 1–8, 2012
- Juan Rios and M. Schoeberl. Hardware Support for Safety-Critical Java Scope Checks. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 31–38, April 2012
- Martin Schoeberl and Juan Ricardo Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 54–61, Copenhagen, Denmark, 2012. ACM

CHAPTER 2

Background and Related Work

This chapter is intended to provide a general overview and background on the topics of safety-critical systems, Java for real-time and safety-critical systems, and the Java processor JOP. The chapter is organized as follows: in Section 2.1 we provide basic concepts about real-time and safety-critical systems. In Section 2.2 we present a survey of programming languages and language subsets used in the development of software for safety-critical systems. In Section 2.3 we look into the topic of using Java for real-time and safety-critical systems. In Section 2.4 we provide an overview of the the safety-critical specification for Java (SCJ) which is the current effort to produce a standardized subset of Java for safety-critical software development. In Section 2.5 we present a brief description of the Java Optimized Processor JOP, an implementation of the JVM in hardware, which is the embedded platform used for our safety-critical Java implementation.

Previous work relevant for this thesis is presented in sections 2.6 to 2.9. In those sections we describe previous work on implementing safety-critical Java profiles on embedded systems (2.6); detecting illegal reference assignments, which is required by the SCJ profile, (2.7); developing use patterns and libraries in the context of SCJ's memory model (2.8); and developing methodologies and benchmarks for real-time Java profiles (2.9). We finish this chapter in Section 2.10 with a summary.

2.1 Real-time and Safety-critical Systems

Real-time systems are those systems where a quantitative notion of time, measured using a physical clock, i.e., a *real clock*,¹ is needed to describe, either totally or partially, the system's behavior [80]. In addition to producing correct computation results, real-time systems need to deliver such computation results in a bounded amount of time, that is, before specific deadlines. Real-time systems need to be in synchronization with the environment with which they interact, that is, they need to have a notion of the timing characteristics of the environment they interact with. This notion of time is particularly important when outputs to the environment are computed as functions of inputs from the environment.

Real-time systems can be classified as *soft*, *firm*, or *hard*, depending on how useful computation results are in the vicinity of their deadlines. More specifically, real-time systems are said to be [42]:

- *Soft*: if the usefulness of computation results degrades after its deadline resulting in a system that operates with a degraded quality of service.
- *Firm*: if the usefulness of computation results is zero after its deadline resulting in a system that operates with a degraded quality of service. Only a small amount of deadline misses can be tolerated.
- *Hard*: if the usefulness of computation results is zero after its deadline resulting in unrecoverable system failure. Hard real-time systems are also called high-integrity systems and can be further divided into mission-critical and safety-critical systems, depending on the consequences associated with a system failure.

Mission-critical systems are those systems where the consequence of failures is the “loss of capability leading to possible reduction in mission effectiveness” [87]. An example of mission-critical systems can be e.g. the sector of low-latency electronic trading [54]. In low-latency trading, computers that execute complex algorithms are used to complete financial transactions before any other competitor. Missing a deadline to complete a transaction may result in high monetary loss but no human life will be in danger. In contrast, safety-critical systems are those systems whose failures can have severe or catastrophic consequences possibly endangering human life. These consequences can be e.g., the loss of life or irreversible damage to the environment. Typical examples of safety-critical

¹As opposed to a *logical clock*, where only qualitative information is provided in the form of event ordering relations [80].

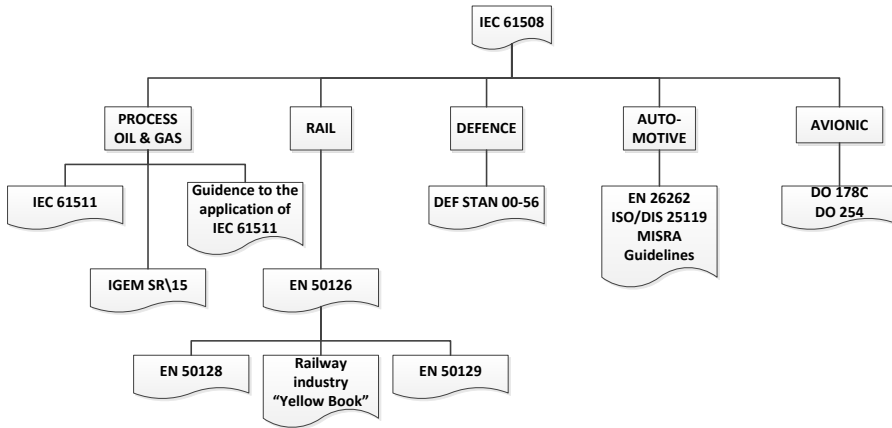


Figure 2.1: Relation of domain-specific second tier guidelines to IEC-61508 safety standard. Redrawn from [120]

systems can be found in the medical, transportation, defense, space, and energy sectors.

Due to the consequences of failures in safety-critical systems, they need to be licensed as fit for use in its intended environment before deployment. Licensing a safety-critical system is done through a careful and rigorous process of certification performed by a competent authority (e.g. the Federal Aviation Administration or the European Aviation Safety Agency for commercial aircrafts). The certification authority has to assess that there is enough convincing *evidence* to *argue* in favor of the system's safety *claims*, the so-called CAE system [109].

One way to conduct the certification assessment is by proving compliance to safety standards. Safety standards specify the assurance processes that should be used either in a prescriptive way by providing methods that must or are highly recommended to be followed, or based on objectives that specify *what* has to be achieved but not *how* [109]. Safety standards are domain-specific, i.e., dependent on the intended operating sector, and are in essence translations to different domain areas (e.g. railroad, aviation, etc) of the safety requirements of the IEC-61508 standard [120]. Domain-specific safety standards are referred to as *second tier guidelines*. For example, the second tier guidelines for the avionics industry is DO-178C, for the automotive industry is EN26262 and for the rail industry is EN50126, see Figure 2.1.

Another approach to certification is the development of *safety cases* [20], where

the system developers are required to explicitly state the system's safety claims, arguments, and evidence. In contrast to the standards-based approach, the safety case approach is more flexible as it allows the developer to customize and fine-tune the methods used to support the safety claims. It however lacks the large collective knowledge base and experience that supports safety standards.

Safety-critical systems are becoming more complex, interconnected, and embedded in many aspects of every day life. With these increased complexity many new challenges are emerging. Some of those challenges, outlined by Knight in [66], are:

- *Integration of multiple functions.* Requirements such as reduction of cost, weight and power dissipation are pushing for the integration of several interacting functions onto the same platform. Furthermore, studies show that the future of software systems is heading beyond systems of systems [79] into ultra-large-scale (ULS) systems [91]. ULS systems will push far beyond their number of, e.g., lines of code; interdependencies among software components; and hardware elements. In addition, it is expected that many of the components integrated into an ULS system will be of safety-critical nature [46].
- *Specification of systems requirements.* Developing correct, unambiguous and complete specifications is not an easy task even for non-safety critical systems. In safety-critical systems, requirements development is more complex because safety is now an essential attribute of the system. The development of system-level safety requirements need to be provided in the form of safety and assurance cases [109] produced by a risk analysis [78, 22]. The risk analysis can be done with, e.g. a fault tree or system hazard analysis.
- *Formal verification.* With the increased complexity of software for safety-critical systems, testing is becoming more difficult and current verification techniques based on testing (e.g. unit testing) are starting to raise doubts on their effectiveness [109, 85]. The use of formal methods as an alternative to testing has been included in DO-178C to “partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification” [108]. The challenges on the use of formal methods are: (1) requirements should be expressed in a formal language and (2) providing formal proofs for complex systems e.g., by model checking, often leads to state explosion.
- *Development time and effort* The most expensive and time-consuming activities carried out to develop and deploy safety-critical systems come

from the certification process. Most of the certification effort is in turn spent on verification activities while the actual process of software development (coding) requires only a small fraction. In some extreme cases, the effort and time spent on software assessment can be so high, that it may become an economic risk (e.g. a nuclear reactor protection system in Canada required 50 man-years of assessment [20]). Therefore, the use of new software technologies, languages, and certification approaches that help minimizing the cost and effort of verification activities are needed.

- *Security* As safety-critical embedded systems become more interconnected, security is becoming a concern. A device may be tampered and forced to work outside its safe operating conditions or critical data used by the system can be manipulated by unauthorized intrusions. The fundamental problem when including security into safety-critical systems is that architecture models used to build security-critical systems (e.g., the Multiple Independent Levels of Security and Safety (MILS) approach [12]) require that multiple software functions sharing common resources maintain a separation between trusted and non-trusted processes. This separation may lead to architectural differences between systems designed for safety and systems designed for security [32](e.g. sharing of information between dependent components may not be possible).

2.2 Programming Languages for Safety-critical Systems

During the period of late seventies and early eighties software for real-time systems was still developed in assembly language. As complexity in such systems increased, there was a need for the use of more expressive languages to simplify the software development task. A first approach was to develop programming languages specifically designed for real-time systems. Example of languages developed during that period are CORAL 66 and RTL/2 [50]. Those languages however were limited in concurrency and I/O features, relying largely on operating system support [29]. In the years to follow, (late 80's/early 90's), several other languages were proposed for real-time systems, but most of them were designed either for specific applications or implemented on a single computer model (e.g. Occam2 for transputer-based distributed systems) [50]. Therefore, they did not found widespread use and remained as research projects only. A notable exception of the languages developed during that period was Ada, which is still used for programming real-time systems. Instead of using programming languages specifically developed for real-time systems, the next step was to use well know and widely available high-level languages such as C.

For safety-critical systems however the criteria used to select a programming language was based on how the features of the selected programming language help on achieving specific requirements imposed by an applicable safety standard. Due to the large number of standards (see e.g. [120] pages 125–126), it is not possible to find a single language that can meet all possible requirements. However, research has been done to identify general requirements that programming languages should meet in order to be used in safety-critical systems. For example, in [36] requirements specific to UK's MoD² Def. Stan. 00–55 are analyzed and an assessment criteria consisting of eleven rules is provided. Those rules are then used to evaluate, amongst others, C and Ada for safety-critical systems. A similar analysis is done in [134] where the requirements of the avionics standard (DO-178B [107] at that time) and the IEC 61508 standard are also considered. Similarly, a wider spectrum of safety standards is considered in [51] resulting in a minimum set of general requirements where, amongst others, IEC, RTCA and MoD standards agree.

A set of guidelines, derived from standards and research literature, on the use of software languages in safety systems for nuclear power plants is presented in [56]. That study provides generic safe programming attributes, i.e., independent of the language used, and detailed recommendations, i.e., specific to commonly used languages such as C, C++ and Ada. The attributes are grouped into four top levels “which define a general quality of software related to safety” [56]. Each top level is further divided in intermediate and base attributes. The four top levels are: reliability, robustness, traceability, and maintainability. Intermediate attributes include e.g., predictability of memory utilization with a related base attribute that requires minimizing dynamic memory allocations.

Standardized guidelines for the use of programming languages in safety-critical systems are available for two languages: C and Ada. For C, the MISRA C standard [16] defines coding standards enforced through 16 directives and 143 rules. The rules³ are designed to facilitate static analysis and compliance with them can be checked solely by analyzing the source code. Compliance to a directive⁴ may require a revision of requirements documentation. For Ada, ISO/IEC DTR 15942 [63] provides standardized guidelines for its use in safety-critical systems. The document presents, amongst other guidelines, the impact that specific features of the Ada language have on verification techniques. The standard incorporates the Ravenscar profile [28] for the tasking model. The Ravenscar profile is a subset of the available tasking features of Ada that allow the timing behavior to be checked using e.g., rate monotonic analysis [63].

²Ministry of Defence

³E.g., “Rule 70: Functions shall not call themselves, either directly or indirectly.”

⁴E.g., “Directive 3.1: All code shall be traceable to documented requirements.”

The use of the Java language for safety-critical systems is assessed by Kwon in [68]. This assessment is based on a total of 23 requirements divided in two levels: Level 1, which are mandatory and Level 2, considered desirable or optional. The requirements are derived by analyzing a wide variety of safety standards from different domains e.g. UK's MoD Def. Stan. 00-55 and 00-56, and US' DoD⁵ "STEELMAN" [130]. The outcome of this study is the Ravenscar-Java profile which, as its counterpart for Ada, restricts some of the features of Java and enhances other areas (e.g. the scoped memory model) in order to increase predictability of Java programs.

From all those studies, one criterion on which all of them agree is that a programming language to be used for safety-critical systems has to be a subset of a more complex, well known language. This subset has to be augmented with real-time constructs to reflect the real-time programming paradigm. The use of a subset of a programming language on the one hand imposes the problem of how to enforce it and on the other hand provides the advantage that it can be formally defined. A formal definition of the full language can be very difficult.

Although some of the mentioned studies on identifying requirements for safety-critical programming languages are not recent, they are still applicable because the development of safety standards is a slow-paced changing field. For example, the current standard in the avionics industry, DO-178C, was published recently, in 2011, nearly 20 years after its predecessor, DO-178B, to address new technologies such as object oriented languages and the use of alternative options to testing (e.g., formal methods).

2.3 Java for Real-time Systems

Since its introduction in 1995 [48], the Java programming language has been widely used in many application domains. Part of its success can be attributed to being a strongly typed, architecture independent language with automatic memory management. As a strongly typed language, Java facilitates the implementation of extensive compile-time checks and avoids errors associated with pointer arithmetic; its architectural independence increases portability of applications and an automatic memory management helps to avoid e.g. memory leaks and dangling pointers.

However, many of Java's features are not adequate for (hard) real-time applications because they affect predictability or complicate different types of verification analysis and tests required as evidence to support compliance to a safety

⁵U.S. Department of Defense

standard. The features that make Java not suitable for real-time systems, and specially for high-integrity systems, can be classified in three categories: (1) object oriented programming features, (2) loosely defined aspects of the language, and (3) lack of services to implement a hard real-time programming model.

Object oriented programming features such as inheritance, polymorphism and dynamic dispatch can introduce different types of complications in software developed for safety-critical systems. For example, predictability in the control and data flow of a program is affected by dynamic dispatch because, e.g., which version of a method to call depends on the runtime type of the receiver object. Testing and different forms of analysis, e.g. structural coverage analysis (decision coverage and modified condition/decision coverage analysis), and source code to object code traceability become more complex.

Furthermore, DO-332, one of DO-178C supplements, requires *correct type substitutability*, that is, if inheritance is considered a specialization relationship, if a method call expects type C, then type S, which is a subtype of C, is also acceptable. The issue here is how to guarantee that the method defined in S is appropriate, that is, it does not strengthen preconditions and postconditions and invariants are not weakened [72].

Loosely defined aspects of the language help making Java code more portable across implementations. However, this freedom can lead to different behaviors. For example, thread scheduling requires the use of the underlying capabilities of the system and some implementations (e.g., IBM's WebSphere) ignore Java threads' priorities and assign system default priorities. Therefore, no guarantees can be made on fairness in the execution of threads or even if threads make progress at all [71]. In addition, when there is contention for processing resources, there is no guarantee that the highest priority thread will be executed in preference to lower priority threads [27]. Another example of undefined behavior is how the threads in the entry set (threads suspended waiting for a monitor) or the wait set (threads suspended by calling the `wait()` method) are managed, i.e., in which order are the threads added to and removed from the set.

Hard real-time programming constructs Originally, Java was not developed to be used in real-time systems. Instead, it was created to solve some of the problems that the use of the C language presented for programming networked embedded systems [48]. To be usable for real-time embedded systems, the Java virtual machine and its operating environment must support the following [27]:

- Strict thread priority control, that is, execution of threads has to be done according to the priorities specified in the Java threads.

- Priority based preemption, in order to implement the most common scheduling policies for hard-real time systems
- Priority inversion control, to avoid high priority threads being blocked by lower priority threads and potentially miss a deadline.
- Synchronous and asynchronous event processing
- Access to physical memory, to control I/O devices
- Ability to schedule hardware interrupts, that is, to execute interrupt service routines under control of the scheduler.

One of the first publications proposing the use of Java for real-time systems can be found in [88]. In that work, Nilsen argues that “by combining certain Java programming conventions with special implementation techniques, it is possible to support varying degrees of real-time reliability” [88]. The work is focused on highlighting important issues and to suggest general solutions (e.g. a portable model for real-time computation) with an emphasis on restricting implementation choices.

Nilsen’s proposal triggered a reaction on the National Institute of Standards and Technology (NIST) to produce a requirements document to extend the Java platform for real-time [90]. NIST published a document entitled *Requirements for real-time extensions to the Java platform* [31] from where two compliant specifications were latter produced: the Real-time Specification for Java (RTSJ) [26] and the Real-Time Core Extension Specification for the Java Platform (RT Core) [62]. Of these two specifications, the RTSJ received more attention both by academia and industry because RTSJ was initially supported by Sun Microsystems (now part of Oracle Inc.) and IBM. In fact, the development of RTSJ started the Java Community Process (JCP) with Java Specification Requests (JSR) 1. In contrast, RT Core did not enjoy good acceptance by the Java community because RT Core proposed modifications to the Java language [57].

The development of RTSJ had the goal of specifying how real-time behavior can be achieved with Java without changing the language at all [27]. There are seven areas that RTSJ enhances for the development of real-time applications [26]:

- Thread Scheduling and Dispatching: The minimum requirement is a fixed priority preemptive dispatcher. New thread classes are defined, the `Real-timeThread` and `NoHeapRealtimeThread`, both of them executed according to specific scheduling and release parameters. Instances of `RealtimeThread` can be affected by the GC while `NoHeapRealtimeThread` are not subject to GC interruptions.

- **Memory Management:** There is no particular GC algorithm required but if one is used, it should be precisely characterized in terms of its “effects on the execution time, preemption, and dispatching of real-time Java threads” [26]. In addition, there are regions of memory, i.e. `ScopedMemory` areas, where allocation and deallocation of objects is not affected by GC interference.
- **Synchronization and Resource Sharing:** The `synchronized` keyword should include, as part of its implementation, the appropriate means to control priority inversion. In addition, wait-free queues are provided for communication between threads affected by GC and threads not affected by GC.
- **Asynchronous Event Handling:** RTSJ provides both, mechanisms to represent and logic to execute external events. Occurrence of an external event triggers the execution of logic under the control of the system’s scheduler.
- **Asynchronous Transfer of Control:** A mechanism that allows the implementation of code that can be safely interrupted. Fine grade control is provided over which methods can be interrupted therefore, non-interruptible code can be used to avoid leaving objects in an inconsistent state. In addition, synchronized code (methods or blocks) is non-interruptible therefore the risk of leaving unreleased locks is avoided.
- **Asynchronous Thread Termination:** By extending the functionality of `Thread.interrupt()`, safe termination of threads can be implemented.
- **Physical Memory Access:** Allows the construction of objects in physical memory and byte-level access to physical memory.

2.3.1 Java for High-integrity Real-time Systems

Although RTSJ’s enhancements provide real-time functionality and increased predictability of Java applications, it is still too complex to be used on software components for safety-critical systems. A number of studies have proposed additional profiles intended for the use of Java on safety-critical systems. For example, the work done by Puschner and Wellings in [98] restricts many of the features of the RTSJ and proposes a profile for high-integrity real-time Java programs based on Ravenscar-Ada’s [63] concurrency model. The profile has a fixed number of threads that can be either time-triggered (periodic) or event-triggered (sporadic) and interact only through shared data. The profile divides the execution of applications in two phases, an initialization phase and a mission phase. Non time-critical activities (including the creation of all threads that the application will use) take place in the initialization phase while time-critical

activities (i.e. the actual execution of threads) are executed in the mission phase. There are also additional restrictions in memory management (e.g. prohibits the use of a GC), use of shared resources (e.g. requires the use of priority ceiling emulation to avoid deadlocks), and time and clocks (e.g. RTSJ's `Timer` is not allowed). Details can be found in [98].

The work of Puschner and Wellings served as a starting point to define the Ravenscar-Java (RJ) profile which inherits most of the features of its predecessor profile (e.g., two execution phases, only periodic and sporadic threads) and adds features such as annotations for temporal and memory usage analysis, and a restricted form of memory used for dynamic memory allocation in the form of nested scopes.

The work done by the HIJA project also produced a profile for high-integrity Java [5] that adds amongst others, the following limitations:

- Only bound asynchronous event handlers, either periodic or sporadic, are allowed.
- Only uses the default RTSJ preemptive priority-based scheduler with no dynamic priorities.
- Scheduling within priority is FIFO.
- Deadline miss detection but no CPU-time monitoring.
- No use of the `synchronized` statement is allowed.
- No suspension for any reason is allowed within a synchronized method.
- Priority inversion is controlled by use of the RTSJ's priority ceiling emulation protocol.
- Application reinitialization or mode changes are supported.

A different approach is taken by Schoeberl et al. and in [118] they propose a profile for safety-critical Java where, instead of providing a subset of the RTSJ, the authors start from scratch to build a reduced specification for hard real-time Java. The profile also keeps a fixed number of periodic and sporadic threads but with, amongst others, the following changes:

- Considers the `RealTimeThread` as a fundamental concept whose constructor needs only application specific parameters.

- Derived thread types such as `NoHeapRealTimeThread` are eliminated.
- Schedulable objects are released according to time and not priorities. If needed, priorities can be computed by mapping of timing requirements to priorities by e.g. deadline-monotonic assignment.
- Only relative time is conserved, absolute time is not part of the profile.
- Maintains the initialization and mission phase as previous profiles but adds an additional phase, a stop phase, used for threads to perform optional cleanup procedures.
- Does not support mode changes.

The safety-critical Java (SCJ) profile is being developed under the Java specification request 302 (JSR-302) and is the current effort to standardize a Java profile for safety-critical systems. Its main characteristics, namely the concurrency, execution, and memory model, are described in the next section.

2.4 Safety-critical Java Specification (JSR-302)

Similar to the approach followed by the MISRA C standard and the Ravenscar-Ada profile, SCJ provides a smaller, tightly defined subset of the Java programming language. It is intended to be restricted enough to reduce testing and verification complexity. SCJ is based on the RTSJ and its programming model is more restricted in terms of concurrency, synchronization, memory, input and output, clocks and timers, and exception processing [77]. Three levels of compliance with different degrees of complexity are defined, namely Level 0 (L0), Level 1 (L1), and Level 2 (L2) where L2 is the more complex.

In this section we describe the three main characteristics of SCJ namely its execution, concurrency, and memory models.

2.4.1 Execution Model

SCJ's programming model is based on the execution of missions. Missions encapsulate modes of operation and separate the execution of time-critical from non time-critical operations. They consist of a bounded set of registered managed schedulable objects (MSO) executed either as a cyclic executive (L0) or under the control of a fixed-priority preemptive scheduler (L1, L2). MSOs are

created with parameters that cannot be changed after they start executing. Such parameters define a real-time priority, specify the nature of the MSO's execution (periodic or aperiodic), and binds the MSO to a memory area.

An SCJ application can have one or more missions executed in a predefined sequence. The order in which the next mission to execute is selected is under the control of a *mission sequencer*. A mission has three phases: initialization, execution, and cleanup. During initialization, all non time-critical operations are executed (e.g., setting up data structures, initializing peripheral devices) and all MSOs needed by the mission are created and registered. In the execution phase, all time-critical operations are executed by the MSOs registered to the mission. Any MSO can trigger mission termination at any time. Once a mission termination request has been made, all currently executing MSOs will finish their last release and no further scheduling of MSOs releases is allowed. The cleanup phase starts once all MSOs have finished their last release. This phase is used to remove any data, structures, or configurations used by a particular mission and that are not needed by the next mission.

L0 and L1 applications can have only one mission executing at all times. Parallel missions are only allowed at L2. Parallel missions are started by nested sequencers registered as MSOs for another mission.

2.4.2 Concurrency

Depending on the required type of real-time activity, an MSO can be a *periodic event handler* (PEH) or an *aperiodic event handler* (AEH). The use of event handlers is preferred to the use of threads because all the activities for an individual release are encapsulated in a single method [132], namely the handler's `handleAsyncEvent()` method. Threads are only available at L2 in the form of *managed threads* that are a restricted version of RTSJ's `NoHeapRealTimeThread`. Managed threads are useful to implement activities with an execution pattern different from that of an aperiodic or periodic task, e.g. background activities that run at all times.

2.4.3 Memory Model

In SCJ, the use of the heap is not allowed. Instead, memory management is based on memory areas free of garbage collector interactions called *scoped memories*. Scoped memories are introduced in SCJ in order to make object allocation and deallocation time and space predictable. The concept of scoped

memories is inherited from the RTSJ and depending on the intended life of the objects allocated in a scoped memory, there can be *immortal*, *mission*, or *private* memories. Immortal memory is used to store objects that will live for the whole execution of the application, mission memory holds objects that belong to a specific mission and private memory holds objects used only by a single MSO. Objects allocated in immortal memory can be accessed by any MSO and objects in mission memory are accessible only to the MSOs belonging to the mission. As in RTSJ, static initializers run in immortal memory and Class objects and static fields reside in immortal memory.

The different scoped memories are represented by instances of the `ImmortalMemory`, `MissionMemory` and `PrivateMemory` classes. The latter two extend a common SCJ class, namely `ManagedMemory`. Unlike the RTSJ, it is not possible to explicitly create instances of memory objects, instead, SCJ's infrastructure creates such objects in response to certain user requests or according to the application execution phase. The SCJ infrastructure creates and enters `MissionMemory` objects during the mission initialization phase while `PrivateMemory` objects are also created at mission initialization but entered during the mission execution phase. Memory objects represent an allocation context but the memory objects themselves are created outside the allocation context they represent e.g., the object representing the initial mission memory resides in immortal memory.

Each MSO is created with an initial private scope that is entered by the infrastructure on each release and is left at the end of the release. On scope exit, all the objects allocated in that scoped memory are deallocated. An MSO can also enter nested private scopes which are used to recycle memory space on every MSO release. The order in which MSOs have entered scoped memories form a stack structure that can only grow linearly.

Without a garbage collector, memory management becomes a task of the application developer. The application developer therefore needs to be aware of where objects are allocated in order to avoid creating illegal reference assignments. An illegal reference assignment is created when an object stores a reference to another object located in a shorter-lived scoped memory. To avoid creating such references, a set of rules have to be enforced by the JVM on every reference assignment operation (Sections 2.7 and 3.5 provide more details about the reference assignment rules and methods to enforce them).

2.4.4 An SCJ HelloWorld Example

Figure 2.2 shows how a minimal, but complete, L1 SCJ *HelloWorld* example looks like. An SCJ application needs four elements: (1) a *Safelet*, to bind the application to the runtime environment, (2) a mission sequencer, to provide the order in which missions are executed, (3) one or more missions, to encapsulate modes of operation and provide a separation between time-critical and non time-critical operations, and (4) one or more managed schedulable objects, to provide real-time activities with different execution patterns.

The *Safelet* class contains three methods that should be implemented: *immortalMemorySize()*, *initializeApplication()*, and *getSequencer()*. The *immortalMemorySize()* method (line 6) returns the amount of immortal memory that this SCJ application requires. The *initializeApplication()* method (line 7) provides an entry point to allocate *immortal* objects. Immortal objects can also be allocated with static class initializers. The *getSequencer()* method (line 10) returns the sequencer object that controls the order in which missions belonging to this application will be executed.

The mission sequencer needs to implement the *getNextMission()* method to get a reference to the next mission to execute. In this example, our SCJ application extends the *Mission* class and hence the implementation of the *getNextMission()* method (line 17) returns a reference to the *app* object which was allocated in immortal memory (line 2). For this example, the mission sequencer returns a non null mission only the first time its *getNextMission()* method is called. Successive invocations (after mission cleanup) will return a null mission resulting in the termination of the application.

As with the immortal memory requirements, every mission needs to define the maximum amount of *mission memory* that it requires. The *missionMemorySize()* method (line 25) provides this value. The *initialize()* method (line 28) is used to create mission-specific data structures and to create and register all MSOs that belong to the mission. In our example, there is only one PEH instantiated as an anonymous class (line 33).

The active logic of our example is implemented by overriding the *handleAsyncEvent()* method (line 42). This minimal example will periodically print a “Hello World” message five times before issuing a mission termination request (line 46). In SCJ, the use of *System.out* is not allowed in order to “reduce the size and complexity of the System class” [77]. There is however support for a reduced set of the *java.io* package that can be used to implement a class that prints messages to a console terminal. The period of this PEH corresponds to 150 ms and is specified in the *PeriodicParameters* parameter of its constructor.

```

1  public class HelloSCJ extends Mission implements Safelet {
2      static HelloSCJ app = new HelloSCJ();
3      static Terminal term = Terminal.getTerminal();
4
5      /* Safelet methods */
6      public long immortalMemorySize() {return 16384; }
7      public void initializeApplication () {/* Set immortal data */}
8
9      @Override
10     public MissionSequencer getSequencer() {
11         return new MissionSequencer<Mission>(new PriorityParameters(11),
12             new StorageParameters(1024, null, 1024, 128, 128)) {
13
14             boolean served = false;
15
16             @Override
17             protected Mission getNextMission() {
18                 if (!served) {served = true; return app;}
19                 else { return null; }
20             }
21         };
22     }
23     /* Mission methods */
24     @Override
25     public long missionMemorySize() {return 4096;}
26
27     @Override
28     protected void initialize () {
29         long totalBackingStore = 1024, maxMemoryArea = 512;
30         long maxImmortal = 128, maxMissionMemory = 128;
31         long[] sizes = null;
32
33         new PeriodicEventHandler(new PriorityParameters(12),
34             new PeriodicParameters(new RelativeTime(),
35                 new RelativeTime(150, 0)),
36             new StorageParameters(totalBackingStore, sizes ,
37                 maxMemoryArea, maxImmortal, maxMissionMemory)) {
38
39             int cnt = 0;
40             @Override
41             public void handleAsyncEvent() {
42                 term.writeln ("Hello World!");
43                 cnt++;
44                 if (cnt > 4) {
45                     Mission.getCurrentMission().requestTermination ();
46                 }
47             }
48         }.register ();
49     }
50     /* Start the application from a main method */
51     public static void main(String[] args) { ScjLauncher.Main(app); }
52 }

```

Figure 2.2: Example of an SCJ HelloWorld program

Note that, as with immortal and mission memory, the memory requirements for the PEH's private memory need to be specified. This is done with the storage parameters argument (line 36) which specifies the size of the backing store reservation (`totalBackingStore`), the per-release maximum memory area (`maxMemoryArea`), and the maximum immortal and mission memory the PEH is allowed to use (`maxImmortal` and `maxMissionMemory`).

How an SCJ application is started is implementation dependent and in the last part of this example (line 51) we show one possible way to start the application. We define a public static `Main()` method in a class located in the SCJ package, `ScjLauncher` in this example, and call that method from the main method of the application. For an RTSJ-based implementation, the initial main thread will run by default in the heap and the SCJ profile does not allow the use of the heap memory. Therefore, the `ScjLauncher` class will create a `NoHeapRealtimeThread` in immortal memory and after that this new `NoHeapRealtimeThread` will execute the methods of the safelet object passed as argument to the `ScjLauncher.Main` method.

2.5 Java Optimized Processor JOP

JOP is an implementation of the JVM in hardware [113]. JOP uses microcode as its native instruction set. Most of the bycodes defined by the JVM are implemented as a sequence of microcode instructions, only the more complex bycodes, e.g., the `new` and `newarray` bycodes, are implemented in Java. The mapping of bycodes to microcode instructions is done using a translation stage which converts a bytecode instruction into the start address of a table containing the microcode sequence that implements that bytecode. This translation stage takes exactly one clock cycle and thus it can be pipelined [113, 112].

JOP is a RISC embedded processor, implemented using a stack architecture, designed for (hard) real-time systems. A number of architectural features help achieve predictability in the execution of Java programs. The most important features are listed below:

- Data dependencies are eliminated from the stream of bytecode instructions.
- It has a simple 4-stage pipeline that avoids the use of branch prediction logic, prefetch units, forwarding logic (no dependencies between bytecodes), and pipeline stalls.

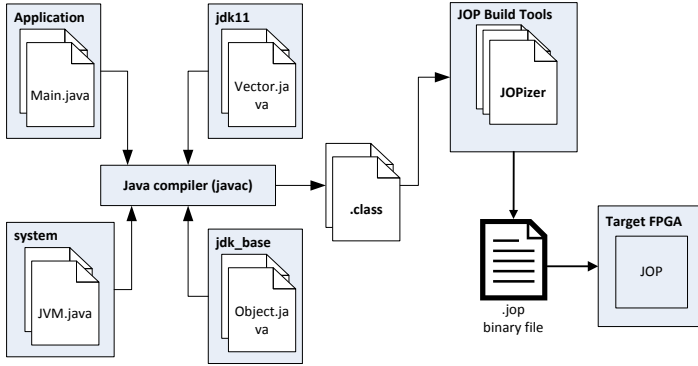


Figure 2.3: Application build tool-chain of JOP

- It uses time predictable caches: a stack cache, that substitutes the data cache and caches local variables and the operand stack; and a method cache, that substitutes the instruction cache and caches entire methods and therefore restricts cache misses to `invoke` and `return` bytecodes

These architectural features result in the possibility to predict the execution time of every bytecode. Moreover, as a pure Java system, there is no underlying OS layer and therefore there is no dependency on OS features that may complicate the execution time analysis.

Translating Java applications into files that can be executed on JOP follows the process illustrated in Figure 2.3. Java class files are created from the application source and the different libraries using a standard Java compiler. The class files are processed and linked into a single file that can be downloaded into the JOP processor for its execution. The resulting output file contains the following information:

- References to different boot and initialization structures
- An area where static fields are stored
- A method table, which contains the bytecodes of every method
- A section with pointers to the class initialization methods
- A table holding all constant strings
- An area with the class information table
- An area with the class `Class` objects, generated at application build time

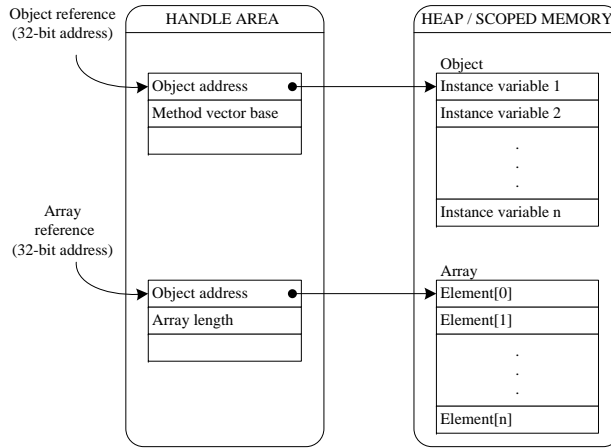


Figure 2.4: Object and array layout in JOP

Only the classes used in the application will be processed and linked into the output file. Furthermore, a correct class initialization order will be determined in order to avoid circular dependencies between class initializers.

Access to memory and internal structures is needed to implement certain features of the VM, e.g., the creation of objects. However, there are no bytecodes available that allow such a low level access to memory. In JOP, this access is obtained through *native* methods. Native methods are available as part of the API accessible to the programmer. As the native language of JOP is the microcode, JOP's native methods are translated during application build into special bytecodes that correspond to those undefined bytecodes in the JVM specification.

Objects and arrays in JOP are represented by a structure composed of the two parts described below and illustrated in figure 2.4:

- A header, a structure that contains meta-data information, called the object's handle. It contains amongst others, pointers to the object itself, a pointer to the method table or array length, and information for the garbage collector.
- The object itself, a structure which contains instance variables or, for array objects, the elements of an array.

The first value in the object handle, at offset 0, points to the object instance. The value at offset 1 stores a pointer to the runtime class structure that represents the type of the object. The pointer to the class structure points to the first element of a method table. The use of separate areas for a handle and object instance is beneficial for the implementation of a garbage collector. Only the reference contained in the object handle needs to be updated. The drawback is the double pointer dereferencing to access an object's instance [112, 131].

JOP's API provides a basic class, called `RtThreadImpl`, to implement real-time activities [110]. Periodic or aperiodic activities are supported through two additional classes, namely `RtThread` and `SwEvent`, that delegate functionality to the `RtThreadImpl` class. JOP's `RtThread` class allows the implementation of periodic activities that are scheduled for execution according to their priority. The `SwEvent` class is used for aperiodic activities, also scheduled for execution according to their priority but with the difference that they need to be explicitly released by a call to its `fire()` method. It is necessary to specify timing parameters on any of these classes that extend `RtThreadImpl` otherwise they are considered as non real-time threads and will be executed at normal Java thread priorities.

JOP's real-time threads are executed under the control of a fixed priority pre-emptive scheduler. Every thread is assigned a unique priority in order to avoid FIFO queues within priorities. Executing a synchronized method or statement disables all interrupts, including the timer interrupt that triggers the scheduler. In this way, critical sections are executed at the highest possible priority of all threads, thus effectively implementing a priority ceiling protocol where the ceiling is set to the maximum priority that any thread can possibly have.

2.6 High Integrity Java Profiles on Embedded Systems

Safety-critical Java is relatively new, and there are only a few implementations available that specifically target embedded systems. We describe in the following sections previous work that has been done to implement different high integrity Java profiles.

2.6.1 Ravenscar-Java in the aJ-100 Processor

Søndergaard et al. in [123] provide an implementation of the Ravenscar-Java (RJ) profile, first proposed by Puschner and Wellings in [98] and further elaborated by Kwon and Wellings in [70] and [68]. The implementation targets industrial applications and uses an aJ-100 processor developed by aJile Systems [7]. The aJ-100 is a 32-bit microprocessor that directly executes JVM bytecodes (implemented in microcode) as its native instruction set. In addition, it provides a microcode programmed real-time kernel that provides, among others, support for scheduling, context switching and object synchronization.

RJ's periodic threads are implemented by delegation to aJile's `PeriodicThread` class provided by the aJ-100 API. Much of the complexity of using aJile's API for setting periodic activities (e.g. configuring the piano roll structure) is hidden through the use of auxiliary classes. Priorities for the `PeriodicThreads` are set according to a rate monotonic assignment that is done in the initialization phase.

In RJ, only sporadic events are defined. They can be either of type `SporadicEvent` (software generated) or `SporadicInterrupt` (hardware generated). Both types of events are executed in Søndergaard's et al. implementation by a `SporadicEventHandler` (SEH) with a deadline equal to its minimum inter arrival time. An SEH extends RTSJ's `BoundAsyncEventHandler` which implements its permanently bounded event handler as a private inner `NoHeapRealTimeThread`. This thread is created and started as part of the `BoundAsyncEventHandler` constructor. Its `run()` method executes the logic associated with the occurrence of an event. The bounded thread is blocked right after it is started by a call to the `wait()` method with a lock on itself.

The occurrence of an event triggers the following series of actions: (1) SEH's `handleAsyncEvent()` method is executed which checks if the minimum interarrival time has elapsed, if not, the event is ignored, (2) if the minimum interarrival time has elapsed, the SEH's bounded thread `handleAsyncEvent()` method is called which in turn makes (3) a call to the `notify()` method with a lock on the bounded thread object, (4) the bounded thread awakes thus executing the logic associated with the event. Because the bounded thread is defined as a private inner class in the `BoundAsyncEventHandler` class, it can be locked by only one SEH.

From the memory classes, only `ImmortalMemory` and `RawMemory` are implemented, there is no support for scoped memory. `ImmortalMemory` is implemented by turning off the garbage collector of the aJ-100 processor thus making the heap an implicit immortal memory. `RawMemory` uses aJile's low-level memory access API.

2.6.2 oSCJ

One of the first implementations of safety-critical Java on an embedded platform is presented by Plsek et al. in [96] as part of the open safety-critical Java (oSCJ) project [96]. Plsek et al. provide an implementation of SCJ's Level 0 running on the OVM virtual machine [15]. The OVM is a framework that allows alternate implementations of core VM functionality (e.g. different versions of priority inheritance monitors) in order to build and test VMs with different features. OVM uses an ahead-of-time compiler to translate Java code to C++ and then it uses the GCC compiler to obtain machine code. SCJ's implementation on OVM runs on an FPGA board executing the RTEMS real-time operating system on a LEON3 processor.

For a Level 0 implementation of SCJ only a single thread is sufficient. For the oSCJ project a single thread performs all tasks, including VM boot and startup procedures, and the execution of the SCJ application. Having a single thread in the system makes synchronization not needed. The SCJ library was designed to be independent of the underlying VM where it is executed. This independence is achieved by defining an interface through which the SCJ library interacts with the VM services. Through this interface memory-related services (e.g. creation, deletion, and switching of memory areas) and time-related services (e.g. get the current time) are delegated to the VM. Thread management is delegated to the operating system (a POSIX environment) through an abstraction layer.

Memory is divided in three levels: the top level, the backing store level, and the scope level. The top level memory holds the `ImmortalMemory` region and the currently active missions object (for a Level 0 application there is only one mission active at all times). The backing store level is used to store the schedulable objects' private memories. The scope level stores the objects created during the execution of a schedulable object. It is unclear from the description in [96] if the space required by nested private memories is taken from the scope level or the backing store level, however, from the explanation of the scope check assignments, it is most likely taken from the scope level.

As the VM is delegated memory-related functions, it also implements the scope checks. The check works by locating and comparing the memory areas where objects are stored. The memory is organized in a way that lower addresses correspond to longer lived memory regions. Therefore, a check is deemed acceptable if the object to be assigned to a reference field is located in a lower or equal memory address range as the object where the field to be assigned is defined. If that condition does not hold, then a second step might be needed to recover the scope nesting relationship of the memory areas.

2.6.3 SCJ on HVM

In [121], Søndergaard and Ravn provide an implementation of SCJ's Level 0 and Level 1 on top of the Hardware near Virtual Machine (HVM) [67] that targets low-end embedded platforms. It is reported that their implementation can fit embedded systems with as low as 16 kB of RAM and 256 kB of flash memory. HVM is a small footprint JVM for embedded devices that compiles Java to C. HVM translates a single application written in Java into self contained standard C code that can be integrated into an existing C-based execution environment without any additional dependencies [67].

Similar to oSCj, HVM's SCJ profile interacts with the underlying VM services through an interface that provides memory management, scheduling, and real-time clock related services. Those services are made available through five inner classes, on top of which the SCJ framework is constructed.

The approach taken by Søndergaard and Ravn differs from oSCJ in that their implementation does not depend on an existing RTSJ implementation. SCJ classes defined in JSR-302 as part of the `javax.realtime` package are declared in the `javax.safetycritical` package. For example, in SCJ, the scoped memory model uses RTSJ's `MemoryArea` and `AllocationContext` as base classes. In contrast, in HVM both classes are located in the `javax.safetycritical` package.

Scoped memory is implemented by the `MemoryArea` class which extends the `VMInterface.AllocationArea` inner class. A stack structure is used to keep track of the active memory areas.

The execution model uses what is called a *primordial mission*, a top-level mission where the mission sequencer is considered as the only handler to execute (recall that both L0 and L1 applications have no nested sequencers). This *primordial mission* runs in `ImmortalMemory` as its allocation context. The mission sequencer is a handler and as such, a private memory is allocated for its execution. The logic of the sequencer is implemented in its `handleAsyncEvent` method where the following activities execute sequentially: (1) set-up of mission memory, (2) selection of next mission, (3) initialization of mission (in `MissionMemory`), (4) start of mission's handlers, (5) wait for mission termination, and 6) mission cleanup.

For a Level 0 application, all handlers are executed in the order they are returned by the `getSchedule` method of the mission. Mission termination is done at the beginning of a major cycle. For a Level 1 application, each handler is assigned a HVM process and all HVM processes are started once the mission becomes

active. A process scheduler selects from a priority queue the next process to release. Selection of next process is started via an interrupt generated by a hardware clock. The priority scheduler detects mission termination if there are no more processes to release.

Other features such as long events and long event handlers, happenings, and the RTSJ interface to hardware (i.e. `RawMemory` related classes) are not implemented. Furthermore, priority ceiling emulation and multi-core features are not supported.

2.6.4 Predictable Java

In the work presented by Bøgholm et al. in [23] a different approach is taken. Instead of providing an implementation of SCJ, their implementation is based on a profile called Predictable Java (PJ). PJ is a Java profile suitable for the development of high-integrity real-time embedded systems that builds on the ideas of [98, 70, 112] and [123]. The profile is based on the execution of event handlers grouped in missions which in turn are also considered event handlers. Furthermore, the profile considers that, as PJ classes are more restricted in functionality than RTSJ classes, PJ should be a generalization and not a specialization of RTSJ. In contrast, SCJ classes as defined by JSR-302 are a specialization of RTSJ classes.

Two implementations are provided, with the restriction of not having mission termination support (see below). The first implementation runs on an ARM-based embedded controller while the second uses Tymesys 1.0.2 RTSJ reference implementation running on an x86 Linux environment. The ARM-based implementation uses a modified JamVM [8] running on the Xenomai [10] real-time extension to the Linux OS. Operating system support is used to implement real-time tasks, scheduling, and synchronization. The RTSJ based implementation delegates functionality to the RTSJ classes using an adaptation layer.

One of the most important points of Bøgholm et al. is that of considering missions as event handlers, in contrast to being only handler containers. They argue that with that change, initialization and termination can be scheduled events. Furthermore, this avoids introducing new class hierarchies (e.g. the role of SCJ's `MissionSequencer` and `Mission` can be performed by the same class) and special mission memories. Individual handlers are added to its corresponding mission only when the mission is initialized, just as in SCJ, but for their execution, complete missions are added to the system's scheduler. Likewise, complete missions are removed from the scheduler at mission termination. Nested missions (i.e. missions started by another mission) occur naturally as a

mission can have another mission as part of its event handlers.

The memory model conserves the same ideas as in SCJ of having handler private memories but the private memory of the outer most mission becomes an implicit `ImmutableMemory`. SCJ style `MissionMemory` is therefore the private memory of each inner mission. In this way, there is no need to define different classes for immortal, mission, and private memories, and just one kind of scoped memory is necessary. That is, the concept is needed but the classes are not [114].

2.6.5 Cyclic Executive for SCJ on Chip-multiprocessors

In [99], Ravn and Schoeberl provide an implementation of SCJ L0 on a chip-multiprocessor version of JOP. They address the problem of generating a valid static schedule that can be used to implement a table driven multiprocessor scheduler. They use the model checking tool UppAal to find a valid schedule.

The authors then proceed to model tasks as timed automatas with and without shared resources. As SCJ's L0 is a cyclic executive, only one thread is needed thus avoiding the need of synchronization. However, for a CMP configuration, synchronization has to be considered for objects shared between cores. For this L0 application, there will be a single thread per core executing part of the table driven schedule and immortal and mission memory are the two memory areas shared between cores.

In their implementation, Ravn and Schoeberl depart from what SCJ allows in a L0 by allowing more than one processor, use plain Java runnables instead of PEHs for the cyclic frames in order to avoid introducing dead code (e.g., `PriorityParameters` are not used in L0), allow task migration as it gives more freedom in the generation of the schedule, and provide detection of frame over-runs that can be queried from the application.

They show that indeed a CMP setting of L0 is feasible with the use of offline tools to generate a schedule and with the additional simplifications that SCJ L0 offer on synchronized access of shared resources.

2.7 Reference Assignment Checks

As part of the contributions of this thesis, we have implemented the reference assignment checks in hardware. The method we used is based on the execution

of write barriers similarly as in [58] and [59] for the RTSJ.

In [58], the scope stack of a thread is scanned to check that the memory area of the destination of a reference assignment is deeper nested than that of the source of the reference. In that implementation, the time to scan the stack is proportional to the number of nested scoped levels. To bound the execution time of the check, the authors limit the levels of nested scopes. In a latter work [59], the performance is improved with the aid of the write barrier support provided by the picoJava-II microprocessor [9] and specialized hardware. The scope stack is stored in an associative memory which allows a faster scanning of the hierarchy of nested scopes. Given the simplifications that the SCJ memory model provides, we do not need complex hardware to store or scan the scope stack to compare nesting levels between the objects involved in reference assignments. Our solution is simpler and requires only that the scope levels be compared.

In the SCJ implementation of Plsek et al. [96] the write barrier works by locating and comparing the memory areas where objects are stored. The memory is organized as a continuous region that starts with the immortal memory then continues with mission memory followed by any private memory or stack of private memories. Lower memory addresses correspond to longer lived memory regions. The check is performed by determining the block of memory where both objects involved in a reference assignment are allocated. A second step might be needed to recover scope information of each object as both objects may reside in the same scope. We do not use the address value of each object itself because that would involve checking that the objects are allocated within a certain range, and that would be more time consuming as the boundaries of the memory region where the objects are allocated need to be known.

In [137], the authors give an analysis algorithm that statically guarantee that no illegal reference assignments can happen. The authors introduce the concept of *scoped types* as a way to encapsulate scoped objects. **Scoped** and **Portal** classes are defined and associated to their defining packages. Nested scopes are in turn associated with nested packages. Accessibility of a scoped class is restricted to instances of classes allocated in the same or nested scopes.

Verifying SCJ memory safety can be done statically by correctly using the annotation model defined in the appendix of the SCJ specification [77]. For example, in [125], annotations are used as part of a two step verification process. In the first step, the scope tree is constructed and checked for errors and in the second step, the tree constructed in the previous step is used to check a set of rules for annotated and unannotated classes. The use of this annotation model is however cumbersome and recent approaches to statically detect illegal reference assignments avoid their use. Some examples are the work of Dalsgaard et al.

in [37] that uses a pointer and escape analysis and in a more recent, yet to be published, work done by Chris Marriott from the University of York. His work uses a formalization of the SCJ memory model using the Circus formal language [93]

2.8 Scoped Memory Use

In this section we describe what has previously been done regarding scoped memory usage. As mentioned before, SCJ is relatively new and therefore the previous work is related to RTSJ's scoped memory. However, previous work on the RTSJ can be related to SCJ as these studies are aimed at eliminating scoped-memory related errors.

2.8.1 Patterns

Benowitz and Niessner introduced patterns used for periodic activities as well as scope aware factories in [19]. In their work, returning objects are allocated in immortal memory using memory pools (objects of fixed size) and memory blocks (byte arrays to allocate varying size objects). The use of memory pools in regions other than immortal memory are explored in [25] where the communication between several components participating in a real-time control loop is described. Each component is defined in a scoped memory and has its own pool of objects in a scoped memory other than immortal. Communication between components is performed by copying values. Recycling objects is an appealing solution to avoid running out of immortal memory. Nevertheless, in this work we explore other possibilities that include the use of SCJ's mission and private memories.

Pizlo et al. investigated design patterns for the RTSJ in [94]. The authors document several design patterns for the effective use of scoped memory regions. From their work only the *scoped run loop* pattern has a direct equivalent in SCJ. The rest are either RTSJ-specific (e.g., the *wedge pattern*), use features not available in SCJ (e.g., portals), or introduce violations to the reference assignment rules (e.g., the *handoff pattern*).

The patterns catalog of Benowitz and Niessner [19] is extended with another collection of patterns in [34]. For this collection, the Memory Tunnel pattern is particularly interesting. It is used to move data between threads executing in different memory areas. It has however, the problem that forces a "safe"

violation of the reference assignment rules. It relies on a memory tunnel structure which is constructed using native methods to bypass the scope constraints regarding reference assignment checks. As SCJ is intended for certification under standards such as DO-178C, this behavior is likely to reduce the chances of passing any certification. Thus, we use a different approach to move data between scopes that goes through mission memory.

Kwon and Wellings in [69] proposed to map memory areas to Java methods in a user-controlled fashion. Motivated by the overhead incurred when enforcing the single parent rule and reference assignments [24], they propose a model that reduces the need for such checks. This is achieved by associating a memory region with methods that have been annotated. Memory areas are entered when the method is invoked, effectively changing the allocation context. Objects created within this context are collected when the method returns. References to objects created outside the current method can be passed as parameters or as local objects in the enclosing object that the method belongs to. References to objects in a callee method are not allowed. If there is the need to return objects after a method is executed, it is done by specifying an additional parameter that can define where to store a returned object. The approach uses RTSJ scoped memories in a restrictive style that is very similar to SCJ private memories. The annotation facility hides the complexities of using the memory API to encapsulate methods and to return objects.

The Lifecycle Memory Managed Periodic Worker Threads pattern (LMMPWT) described in [40] presents an RTSJ framework where a group of no-heap periodic threads cooperate together to complete a task. This pattern focuses on object lifetime management and does not require an explicit understanding of scopes. Lifetimes assigned to objects are divided in four categories that can be directly related to the lifetime of immortal, mission, private and nested memory allocated objects in SCJ. Movement of data between scopes and creation of objects in arbitrary memory areas is done with an encapsulated use of the memory API. The drawback of this implementation is that it relies heavily on reflection for manipulation of objects and the `java.lang.reflect` package is not part of the SCJ specification.

In [137], the authors introduce the concept of scoped types as a way to encapsulate scoped objects. Scope and portal classes are defined and associated with their defining packages. Nested scopes are in turn associated with nested packages. Accessibility of a scoped class is restricted to instances of classes allocated in the same or nested scopes. This work is later extended in [97] where the authors document a number of programming idioms to manipulate scopes.

2.8.2 Libraries

In [39], the authors describe some of the challenges faced while implementing IBM's WebSphere, a RTSJ-compliant commercial Java virtual machine. One of the challenges here is integration with the existing JCL. Integration is challenging because objects are allocated in the memory area where the current thread runs, thus making all classes in the JCL potentially unsafe for shared use between different types of threads.⁶ WebSphere provides a small subset of classes that are safe for shared use between RTSJ threads. In the context of the WebSphere JVM, "safe" means free of throwing `MemoryAccessError` exceptions, i.e., errors that are thrown when attempting to refer to an object in an inaccessible memory area.

Automatic identification of no-heap safe classes is the topic of [41]. Dibble presents a taxonomy to classify existing classes according to the degree with which they can be shared by all types of RTSJ threads. This taxonomy is based on the existence of static or instance variables that can store references to objects in heap and no-heap memory areas. A list of potentially unsafe classes, obtained through static analysis, is also presented. Dibble's analysis identifies all classes that have non-final static reference fields as unsafe.

The Javolution project [38] is one of the first attempts to produce an extended library of reusable, time deterministic, no-heap safe classes. In that project, issues such as sharing objects between scopes are eliminated. In addition, extra steps, e.g., explicitly switching between memory areas, are handled automatically. Javolution allows dynamic resizing of collections and allocates the required extra storage from immortal memory. Allocations in immortal memory can become a memory leak when elements are removed from the collection. Javolution's dynamic memory allocation and reliance on exception handling makes static program and worst-case execution time analysis difficult.

In [53] the development of reusable libraries targeted for real-time Java is presented. The authors present classes that can be used as drop-in replacements for three types of collection classes: `List`, `Set`, and `Map`. The authors' approach is based on recycling objects from a fixed-size pool of objects. As a consequence, operations such as insertion or deletion can be bounded and unpredictable resizing operations are avoided. Nevertheless, because the elements of a collection must be mutable, users have to provide their own way in which elements can change state. This work focuses on known execution times and memory consumption rather than on ensuring scope-safety.

⁶RTSJ defines `RealtimeThread` and `NoHeapRealtimeThread` in addition to standard Java threads.

The focus of the mentioned studies has been on performance; scope-safety is treated by avoiding assignments to heap memory; or provide safe subsets of classes with non-deterministic behavior. In contrast, our work concentrates on scope-safety by analyzing design patterns and idioms. We also explore deterministic behavior, both for execution time and memory consumption.

2.9 Testing Real-time Features in Real-time Virtual Machines

Corsaro and Schmidt provide in [35] an evaluation of two real-time Java implementations. They compare the Timesys RTSJ reference implementation [127] with their own RTSJ implementation, namely jRate. In that work, Corsaro and Schmidt developed RTJPerf, a synthetic, workload-based benchmark to test time efficiency in an RTSJ compliant (for v1.0.1 of RTSJ at that time) JVM. They provide tests to measure linear time memory allocations, the delay to service asynchronous events, overhead on thread switching and preemption, and the accuracy of timers. A very similar study done by McEnery et al. in [82], provides an empirical evaluation of two main-stream, commercial implementations of RTSJ. They compare Sun's (now part of Oracle Inc.) Java RTS and Aicas' JamaicaVM. McEnery et al. assess the efficiency and predictability of the two commercial implementations by providing tests for memory allocation, thread management, synchronization, and asynchronous event handling.

From these two works we found that some tests, e.g. the linear-time memory tests, are relevant in the context of SCJ and can be adapted to our current evaluation. However, there are other tests that (1) are not applicable to SCJ e.g. the latency to dispatch unbounded asynchronous event handlers from the firing of an event, and (2) use features not allowed in SCJ e.g. RTSJ's `RealtimeThread` and self-suspending methods.

In [43], Doherty provides a benchmark to test RTSJ-based real-time Java implementations. His benchmark, SPECjbb2005rt, based on the SPECjbb2005 benchmark [124], is enhanced to provide throughput and response time metrics. However, SPECjbb2005rt focuses on soft real-time applications and does not make use of RTSJ's features intended for hard real-time systems such as `NoHeapRealtimeThreads` and does not implement tests that use immortal or scoped memory, which are integral parts of SCJ. Moreover, SPECjbb2005rt has not been made publicly available nor has it been adopted by the SPEC corporation. Nonetheless, Doherty's work emphasizes the need for standardized benchmarks that can be used to compare different implementations of (hard) real-time JVMs.

Instead of providing a collection of tests that independently evaluate embedded real-time Java features, in [65], Kalibera et al. present the CDx benchmark, an open source, application-based benchmark that can be adapted to run both on standard and RTSJ compliant VMs. CDx can be used for soft and hard real-time systems as it uses RTSJ's hard real-time features. CDx has one periodic thread used to detect potential aircraft collisions, based on simulated radar frames. The metrics that CDx provides are response time, computation time, and jitter for the collision detector periodic thread. A refactored version of CDx, miniCDj, that uses the SCJ API was developed and used in [96] to test Purdue's L0 SCJ implementation. The miniCDj benchmark was developed for v 0.76 of JSR-302 and the current version of JSR-302 is 0.94. Therefore, some modifications were necessary to run it in our implementation.

Any JSR requires a technology compatibility kit (TCK) and in [136], Zhao et al. describe their initial work towards developing a TCK for SCJ where focus was on functional and behavioral tests. The test cases were derived from SCJ's specification to evaluate features such as the mission life cycle, concurrency and scheduling, memory, clocks and timers, and exceptions. We will use a subset of the tests described in their work, as not all of them can be applied to our implementation e.g. tests that evaluate L2 features.

2.10 Summary

In this chapter we have presented basic concepts related to safety-critical systems. We have provided an overview of programming languages for safety-critical systems and showed that an acceptable approach in the selection of a language for safety-critical software is to limit the features that a well know existing language provides.

We showed what are some of the main problems associated with the use of Java on real-time systems and how the RTSJ solves many of them. As RTSJ is too complex to be used on safety-critical systems, different profiles have proposed for the use of Java on safety-critical systems. Some of those proposed profiles use a restricted subset of the RTSJ (e.g., the Ravenscar-Java profile) while others propose to keep only Ravenscar-style tasking model but abandon the idea of using RTSJ as a basis.

The current proposal to standardize Java for safety-critical systems is the safety-critical Java (SCJ) profile which takes many of the ideas of previous work done on using Java for high-integrity systems. We have described SCJ's basic concepts which are mainly its programming and memory model. Together with this

description we have presented a basic but complete SCJ `Hello World` example to illustrate how an SCJ application looks like.

We have described the Java optimized processor JOP which is our target for our SCJ implementation. We have presented the main architectural features that make JOP a time-predictable embedded processor suitable for an implementation of SCJ.

Finally, we have presented some of the previous work done related to implementing high integrity Java profiles on embedded systems, on the subject of scoped memory reference assignment checks, scoped memory usage, and on evaluation methodologies for real-time Java virtual machines. Those subjects constitute the core of this work.

CHAPTER 3

Safety-critical Java on an Embedded Java Processor

In this chapter we present a detailed description of the implementation of the SCJ profile in the context of a Java processor. Our target is the Java Optimized Processor (JOP) [113] and we cover Level 0 and Level 1 of the SCJ profile. Our implementation is based on version 0.94 of the SCJ specification [77].

This chapter is structured as follows: in Section 3.2 we describe our solution to the issue of using classes that belong to multiple packages (`javax.realtime`, `javax.safetycritical`, and system packages). In Section 3.3 we provide details on the concurrency and scheduling runtime of our implementation, details on the implementation of managed event handlers, and our limited support for SCJ in a multi-core version of JOP. In Section 3.4 we present the scoped memory. In Section 3.5 we present our approach to perform reference assignment checks in hardware. Section 3.5 is based on the following published article: *Hardware Support for Safety-Critical Java Scope Checks* [103]. In Section 3.6 we describe the facilities to interact with external devices, i.e. the raw memory and managed interrupt mechanisms. We finish the chapter by highlighting the main parts of our implementation in the summary of Section 3.7.

3.1 Overview

Our SCJ infrastructure runs on top of the Java processor JOP and therefore it is completely implemented in Java. There is no underlying operating system (OS) and hence we do not depend on any OS service to provide real-time functionality (e.g, scheduling, priority inversion control) or access to low-level memory and JVM structures. Those services are provided by JOP.

We have targeted L0 and L1 of the specification and in Figures 3.1 and 3.2 we show a high level view of the implemented SCJ's concurrency and memory classes. We do not present similar figures for the rest of SCJ's implemented features because Figure 3.1 and 3.2 are the classes that interact the most with JOP's underlying real-time services.

Figure 3.1 shows the dependencies between SCJ's managed schedulable objects and JOP's real-time classes. The `SwEvent` class provides a `fire()` method used to release the software event handler and the `RtThreadImpl` provides the `wait-ForNextPeriod()` method used to block a periodic task until its next release time. The `AperiodicEventHandler`, `AperiodicLongEventHandler`, and `PeriodicEventHandler` make use of those services. Interaction with the memory system, e.g., entering mission or private memories on handler release, is facilitated by the `SysHelper` class.

The implemented memory classes are shown in Figure 3.2. In our SCJ scoped memory implementation we delegate all functionality to the `Memory` system class through the `SysHelper` class. When a `ManagedMemory` object is created, a delegate `Memory` object that represents the actual system memory is also created. As the `Memory` class has only package protected constructors this delegate object is created using the `SysHelper` class. This `Memory` object is used as the argument for the `SysHelper` methods that operate on the system's memory class. Each `Memory` object has a `parent` and in the case of nested private memories the parent will also have an `inner` memory. This class also defines the singleton `immortal` memory.

The rest of the concurrence and memory hierarchy classes inherited from RTSJ is not shown, as they are empty classes in our implementation.

3.1.1 Design Decisions

For our implementation of the SCJ profile, the following design decisions were made:

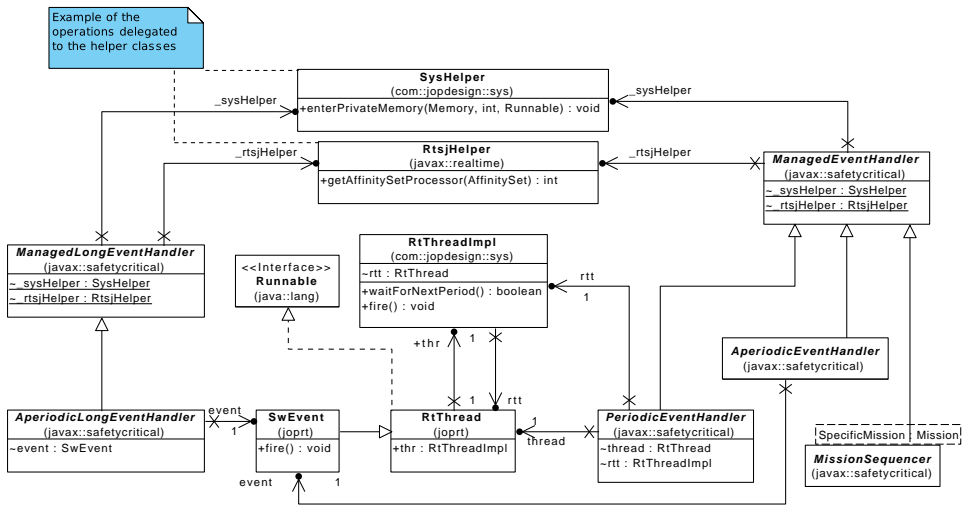


Figure 3.1: Overview of SCJ's concurrency classes implemented in JOP

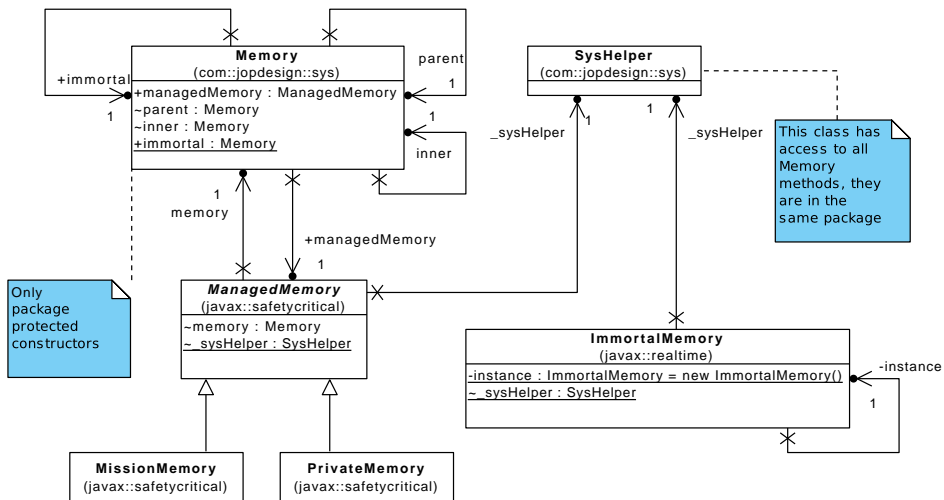


Figure 3.2: Overview of SCJ’s memory classes implemented in JOP

1. *Keep implemented classes in the `javax.safetycritical` package:*

Many of the RTSJ classes that make up the core of the SCJ profile (concurrency and memory) are basically empty classes in SCJ because most of their methods are restricted. Therefore we will keep most of the implementation functionality in the SCJ package.

2. *Delegate VM access to JOP's system classes:*

Some features of the SCJ profile require access to the underlying VM resources, such as the scoped memory model. JOP's system classes provide such functionality. As a consequence we need a way to share package private information between system classes and the SCJ classes. Simply changing accessibility to public is not an option, as such system classes should be restricted from application developers.

3. *Only L0 and L1 features:*

We target only these two levels because of two reasons: (1) their execution model is more familiar for real-time developers and (2) the more dynamic features of L2 will require threads to be constructed and added to the scheduler during mission phase and currently JOP creates all the real-time thread structures before mission execution. Therefore, there is no support for nested missions, managed threads, or self-suspending code (`wait/notify` methods).

4. *The mission sequencer is executed by the main thread:*

L0 and L1 applications can only have a single mission sequencer which is not registered as an MSO of any mission. This means that there is no need to run the sequencer activities under the control of the scheduler, as its execution points are clearly defined (at mission initialization and mission termination). Therefore the main thread is enough for executing the sequencer's activities. It can be considered that in our implementation the sequencer is not a managed schedulable object.

3.1.2 Limitations

The following features that SCJ requires are not part of our implementation:

- `OneShotEventHandler` class is not implemented. Its implementation will require to modify JOP's scheduler in order to schedule an interrupt at some time in the future.
- L2 features are not provided. This means that there cannot be nested sequencers, parallel missions, and `wait()/notify()` methods.

- No user-defined clocks, a feature that should be available at L1.
- There is no proper implementation of the priority ceiling emulation (PCE) protocol. Instead, for a single core version of JOP, priority inversion control is implemented by using a single monitor counter and by disabling all interrupts during a synchronized method.
- No **Happening** and POSIX-related classes are implemented. We do not have an operating system layer therefore we do not need to handle POSIX signals. Furthermore, according to the SCJ specification “Happening represent events that can be triggered based on some event external to the VM”. For our implementation we have first level interrupt handlers in Java and aperiodic event handlers that can provide similar functionality.

3.1.3 Building and Running an SCJ Application in JOP

To run an SCJ application on JOP, such as the `HelloWorld` code presented in Figure 2.2, we have two options: (1) to use JOP’s simulator on a standard PC or (2) to use an FPGA. The build process is automated with a `Makefile` and the command that needs to be executed in a command line window is:¹

```
make { japp | jsim }
```

The `japp` option builds and downloads the application to an FPGA and the `jsim` option builds and executes the application in JOP’s simulator. The following variables need to be configured directly in the `Makefile` or passed as arguments when executing the previous command line:

```
USE_SCOPES = { true | false }
USE_SCOPE_CHECKS = {true | false}
CLASS_OBJECTS = {true | false}
IMM_MEM_SIZE = {size}
P1 = <path to the application Java source>
P2 = <package name>
P3 = <main class>
```

The `USE_SCOPES` argument enables scoped memory usage and disables the GC, the default value is `false`. The `USE_SCOPE_CHECKS` option enables the use of

¹See [112] for further details on the build process of JOP

reference assignment checks, the default value is `true`. The `CLASS_OBJECTS` option enables the generation of Class objects at build time. The size of the application grows when Class objects are used. Class objects are needed in the SCJ implementation to use `MemoryArea`'s `newInstance(Class type)` method and the `SizeEstimator` class. Its default value is `false`. The `IMM_MEM_SIZE` defines the total *usable* immortal memory size that the application will use i.e., this value does not include the amount of memory consumed by VM structures allocated in immortal memory such as runtime class structures, Class objects, constant strings, etc. There is no default value for this parameter.

3.2 Package Crossing

One important restriction in the development of SCJ implementations is that of having classes defined in different packages, making it difficult to share information that is part of the framework but shall not be made public. In an SCJ implementation, there will be at least two packages that need to share information: `javax.realtime` and `javax.safetycritical`. Those packages may require access to implementation specific classes to control, e.g., memory resources. In addition to the SCJ and RTSJ packages, we have core classes such as `Memory` and `Scheduler` defined in the `comm.jopdesign.sys` package, and classes for periodic and aperiodic activities in the `jopr` package. One of our objectives is to keep all the implementation code in the SCJ package thus we need a way to call package protected methods and fields within these four packages without making them public for application developers. We cannot use concepts like the *friend* keyword of C++ or *child* packages of Ada as similar features are not available in Java.

One option to access package private fields and methods without making them public is through the reflection API. SCJ restricts the use of the reflection API from application developers but does not forbid its use by the SCJ infrastructure. However, its use can increase the complexity of validating and testing safety-critical applications as the benefits of compile-time checks are lost because, e.g., we only know at run time if a reflective method invocation will fail.

Another option is to use an approach similar to that used in the `SharedSecrets` class in the `sun.misc` package of a standard Java distribution. The purpose of that class is precisely to access private information without using the reflection API. Access to private methods or fields is accomplished by: (1) defining a public interface that specifies the methods that a *client* class can access, (2) granting access to a *server* class' private information through an implementation of that

interface, (3) accessing the *server* class' private information through an object, accessible through a third package, that implements the interface. To illustrate this approach Figure 3.3 shows an example of how the `java.lang.System` class gains access to private methods and fields of the `java.io.Console` class.

The drawback with that approach is that the third package class (e.g. `SharedSecrets` in `sun.misc`) needs public methods to set and get the interface implementing object that provides the access to the private information. Therefore anyone can set a new object that implements the interface, thus overriding any previously defined object.

In our approach² we define a class that acts as a *proxy* between classes in two packages. The *proxy* class is final, has only a private constructor, and is in the package where we want access to package protected methods and fields. The *proxy* class is instantiated in a static initializer and registered to a *client* class located in a different package. The *client* class can therefore access all the methods that the *proxy* class makes available. An example is shown in Figure 3.4. Here, the `SysHelper` class is the *proxy* class defined in the `com.jopdesign.sys` package. The `ManagedMemory` class is the *client* class registered to use the `SysHelper` object. We see in line 20 that even if the *client* class is registered through a public method, there is no way to create a `SysHelper` object, as it has a private constructor. The drawback is the overhead caused by an additional method invocation.

3.3 Concurrency and Scheduling

SCJ allows only managed schedulable objects, i.e. schedulable objects that are registered to a specific mission. The logic in those objects is executed either by a cyclic executive loop or a fixed priority preemptive scheduler. In this section we provide the details of our implementation of SCJ's concurrency model in JOP.

3.3.1 Missions and Mission Sequencer

Mission execution runs under the control of a mission sequencer that, according to the SCJ specification, is required to be a managed event handler. However, for L0 and L1 applications we can simplify the sequencer. In an L0 application there is only a single thread executing at all times and the cyclic executive loop

²We thank Torur Strøm for the suggestion on the singleton delegator

```

1
2 // In java.lang.System
3 ...
4 public static Console console() {
5     if (cons == null) {
6         synchronized (System.class) {
7             cons = sun.misc.SharedSecrets.getJavaIOAccess().console();
8         }
9     }
10    return cons;
11 }
12 ....
13
14 // In sun.misc.SharedSecrets
15 ...
16 public static void setJavaIOAccess(JavaIOAccess jia) {
17     javaIOAccess = jia;
18 }
19
20 public static JavaIOAccess getJavaIOAccess() {
21     ...
22     return javaIOAccess;
23 }
24 ...
25
26 // In sun.misc.JavaIOAccess
27 public interface JavaIOAccess {
28     public Console console();
29     // Other methods
30     ...
31 }
32
33 // In java.io.Console, the class with the private data
34 // to be shared
35 ...
36 static {
37     sun.misc.SharedSecrets.setJavaIOAccess(new sun.misc.JavaIOAccess() {
38         public Console console() {
39             ...
40             // Console has private constructor
41             cons = new Console();
42             return cons;
43         }
44     });
45     // Other methods
46     ...
47 }
48 ....

```

Figure 3.3: An example of how class-private information is shared using Sun's SharedSecrets class

```

1  // The client class, in com.jopdesign.sys package
2  public final class SysHelper {
3
4      static{
5          SysHelper sysHelper = new SysHelper();
6          ManagedMemory.setSysHelper(sysHelper);
7          ...
8      }
9
10     private SysHelper(){}
11     public void enterPrivateMemory(Memory m, int size, Runnable logic){
12         m.enterPrivateMemory(size, logic );
13     }
14     ...
15 }
16
17 // The client class, in javax. safetycritical package
18 public abstract class ManagedMemory ... {
19     static SysHelper _sysHelper;
20     public static void setSysHelper(SysHelper sysHelper) {
21         _sysHelper = sysHelper;
22     }
23
24     @SCJAllowed
25     public static void enterPrivateMemory(long size, Runnable logic) ... {
26         ...
27         _sysHelper.enterPrivateMemory(current.memory, (int) size , logic );
28         ...
29     }
30 }

```

Figure 3.4: An example of how package-protected information is shared using our singleton delegator

of an L0 mission can therefore be executed by the sequencer thread implemented by the main thread.

For an L1 application, there will be several threads executing concurrently under the control of the scheduler but no nested (concurrent) sequencers, i.e. only the initial mission sequencer which is not required to be registered to any mission. The operations required to execute a L1 mission are thus performed by the main thread (e.g. create and enter the mission memory, do the mission initialization, execution, and cleanup, etc.). The main thread also arranges to start, on the transition to mission mode (after mission initialization), all the event handlers registered to the mission.

For both L0 and L1 applications, after entering mission mode the main thread waits in a polling mode for a mission termination request. Once a mission

termination request is received, L0 applications will be terminated at the end of the currently executing major frame. For L1 applications no more firings of AEHs/ALEHs or releases of PEHs are allowed and only the currently executing managed event handlers will run until completing their current release. AEHs and ALEHs check the current mission's `terminationPending` flag before firing while PEHs check the same flag before scheduling a new release (see Section 3.3.2).

3.3.1.1 Mission change

For L0 and L1 applications, mission change uses a synchronous mode change protocol. In synchronous mode change protocols new-mission handlers are released only after all old-mission handlers have completed their last pending release upon a mission change request [101]. Moreover, the semantics of mission finalization specified in SCJ suggest that a synchronous, single offset protocol [101] to be used, as the release of all new-mode handlers is allowed only after a certain offset, measured from the time a mode change (mission termination) has been requested. The timing of a mission change is as illustrated in Figure 3.5 where the following components are identified:

1. **Termination of last releases of active MSOs:** The time it takes to finish for all active MSOs at the time of the mission change request. In the worst-case, this value will be equal to the worst case response time of the set of old-mission MSOs.
2. **Execution of MSO's `cleanup()` methods:** The time it takes for all MSO's `cleanup()` methods to be executed by the mission sequencer.
3. **Execution of the old mission `cleanup()` method:** The time required for the execution (also by the mission sequencer) of the mission's own `cleanup` method.
4. **New mission initialization:** The time required for the initialization of the next mission to be executed.

The mission change offset is therefore equal to the sum of all the four components described above. The advantage of using a synchronous mode change protocol is that there is no overload during the mission change transition [101] as no new-mission MSOs are allowed until the old ones have terminated.

In an L0 application, when a mission change request arrives, following the semantics in SCJ imply that other PEHs belonging to the current or any other

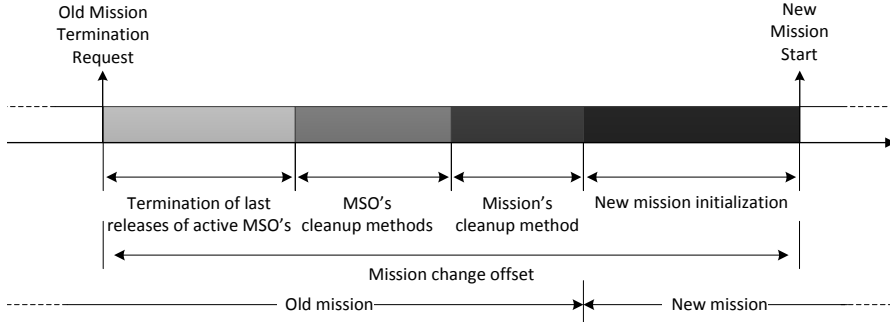


Figure 3.5: Mission change timing for SCJ L0 and L1 applications

frame will not run. This may have an undesired effect if long tasks are divided into smaller tasks that execute in different frames. Mode changes in cyclic executives are usually done at the end of a major frame [128, 18] therefore in our implementation we perform mission change for L0 applications only at the end of the currently executing major frame. The longest delay before the new-mission PEHs can execute (i.e. the offset time in the single offset protocol of [101]) will be equal to the duration of the major cycle plus the time required to run the PEH's and mission's cleanup methods plus the time to initialize the next mission. Executing a new mission in a L0 application is straightforward, the sequencer thread, i.e. the main thread in our implementation, only needs to run the new cyclic schedule loop of the new mission in a resized mission memory.

For L1 applications we follow the mission finalization semantics described in the SCJ specification. As the managed event handlers of a L1 missions are executed by JOP's scheduler, to run new missions we need to (1) remove all the old-mission handlers from the scheduler and (2) add the new-mission handlers to the scheduler. The details of this procedure are covered in Section 3.3.3. The maximum delay in this case will be equal to the sum of the worst-case response time of all handlers executing up to the mission change request plus the time required to run the MEH's and mission's cleanup methods plus the time to initialize the next mission, as depicted in Figure 3.5.

SCJ does not specify any timing requirements for the sequencer as there are no explicit release parameters associated with a `MissionSequencer` object. The only requirement is that it should be a managed event handler and according to the SCJ specification, managed event handlers that do not specify release parameters are considered to be aperiodic (see [77], p. 74).

If there is an upper bound on the delay for the next mission to start its execution, then the execution time of the initialization and cleanup phases of a mission have to be bounded by the application developer. The SCJ specification however assumes that initialization and cleanup phases are used for non time-critical operations and hence implicitly ignores any time constraints on those phases. Consider for example the case where a mission change request occurs due to a failure which will make the system to enter a fail-safe mode of operation or if the system needs to enter a different mode of operation as a result of a change in the environment, e.g. a change from flight to landing mode in an airplane. In both cases the mode transition delay has to be bounded. For an L2 application, mission change may even delay other concurrently executing handlers of a different mission, depending on the priority of the nested sequencer performing the mission change.

3.3.2 Periodic and Aperiodic Event Handlers

As mentioned in Section 2.4.2, the SCJ specification defines two kinds of real-time activities: event handlers and managed threads. Event handlers can be either periodic, i.e. instances of the `PeriodicEventHandler` class, or aperiodic, i.e. instances of the `AperiodicEventHandler` class. Therefore, it is natural to use JOP's real-time classes, the `RtThread` and `SwEvent` classes (see Section 2.5), to implement SCJ's `PeriodicEventHandler` and `AperiodicEventHandler` classes. An extract of the implementation of SCJ's handler classes showing the main computation loop is presented in Figures 3.6 and 3.7 respectively.

In Figure 3.6, the main periodic loop is in the infinite for loop between lines 30 to 42. On every handler release, its associated private memory is entered and the logic of the runnable defined in line 20 is executed. This runnable encapsulates the handler's `handleAsyncEvent()` method that is to be overridden in the application. After a release has completed, a mission termination flag is checked (line 33). If the flag is not set, then the handler is scheduled to be released at its next period by using the `waitForNextPeriod()` method. The `waitForNextPeriod()` method performs late deadline miss detection and returns true if there has not been a deadline miss. If a deadline miss occurs, then the `missCount` counter is increased by one and the `executeMissHandler()` hook is executed.

In safety-critical systems the absence of deadline misses shall be proved before deploying the system. However, the SCJ specification allows miss deadline detection to facilitate the implementation of fault tolerant systems. Therefore, we include the `missCount` variable and the `executeMissHandler()` method as

```

1 public PeriodicEventHandler( PriorityParameters priority ,
2   PeriodicParameters release , StorageParameters storage, String name) {
3   super( priority , release , storage , name);
4
5   this.storage = storage;
6   this.release = release;
7   this.start = this.release.getStart ();
8   this.period = this.release.getPeriod ();
9
10  // Check for overflows, compute a period "p" and an offset "off"
11  // to be used in an RtThread
12  ...
13  // "m" is defined in ManagedEventHandler class
14  m = Mission.getCurrentMission();
15
16  if (!(m instanceof CyclicExecutive)) {
17    privMem = new PrivateMemory((int) storage.maxMemoryArea,
18      (int) storage.totalBackingStore );
19
20    final Runnable runner = new Runnable() {
21
22      @Override
23      public void run() {
24        handleAsyncEvent();
25      }
26    };
27
28    thread = new RtThread(priority.getPriority (), p, off) {
29      public void run() {
30        for (;;) {
31          privMem.enter(runner);
32          // do not schedule this task to run again
33          if (m.terminationPending) {
34            break;
35          }
36
37          if (!waitForNextPeriod()) {
38            missCount++;
39            // implementation specific method
40            executeMissHandler();
41          }
42        }
43      }
44    };
45    ...
46  }
47 }

```

Figure 3.6: An extract of the SCJ's `PeriodicEventHandler` class implementation in JOP.

means for the developer to implement actions in response to missed deadlines.³ Both the deadline miss counter and the miss handler hook can be removed without affecting the functionality of the PEH. Removing such functionality might indeed be necessary when undergoing a certification process in order to avoid introducing dead code if fault tolerance is not a requirement.

For the aperiodic event handler, illustrated in Figure 3.7, the code of lines 24 to 27 is executed on every release i.e., every time the `fire()` method of its associated `SwEvent` is called. As with the PEH, this method enters a private memory and executes a runnable that encapsulates the `handleAsyncEvent()` method that is to be overridden in the application. The constructor of this `SwEvent` (line 22) requires two arguments: (1) its priority, and (2) a minimum inter-arrival time (MIT). The MIT value is required to execute the `SwEvent` at a real-time priority, otherwise ($\text{MIT} = 0$) it is executed at normal priority. SCJ does not allow the use of sporadic events, i.e., aperiodic events with a MIT and a deadline. Therefore there is no MIT violation or deadline miss enforcement and the MIT value passed to the `SwEvent` constructor can be any positive value other than zero.

The implementation of the `AperiodicLongEventHandler` class is similar to `AperiodicEventHandler` except that its release method has an argument of type `long` attached to it. The necessary changes to the `AperiodicEventHandler` implementation are trivial, and require only to pass the `long` argument to a firing of a `SwEvent`.

³Can also be used for debug and testing purposes

```

1  public AperiodicEventHandler( PriorityParameters priority , AperiodicParameters release ,
2                               StorageParameters storage, String name) {
3      super( priority , release , storage, name);
4
5      // "m" defined in ManagedEventHandler class
6      m = Mission.getCurrentMission();
7
8      // Throw exception if trying to add an AEH to a CyclicExecutive
9      ...
10
11     privMem = new PrivateMemory((int) storage.getMaxMemoryArea(),
12                                (int) storage.getTotalBackingStoreSize ());
13
14     final Runnable runner = new Runnable() {
15         @Override
16         public void run() {
17             handleAsyncEvent();
18         }
19     };
20
21     // There is no enforcement for minimum inter-arrival time violations
22     event = new SwEvent(priority.getPriority (), 1) {
23         @Override
24         public void handle() {
25             if (!m.terminationPending)
26                 privMem.enter(runner);
27         }
28     };
29     rtt = event.thr;
30 }

```

Figure 3.7: An extract of the SCJ's `AperiodicEventHandler` class implementation in JOP.

3.3.2.1 L0 Simplifications

In an L0 mission there is only one handler executing at all times and therefore we can provide the following simplifications in our implementation:

- We avoid the creation of individual private memory areas for each `PeriodicEventHandler`. We create only one private memory, size it according to the worst case memory requirements of all `PeriodicEventHandlers` registered in the mission, and reuse it on each handler release.
- We do not need to create the additional runnable and `RtThread` objects (lines 20 and 28 in Figure 3.6) to execute each `PeriodicEventHandler`. For cyclic executives, the handler's `handleAsyncEvent()` method is called directly by the main thread, without the intervention of JOP's scheduler, according to the `CyclicSchedule` returned by the cyclic executive's `getSchedule()` method.

L0 missions allow only for the execution of PEHs. To enforce this restriction, trying to add an AEH to a L0 mission will throw an `IllegalArgumentException()`. This check is performed in the constructor of the `AperiodicEventHandler` class (the omitted code, replaced by the ellipsis in line 9 of Figure 3.7). Another option to enforce this restriction could be to silently ignore the AEHs when they are registered to a L0 mission. However, silently ignoring it will still allow the creation of the `AperiodicEventHandler` object, its corresponding `SwEvent` object, and the memory required by the stack of the `SwEvent`, as the memory space for all of those objects is taken from the memory area where the `AperiodicEventHandler` is created (i.e. `MissionMemory`).

The check performed at line 16 in Figure 3.6 is used to enforce those simplifications.

3.3.3 Scheduler

The scheduler in JOP is implemented in Java as a first level interrupt handler attached to the programmable timer interrupt. In general, for an N-core version of JOP there will be one scheduler object per core with the structure shown in Figure 3.8. Besides the timer interrupt, the interrupt handler can be triggered in software by: (1) a call to `waitForNextPeriod()` at the end of a `PeriodicEventHandler` release, and (2) completing a release of an `AperiodicEventHandler`. The `waitForNextPeriod()` method preforms deadline miss detection of

the `PeriodicEventHandler` and triggers a software interrupt that executes the scheduler. When an `AperiodicEventHandler` completes, it blocks itself and, as the `PeriodicEventHandler`, a software interrupt is scheduled. The timer interrupt is reprogrammed on every invocation of the scheduler to fire at the next release time of the highest priority thread.

As can be seen in Figure 3.8, each scheduler object uses three different arrays that contain references to the next release times for threads (*next*), software event state (*event*), and references to `RtThreadImpl` objects (*ref*). There is also a priority ordered linked list of `RtThreadImpl` objects used to assign the elements of the arrays during the transition to mission mode. Elements in the linked-list are added as event handlers are registered into a mission meaning that unregistered handlers will not be executed, as required by the SCJ specification.

Every time the scheduler is called, its `run()` method is executed and the following actions take place: (1) the context of the current thread is saved, (2) the priority ordered array of threads is scanned and the highest priority thread ready to run is selected, and (3) the context of the new thread to run is restored.

Because the managed event handler objects are allocated in mission memory during execution of the mission's `initialize()` method, we also allocate the scheduler, the arrays, and the linked-list in mission memory in order to avoid illegal references. There are however, certain objects we cannot allocate in mission memory, such as the interrupt handler array and the array of schedulers. This means that there will be some illegal reference assignments we cannot avoid. Nonetheless, those can be regarded as “safe” illegal assignments because we can be sure that the referred objects in mission memory will be deallocated only after they are no longer required i.e. when a mission is finished. This approach works both for L0 and L1 applications because at L0 we do not use the scheduler at all and at L1 there is only one mission executing at all times.

We can take advantage of these objects being allocated in `MissionMemory` when changing missions as they will be collected once we leave mission memory. To start a new mission, we need to recreate the thread linked list and the different arrays which can be done in the transition to the mission mode. When we leave mission memory, no `ManagedEventHandler` is executing as any future release is prohibited after a request to terminate the currently executing mission thus making it safe to collect the scheduler object and related data structures.

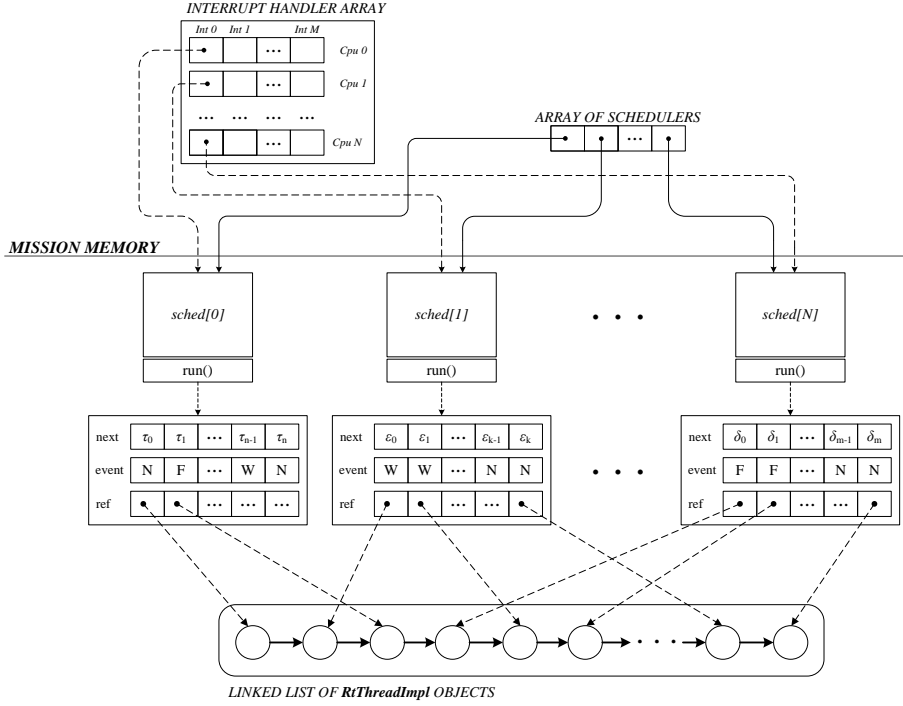
IMMORTAL MEMORY

Figure 3.8: Location of JOP's scheduling run-time structures within SCJ's memory areas

3.3.4 Multi-core Support

Multi-core support is provided in SCJ through scheduling allocation domains. A scheduling allocation domain consists of a group of processors on which an MSO can be executed. For an L0 application, there is a single allocation domain with a single processor. On an L1 application, there can be more than one allocation domains but each of them must have a single processor and therefore MSOs will form a fully partitioned system. For L2 applications, allocation domains can have more than one processor but each processor must not be shared among allocation domains. The MSOs of an L2 application are scheduled using global scheduling. Scheduling allocation domains are represented by affinity sets which are nothing more than a way to assign MSOs to be executed in specific processors.

For our SCJ implementation, by default each MEH is assigned to an affinity set whose processor is processor number zero. It is possible to change this default assignment but it has to be done before the MEH is registered to its enclosing mission during the mission initialization phase. Trying to change the affinity after the register method has been called has no effect.

For a CMP version of JOP, only core 0 executes the main thread, all other cores execute runnables that are created at JVM boot time. The runnables in the other cores act as a “main” thread and hence they are in charge of performing the mission initialization tasks corresponding to each other core, such as registering the core’s scheduler object (see Figure 3.8) to the timer interrupt. After performing these initialization tasks, the runnables wait until a mission termination request. Mission termination requires that all managed schedulables running in different cores be informed about a termination request. This is done by using a static volatile variable that is set when any handler calls the `requestTermination()` method. The sequencer then checks for all handlers to finish executing their last release and informs all other cores about this event. The `cleanUp()` methods of the managed schedulables are called from the sequencer’s event handling thread as required by the SCJ specification therefore only core 0 will execute the `cleanUp()` methods.

Being able to call the `cleanUp()` method from the core where the managed schedulable object was executed can be advantageous if the `cleanUp()` method is used, e.g., to reset the state of a peripheral device connected only to that core. However, as the SCJ specification requires that a private memory area is provided for the execution of the `cleanUp()` methods (probably for its reuse by every cleanup method), we will need to create a private memory for each core in order to execute cleanup methods of its handlers thus increasing the requirements on the mission memory size. This additional memory will then have to be considered when sizing the mission memory.

After all the cleanup methods of all MSOs are executed, the mission cleanup method is then executed and the sequencer exits the mission memory, removing all data structures allocated in mission memory, including all scheduler objects.

3.4 Memory

SCJ defines three different classes for its memory model. However, all of them can be implemented with a single **Memory** class as presented in [114]. For our implementation the different SCJ memory classes delegate functionality to this system’s **Memory** class. As mentioned before, we want to keep all of the SCJ

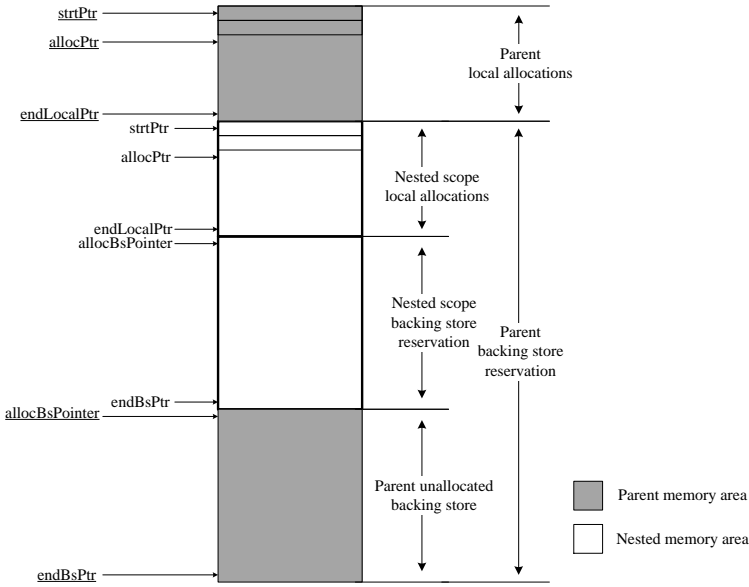


Figure 3.9: Overview of the scoped memory layout in JOP.

implementation in the `javax.safetycritical` package. Therefore, all of the memory related classes in the `javax.realtime` package are basically empty classes. The exception is the `ImmortalMemory` class but all of its methods are delegated to the `Memory` class.

The scoped memory layout is shown in Figure 3.9. Each `Memory` instance represents a memory area and uses five pointers. The region between `strPtr` and `endLocalPtr` represents the local allocation space, i.e., the memory space where objects will be allocated when the current execution context is that memory area. The region between `endLocalPtr + 1` and `endBsPointer` is the reserved backing store, i.e., the memory space from where memory for nested scopes is taken. The two additional pointers, `allocPtr` and `allocBsPtr` indicate the next available memory word for object allocations and nested scope allocations respectively. Figure 3.9 represents the situation where the grayed memory area is the parent of the white memory area. The underlined pointers correspond to the parent memory (i.e. the grayed area) and the non-underlined pointers correspond to the nested memory.

In JOP, the `Memory` class is a core class used not only in the SCJ implementa-

tion but also by other system classes such as the GC⁴ when objects are created. Therefore, application developers should not be allowed to access it. To be able to access it from two different packages (the `javax.realtime` package for `ImmutableMemory` and the `javax.safetycritical` package for `MissionMemory` and `PrivateMemory`) we use the singleton delegator class `SysHelper` (see Section 3.2). See Figure 3.2 for a class diagram of the memory hierarchy in our implementation.

Immutable memory is created at application initialization where a portion of the total system's memory is reserved for immortal allocations. Its value has to be set as a parameter in the configuration file that builds the application (see Section 3.1.3). The value returned by the safelet's `immutableMemorySize()` method is compared against that predefined value to check if there is enough immortal memory to fulfill the application demands. The remaining system memory becomes the backing store for the mission memory.

Mission memory is initially sized according to the `StorageParameters` of the sequencer and later resized to the current mission's memory requirements. The resized portion becomes the available mission memory for local allocations and the remaining memory is the backing store for private memory allocations.

MSO's private memories are created as MSOs are created (see Figures 3.6 and 3.7, lines 19 and 11 respectively), their local allocation size is the value of the `maxMemoryArea` argument and their total size (local allocation plus reserved backing store for nested private memories) is the value of `totalBackingStore` argument, both argument of the `StorageParameters` object passed to the constructor of `PeriodicEventHandler`, `AperiodicEventHandler`, and `AperiodicLongEventHandler`. Nested private memories are logically and physically nested within its parent scope and their backing store space is created by taking all the remaining backing store portion of its parent (`totalBackingStore - maxMemoryArea`).

3.5 Scope Checks

The different scoped memories in SCJ (see Section 2.4.3) store objects with different lifetimes. Immutable memory is used for objects that will live for the whole VM lifetime, mission memory is used for objects that will live for the mission lifetime, and private memories are used for objects that live for the

⁴SCJ does not allow the use of a GC however, the original implementation of JOP has a GC. It can be disabled when using scopes but the code to create objects with the `new` and `newarray` bytecodes is in the GC class.

duration of a MSO release. As there is no GC in SCJ, programmers have to be aware of where objects are allocated, thus increasing the risk of leaving dangling pointers by storing references to objects no longer existent. Therefore, the JVM must check the referential integrity by ensuring that objects allocated in a memory area only store references to objects allocated either in the same or in a longer lived memory area. Enforcing this referential integrity thus becomes a source of execution time overhead for an application. In this section, we examine how, given the simplified memory model of SCJ, a single scope nesting level can be used to check the legality of every reference assignment. We also show that with simple hardware extensions we can check reference assignments without the overhead of a software based solution and improve the execution time of applications with frequent reference assignment operations.

3.5.1 Referential Integrity and the Scope Stack

In the RTSJ, there are two operations that need to be performed to guarantee referential integrity: (1) *every time a scoped memory is entered*, the JVM must check that the scoped memory has been entered from its parent scoped memory, this is known as the single parent rule, and (2) there are specific reference assignment rules that need to be checked on *every field reference assignment*. The scoped model of SCJ significantly reduces the complexity of ensuring the referential integrity. This complexity reduction comes from two restrictions imposed on SCJ's memory model: (1) scopes are private to each MSO and (2) applications cannot enter arbitrary memory areas. These two restrictions make RTSJ's single parent rule implicitly satisfied. The result is that when an MSO enters a scoped memory there is no need to check if the MSO has entered every ancestor of that scoped memory area (i.e., a `ScopedCycleException` cannot be thrown).

The different scoped memories that an MSO has entered are logically organized as a linear stack, with scoped memories holding longer-lived objects nested at deeper levels. The shared memory areas (immortal and mission memory) are at a fixed nesting level within this stack and it is not possible that two MSOs see the same scope in its scope stack at different levels. It is possible however that different MSOs see the same level associated with a different scope. This is not a problem as MSOs can only execute in the scoped memories that are part of its scope stack.

Therefore, a unique nesting level can be assigned to each scope and checking the reference assignment rules is reduced to a comparison between the nesting levels of the source and destination objects in a reference assignment. Figure 3.10 shows how the parenting relationship of scoped memories look in an SCJ imple-

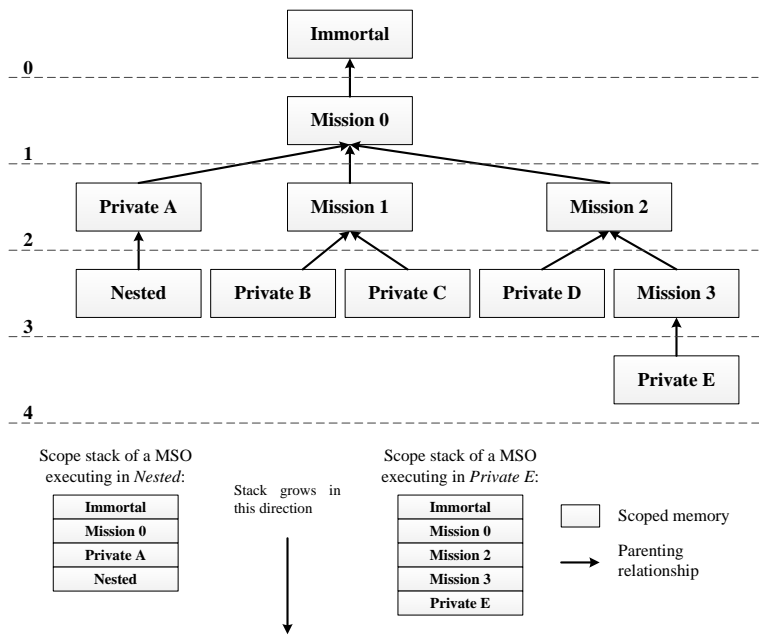


Figure 3.10: Scoped memory hierarchy and a possible scope nesting level assignment (the numbers to the left).

mentation. The numbers to the left represent the scope level nesting and the arrows point to the parent of a scoped memory. In the bottom of Figure 3.10 we can see how the scope stack of two MSOs that are currently executing in the memories labeled as *Nested* and *Private E* looks like. From the figure, we can see that the following scope levels can be assigned to each memory area: immortal memory is level 0, the initial mission memory is level 1, the private scope of a handler is level 2, and nested memories increase the level by 1. For an SCJ L2 application the MSO’s scope stack will also have a linear shape and, provided that inner mission memory objects cannot be leaked to an outer mission handler, the level check is also an option for SCJ L2. In addition, one should be able to guarantee that parallel missions share data only through immortal memory or outer mission memory.

Table 3.1 shows a summary of the valid references in SCJ. The table should be read as follows: an object allocated in one of the scoped memory areas in the leftmost column is only allowed to store a reference to an object allocated in one of the scoped memory areas in the topmost row if the corresponding cell has a check mark. More specifically, objects allocated in immortal memory can

Table 3.1: Valid references in SCJ. The column on the left represents the source memory area and the row on the top represents the target memory area. A check mark represents a valid reference.

Stored in	Reference to Immortal	Reference to Mission	Reference to Private	Reference to Nested
Immortal	✓	✗	✗	✗
Mission	✓	✓ ¹	✗	✗
Private	✓	✓ ²	✓ ³	✗
Nested	✓	✓ ²	✓ ⁴	✓ ³

¹ Only to the same or to the mission memory of an enclosing mission.

² Only to the mission memory of an enclosing mission.

³ Only to the same scoped memory.

⁴ Only to the parent private memory.

only refer to other objects in immortal memory. Objects in mission memory are allowed to refer only to objects allocated either in immortal memory, in the same mission memory, or in the mission memory of an enclosing mission. Objects allocated in private memories can only refer to objects allocated in the same private memory, to objects allocated in an enclosing mission memory, or to objects in immortal memory. Finally, objects in nested scoped memories can only refer to objects in the same nested memory, to the parent private memory, to an enclosing mission memory, or to immortal memory.

Any non-detected violation to these reference assignment rules can leave a reference that points to a memory area with unexpected content producing unexpected results in the execution of an application that can potentially crash the system.

3.5.2 Detecting Illegal Reference Assignments

Our approach to detect illegal assignments is similar to the approach described in [59] that enforces a write barrier upon the execution of instructions that store an object's reference into another object's field or array element. According to the JVM specification [74] those instructions are the `putfield`, `putstatic` and `aastore` bytecodes and when they are executed, the operand stack of the JVM has the following information [74]:

```

putfield:    ..., objectref, value
putstatic:   ..., value
aastore:     ..., arrayref, index, value

```

where the top of the stack (TOS) is the rightmost quantity (the ellipsis indicate don't care values). Note that for the `putfield` and `putstatic` bytecodes, illegal assignments can be produced only when *value* contains a reference to another object meaning that we do not need to check every field assignment. JOP's application build tool provides an optimization that substitutes field assignment bytecodes of references with the special bytecode versions `putfield_ref` and `putstatic_ref`. These special bytecodes are implemented in Java. Therefore, in addition to the `aastore` bytecode, checks are only executed in these special bytecodes and field assignments of primitive values have no additional overhead.

At object creation, whenever the `new` or `newarray` bytecodes are executed, every object is associated with the level of the memory region where it is created (the current allocation context). This information can be stored in the object's auxiliary data. For our implementation we experimented with two possibilities on where to add the scope level information: (1) in the object's header and, (2) in the object's reference. Furthermore, we improved the second option with simple hardware extensions.

Depending on which of the previously mentioned bytecodes is executed, one of the following three simple rules have to be checked to detect illegal memory references. Shorter lived scopes are deeper nested in the stack hierarchy and associated with higher level numbers:

1. The `putfield` bytecode stores *value* into a field of an object whose reference is *objectref*. The reference check has to verify that the level of *value* is less than or equal to the level of *objectref*.
2. For `putstatic`, *value* cannot be in a scoped region, hence the level of *value* (pointer to the object) needs to be checked against level 0, as all static fields reside in immortal memory.
3. The reference check for `aastore` is similar to the test for `putfield` but using the level of *arrayref* instead of the level of *objectref*.

3.5.2.1 Header Based Scope Checks

Since the JVM specification does not require any particular internal structure for objects [74], the additional information can be stored as part of the header of

the object itself. Following the idea exposed in the previous section, at creation time a particular region of the object or array auxiliary data is associated with the scope level, and whenever a reference assignment instruction is executed, this value can be extracted from the object's header. The advantage of this approach is that it can be used by any JVM implementation and is not restricted to be used just in JOP. The drawback is that for each reference assignment check two additional accesses to memory are required to retrieve the scope level information of each object.

The object and array layout of JOP uses an indirection, called a handle. The handle area holds information which usually is part of the object header, e.g., type information, GC information, size, etc. When an object is created, a reference to the object or array handler is obtained. The reference is pushed into the stack before an instruction can operate on it. Within the SCJ implementation on JOP, the GC is disabled and therefore, we can reuse one of the handle fields used by the GC to store the scope level. Figure 2.4 shows the object and array layout in JOP.

3.5.2.2 Pointer Based Scope Checks

Depending on how much memory is available to the system and how data is aligned in memory locations, some bits of an object or array reference can be left unused. Therefore the scope level can be encoded in the unused bits. Having the scope level as part of the reference saves at least the two additional memory reads on the assignment check mentioned in the previous section. Similar to the object header based approach, the pointer based scope checks can be used in any JVM implementation when the maximum memory is restricted.

Addresses in JOP are 32-bit wide and memory is addressed as 32-bit words. This combination allows a maximum heap size of 16 GB of memory, which is more than what an embedded application will probably require. We can therefore use some of the upper bits of the reference to an object's handler, to store the scope level information of each object. We used 7 of the 32 bits of the object's reference pointer to encode the scope level. This allows for the use of 128 levels of nested scopes.

3.5.2.3 Hardware Based Scope Checks

The two methods described in the previous sections are suitable for any JVM, as the data structures required are not implementation specific. However, a

hardware based implementation of the scope checks implies certain knowledge of the underlying construction of the VM and will be implementation specific (e.g. [59] for the picoJava-II microprocessor). For our implementation in JOP, we will use the pointer based method enhanced with simple hardware extensions.

Because in JOP most of the JVM is implemented in hardware, all the information to perform the scope checks can be directly accessed in hardware. The scope level of the objects pointed by *objectref/arrayref* and *value* can be recovered from the upper bits from the registers holding the top of the stack (TOS) and the next of the stack (NOS) during the execution of the relevant bytecodes.

Reference assignment checks can now be efficiently performed since this is reduced to a simple arithmetic comparison between scope levels. This comparison can be implemented in dedicated and simple hardware. In addition, because we have all the information available during bytecode execution, the check itself can be included as part of the bytecode execution performed in the Memory Management Unit (MMU). Recall that the check shall only be done when it is an assignment of a reference, so the hardware needs to know if it is a reference or a primitive data. That information is available on special versions of the *putfield/putstatic* bytecodes and in the *aastore* bytecode. We only need a new microcode instruction to signal the MMU that whenever the mentioned instructions are executed a reference assignment will take place and thus the levels need to be checked.

As the check is performed as part of the bytecode execution, there is only a minimal overhead of one clock cycle, which is the time needed to indicate a reference assignment bytecode to the MMU. There is also the extra cost of adding the scope level information at object creation time. Nevertheless, object creation is an operation with low frequency of execution [111]. An interrupt is generated inside the MMU when an illegal assignment occurs. This flag is used to throw an *IllegalAssignmentError*.

For our pointer and hardware based solution, using some bits of the reference pointer does not affect the behavior of the system because internally, a 32-bit memory address is trimmed to the width of the memory bus attached to the system. Therefore the scope level information is not used when accessing objects in memory. The use of bits that belong to the reference pointer results in a trade-off between the number of scope levels that can be used and the amount of memory that can be addressed by the system. However, it is unlikely that typical SCJ applications will use many levels of nested scoped memories. In L0 and L1 applications, a minimum of two bits will cover for the levels of the immortal, mission, MSO's initial private memory, and one nested private memory per MSO. In an L2 application however, one should also consider that nested missions will require additional bits to encode the scope level, e.g., three

Table 3.2: Execution time (in clock cycles) of the three bytecodes implementing the scope check methods described in this thesis.

Bytecode	Execution Time			
	<i>Handle</i>	<i>Reference</i>	<i>Hardware</i>	<i>None</i>
putfield_ref	185	169	13	12
putstatic_ref	163	157	8	7
aastore	179	163	15	14

bits will be needed to encode the levels illustrated in Figure 3.10 where three missions execute concurrently.

3.5.3 Evaluation

Our proposal was tested by implementing in JOP the three scope check methods described in this section. We used a micro benchmark where we execute the bytecodes performing reference assignment checks, i.e., the **putfield_ref**, **putstatic_ref**, and **aastore** bytecodes. In this micro benchmark we measured the execution time of each mentioned bytecode.

Each bytecode has a different individual implementation and therefore a different individual execution time. Table 3.2 shows the execution times in clock cycles of each of the three aforementioned bytecodes with the different options to perform the scope checks described in this section. The values were obtained using ModelSim by performing a VHDL simulation of JOP. From the table it can be seen that the hardware version is around 10 times faster than the two software versions. Furthermore, the cost in hardware of adding the scope checks is practically negligible as it adds around 32 look up tables and 10 registers when implemented in an Altera DE2-70 FPGA board. This is an increase in about 4% of the MMU where most of the logic is implemented.

To evaluate the improvement within a more complete application, we used two example applications. The first is an application inspired by one of the examples presented in [33], where a probe containing a number of sensors is used to scan the walls of a well. In our example application, we periodically check a set of artificial sensors simulated in hardware using hardware objects as described in [115]. The application creates an array of objects to hold the sensor's results in a scoped memory. The application then enters a nested scope where the actual sensor objects are created; each sensor performs readings and basic calculations

Table 3.3: Sensor application execution time including the three versions of the scope checks.

Sensors	Reference count	Execution time (ms)			Improvement
		<i>Handle</i>	<i>Reference</i>	<i>Hardware</i>	
50	864	14.05	13.90	11.35	18.35%
100	1714	28.00	27.60	22.65	17.93%
150	2564	42.15	41.40	33.85	18.24%
200	3414	56.10	55.15	45.10	18.22%
250	4264	70.20	68.90	56.45	18.07%
				average	18.16%
				std. dev.	0.16%

and stores back the results into the array of objects located in the parent scope. The second application is a scoped version of an N-body simulation (gravitational force). It uses a "brute-force" algorithm, where the resulting force on each body is computed as the result of the field interaction of each body. This application was adjusted to have a number of cross-scope references proportional to the number of bodies and the total time steps used for the simulation.

The detailed results of the two benchmarks are summarized in Tables 3.3 and 3.4. The improvement gain on both tables is the difference in execution times of the hardware based implementation and the object reference software implementation (the reference based implementation is slightly faster than the handler based). Table 3.3 shows an improvement gain of around 18% for the sensor application while Table 3.4 shows that the improvement gain is roughly 0.09% for the N-body simulation.

3.5.4 Discussion

The focus of this section was to evaluate the benefits of providing hardware support for time critical operations such as reference scope checks in the context of SCJ. We found that our hardware implementation adds a minimal timing overhead of one clock cycle at a negligible hardware cost to this operation because the execution of the write barrier is part of the execution of the bytecode itself.

Nevertheless, reference assignment between objects may not be too frequent in a real application and hence the hardware scope check will not make a major im-

Table 3.4: N-Body simulation application execution time including the three versions of the scope checks.

Bodies	Reference count	Execution time (s)			Improvement
		<i>Handle</i>	<i>Reference</i>	<i>Hardware</i>	
2	619	1.635	1.635	1.634	0.10 %
3	1219	4.045	4.045	4.042	0.08 %
4	2019	7.519	7.516	7.511	0.07 %
5	3019	12.050	12.052	12.040	0.10 %
6	4219	17.648	17.648	17.634	0.08 %
				average	0.09 %
				std. dev.	0.01 %

provement in the execution time. In the two examples provided, the execution time was measured with different number of reference assignment operations. For the sensor application, increasing the number of sensors increases the number of references and in the N-body simulation, increasing the number of bodies and/or the time steps for the simulation increases the reference count. For the sensor application, it doesn't make too much sense to have hundreds of sensors since a real application most likely won't need that much. In the N-body simulation it does make sense to have many bodies and hence many reference assignments but their overhead is clouded by the execution time of the rest of math operations.

In the sensor application we can see an improvement gain of around 18% because it does not have heavy computations, it only reads data from the simulated sensors and performs simple calculations on the sampled sensor data. With the N-body simulation application, the improvement is very small as most execution time is spent in floating point operations, which are slow on JOP.

3.6 Interaction with Devices and External Events

Embedded real-time systems need to interact with their environment. To do so, SCJ inherits the concept of raw memory areas, to provide low-level access to physical memory, and first level interrupt handlers in Java, to respond to external events. In this section both facilities and their implementation in JOP is presented.

3.6.1 Raw Memory

Access to physical memory is supported in SCJ through RTSJ's raw memory API classes⁵. The actual access to memory locations is enforced by *accessor objects*. There is one type of accessor object for each primitive data type. Accessor objects are created by factories and these factories allow access to specific types of memory (e.g., IO memory mapped, IO port mapped). factories are registered and obtained from a raw memory manager.

3.6.1.1 Raw Memory Accessor Objects

To implement raw memory accessor objects we have two choices: (1) to use JOP's native methods, or (2) to use hardware objects or arrays [115]. For the first option the **RawMemory** accessor objects (e.g., **RawInt**, **RawIntRead**, etc) delegate the read to and write from memory operations to static native methods. The second option can use either hardware objects or hardware-based arrays.

In addition to native methods, access to external memory in JOP is also possible with *hardware objects* and *hardware arrays* [115]. Hardware objects are used to map external memory locations to object fields. The handle indirection of a hardware object points to the base address of an external memory location. The **getfield** and **putfield** bytecodes are therefore used to read from and write to the memory locations that the hardware object points to. In a similar way, in a hardware array the array elements are mapped to external memory locations. An example showing how to implement the accessor objects using native methods, hardware arrays, and hardware objects is provided in Figures 3.11a, 3.11b and 3.11c respectively.

In Figure 3.11a we see how the **put** and **get** methods of a raw memory accessor object are mapped to static native methods. In Figure 3.11b, a hardware-based array (line 3) is used. Hardware arrays are created in JOP as singleton objects⁶ using a *hardware object factory*, according to a given base address, with a predefined length, and in a static initializer (i.e., they will be located in immortal memory). The **put** and **get** methods of the raw memory accessor object write to and read from the array element at the position indicated by the **offset** argument (lines 4 and 10). For the code in Figure 3.11c, the raw memory accessor object is also a hardware object. It has a single field (line 4) that will be mapped to a memory location when this object is created (also as

⁵RTSJ v. 1.1

⁶In this way we can make sure that only one object instance will access a particular memory region

a singleton using a factory).

3.6.1.2 Usage

The code in Figure 3.12 shows how the raw memory API can be used in a controller for a memory mapped IO device. In this example, the device is an I2C bus controller with several control and status registers. First, the SCJ infrastructure creates a factory that allow access to `IO_MEM_MAPPED` memory areas through accessor objects. The `IOMemMappedFactory` factory in line 2 knows how to create these accessor objects (e.g., instances of `GenericRawMemAccessorHwArray` from Figure 3.11b) The factory is then registered with the raw memory manager, i.e., the `RawMemory` final class (line 3). During the registration process, the raw memory manager ensures that no other factory that allows access to `IO_MEM_MAPPED` raw memory areas has been previously registered, otherwise an `IllegalArgumentException` is thrown.

The application is then able to request from the raw memory manager access to `IO_MEM_MAPPED` raw memory areas. In the example of Figure 3.12, the `I2CBusController` class requests the creation of accessor objects that are used to manipulate the registers of the I2C bus controller device. The accessor objects are created when an `I2CBusController` object is instantiated, as part of its constructor as shown in Figure 3.13. In Figure 3.13 we show two possible approaches to gain access to the device's registers. Figure 3.13a shows the case when the accessor objects representing the I2C device registers are implemented as a single object per register, either using native methods or hardware objects (e.g., lines 11 and 13). In contrast, in Figure 3.13b a single accessor object representing an array is used to group all the IO registers (line 10).

3.6.1.3 Discussion

From the three ways of implementing accessor objects (Figure 3.11), using native methods is the most flexible option. There is no need for additional hardware object factories and we can read and write to any memory area. The other two options are useful if we want to avoid having native methods in application code as such methods break Java's type safety.

The raw memory API allows accessing physical memory areas as Java objects and therefore have the benefits of Java's type safety. However, the raw memory API adds an overhead when compared to a simple solution that uses hardware objects. We see that there is an extra method invocation for reading and writing

```

1  class GenericRawMemAccessorNative implements RawInt {
2
3      private int address;
4
5      public IODeviceHwArray(int address) {
6          this.address = address;
7      }
8
9      public int get() {return Native.rdMem(address);}
10     public void put(int value) {Native.wrMem(value, address);}
11 }

```

(a) Raw memory accessor object using native methods.

```

1  class GenericRawMemAccessorHwArray implements RawIntArray{
2
3      int [] data; // This is a hardware array
4
5      public int get(long offset) {return data[(int) offset]; }
6      public void get(int [] array) {
7          for(int i = 0; i < data.length; i++)
8              array[i] = data[i];
9      }
10     public void put(int value, long offset) {data[(int) offset] = value;}
11     public void put(int [] array) {
12         for(int i = 0; i < data.length; i++)
13             data[i] = array[i];
14     }

```

(b) Raw memory accessor object using hardware arrays.

```

1  class GenericRawMemAccessorHwObject
2      extends HardwareObject implements RawInt {
3
4      private volatile int data; // This field is mapped to a raw memory address
5
6      public int get() {return data;}
7      public void put(int value) {data = value;}
8  }

```

(c) Raw memory accessor object using hardware objects.

Figure 3.11: Example of three implementations of raw memory accessor objects.

```

1  // SCJ infrastructure
2  IOMemMappedFactory factory = new IOMemMappedFactory();
3  RawMemory.registerAccessFactory(factory);
4  ...
5
6  // Application code
7  I2CBusController i2c_a = new I2CBusController(Const.I2C_A_BASE);
8  I2CBusController i2c_b = new I2CBusController(Const.I2C_B_BASE);
9
10 i2c_a.initialize (100, true);
11 i2c_b.initialize (75, false);
12
13 i2c_b.writeBuffer(new int[] { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 });
14 i2c_a.writeRead(75, 100, 5);

```

Figure 3.12: An example of using the `RawMemory` API to provide low level access to an IO device.

physical memory locations. Moreover, if the accessor object is not of the array type, a single object per memory location is required while a single hardware object can provide access to several memory locations by having more than one field. Nonetheless, due to the simple structure of an accessor object, they can easily be inlined as they basically provide setter and getter methods.

3.6.2 Managed Interrupts

Interrupts in SCJ are instances of the `ManagedInterruptServiceRoutine` (MISR) class that extend RTSJ's `InterruptServiceRoutine` class. As with MSOs, MISRs need to be registered to a mission. The actual interrupt service routine (ISR) code is implemented in the `handle()` method of RTSJ's `InterruptServiceRoutine` class.

Similar to the RTSJ 1.1, SCJ supports the notion of first level interrupt handlers in Java. In Java, an ISR can be implemented as a handler or an event [115]. The handler approach uses a method invoked by the hardware while the event approach uses a form of asynchronous event to fire an AEH or unblock a managed thread. The advantages and disadvantages of both methods are described in more detail in [115]. The main code of our MISR implementation is shown in Figure 3.14. A MISR has three elements: (1) a private memory area to execute its code (line 6), (2) a runnable that executes the `handle()` method in this private memory (lines 9 – 18), and (3) a runnable registered as first level interrupt handler (lines 20 – 25). JOP uses an array of runnable objects to attach plain Java objects implementing the `Runnable` interface as interrupt handlers. The

```

1  public class I2CBusController {
2      // configuration constants
3      private static final int CONTROL_OFFSET = 0;
4      private static final int STATUS_OFFSET = 1;
5      ...
6      // A single accessor object represents a single IO register
7      private RawInt control;
8      private RawIntRead status;
9      ...
10     public I2CBusController(int baseAddress) {
11         control = RawMemory.createRawIntInstance(RawMemory.IO_MEM_MAPPED,
12                                                    baseAddress + CONTROL_OFFSET);
13         status = RawMemory.createRawIntReadInstance(RawMemory.IO_MEM_MAPPED,
14                                                      baseAddress + STATUS_OFFSET);
15         // other registers
16         ...
17     }
18     public void setControl(int value) {control.put(value);}
19     public int readControl() {return control.get();}
20     public int readStatus() {return status.get();}
21     ....
22 }

```

(a)

```

1  public class I2CBusController {
2      // configuration constants
3      private static final int CONTROL_OFFSET = 0;
4      private static final int STATUS_OFFSET = 1;
5      ...
6      // The whole IO device is represented by a single RawIntArray accessor object
7      private RawIntArray i2cPort;
8      ...
9      public I2CBusController(int baseAddress) {
10         i2cPort = RawMemory.createRawIntArrayInstance(RawMemory.IO_MEM_MAPPED,
11                                                        baseAddress, 0);
12     }
13     public void setControl(int value) {i2cPort.put(value, CONTROL_OFFSET);}
14     public int readControl() {return i2cPort.get(CONTROL_OFFSET);}
15     public int readStatus() {return i2cPort.get(STATUS_OFFSET);}
16     ....
17 }

```

(b)

Figure 3.13: Example of an I2C bus controller implemented using raw memory. The accessor objects representing the I2C device registers can be implemented as a single object per register (a) or as a single accessor object representing an array (b).

```

1  public ManagedInterruptServiceRoutine(StorageParameters storage, String name) {
2
3      this.storage = storage;
4      ...
5
6      privMem = new PrivateMemory((int) this.storage.maxMemoryArea,
7                                  (int) this.storage.totalBackingStore);
8
9      final Runnable isr = new Runnable() {
10         @Override
11         public void run() {
12             try {
13                 handle();
14             } catch (Exception e) {
15                 unhandledException(e);
16             }
17         }
18     };
19
20     firstLevelHandler = new Runnable() {
21         @Override
22         public void run() {
23             privMem.enter(isr);
24         }
25     };
26 }

```

Figure 3.14: An extract of the `ManagedInterruptServiceRoutine` class implementation in JOP.

number of interrupts that can be registered in JOP can be configured with a global constant. Any interrupt can be inhibited globally or by interrupt number. The `firstLevelHandler` object created at line 20 will be attached to the array of runnable objects when MISR object is registered to a mission.

SCJ requires that for every interrupt priority there is a hardware priority that can be used as the ceiling priority of ISR objects. The ceiling is then used to disable equal and lower priority interrupts during the execution of a synchronized method of an ISR instance. In this way, mutual exclusion between a MSO (that uses shared data from the ISR object) and the ISR can be guaranteed.

In a single core version of JOP, this mutual exclusion is implemented by disabling interrupts during the execution of a synchronized method and by executing the first-level interrupt handler with interrupts globally disabled. In a CMP version, there is a synchronization unit that grants access to shared objects. If the shared object is not locked by any core, then the requesting core is granted the lock; else the thread in the requesting core spin-waits until the lock is released.

Interrupt handlers are executed at hardware priorities, i.e., priorities that are higher than those of ordinary schedulable objects, and can therefore delay the completion of schedulable objects. The maximum interrupt time should be included as a blocking time in a schedulability analysis of the system. It is therefore important to bound the WCET of any ISR. An ISR can be preempted (or interrupted) only by a higher priority interrupt, assuming that the ISR disables only lower or equal priority interrupts. In JOP the timer interrupt, which is a hardware interrupt, has the lowest of the hardware priorities meaning that if low priority interrupts are not enabled during an ISR then scheduling decisions cannot be made. Nested interrupts can make the system more responsive to external events but can increase the blocking times due to interrupts. Interrupts can happen at any time, and therefore a maximum arrival rate will be required in order to include their blocking times into a schedulability analysis.

3.7 Summary

In this chapter we have presented our implementation of the SCJ profile in the Java Optimized Processor JOP. The two major issues we faced during the implementation were (1) the need to share package private information and (2) the use of the scoped memory model. Sharing private information between packages becomes problematic when such information has to be hidden from the application developer, thus changing accessibility modifiers to public is not an option. Our solution to this issue uses a singleton delegation mechanism that avoids the use of reflection. The use of the scoped memory model requires extra care as illegal references can be created within the SCJ framework itself, e.g., when registering a MEH to its corresponding mission. This is an important issue that can complicate implementations of SCJ.

The concurrency model of SCJ is implemented on top of the existing real-time thread facility provided by JOP. We used a simplified implementation of the mission sequencer where we execute its operations on the main thread of the system. This simplification is not restricted to our implementation and can also be used by different SCJ implementations but only on L0 and L1 applications.

Our implementation of SCJ's scoped memory uses a single system class, where functionality of SCJ's memory API classes defined in the `safetycritical` and `realtime` packages is delegated to this system class. The use of scoped memory requires that a set of rules should be enforced on every reference assignment. We have presented two approaches to enforce the reference assignment rules, two of them are implementation independent and use the auxiliary object information to store the scope nesting level. The third approach is specific to our JOP

implementation and uses simple hardware extensions to reduce the execution time of the reference assignment checks.

Our SCJ compliant access to memory mapped IO devices and physical memory is provided with the `RawMemory` API. We have presented two ways of implementing raw memory accessor objects: (1) using native methods and (2) using hardware objects. The use of native methods is most likely what a JVM will use while using hardware objects is specific to our implementation.⁷

First level interrupt handlers use SCJ's `ManagedInterruptServiceRoutine` class which, in contrast to using a plain method invoked by the hardware as interrupt service routine, introduces additional overhead for the interrupt service routine dispatch.

In our implementation we also provide support for L1 SCJ applications in a multi-processor version of JOP. In the multi-processor version, there is, in addition to the immortal memory, one global mission memory shared between all processors. Each processor has its own scheduler and a fixed number of MSOs assigned thereby creating a fully partitioned system. Assignment of MSOs to processors is done at mission initialization time and this assignment cannot be changed during mission execution.

⁷The hardware-near virtual machine, HVM [67], also uses hardware objects

CHAPTER 4

Scoped Memory Use: Patterns and Reusable Libraries

Correct use of the scoped memory model is perhaps the most complex feature of SCJ. Passing arguments and returning results without the use of static fields is not obvious. In Section 4.1 of this chapter we analyze the expressive power of SCJ's memory model and propose patterns for its safe use. We provide a collection of seven scoped-memory use patterns specific to SCJ that can be used for simple subroutines, sequences of subroutine calls, and nested calls. The patterns avoid memory leaks, unnecessary copying of values, and are illustrated with an implementation in the SCJ profile. Section 4.1 is based on the following published paper: *Patterns for Safety-Critical Java Memory Usage* [106].

In addition to scope-aware patterns, we explore on this chapter the topic of scope-aware Java libraries. In Section 4.2 we analyze and propose solutions to common programming patterns and idioms present in the Java class libraries (JCL) that make library classes unsuitable for SCJ. We propose changes to improve the class libraries to avoid the impact of the identified problematic patterns and illustrate these changes in Section 4.3 by implementing a total of five scope-safe classes from commonly used libraries. This chapter is based on the following published paper: *Reusable Libraries for Safety Critical Java* [105].

4.1 Use Patterns and Idioms

Explicit scoping requires care from programmers when dealing with temporary objects, passing scope-allocated objects as arguments to methods, and returning scope-allocated objects from methods.

The SCJ specification avoids having references to arbitrary memory areas to forbid MSOs from entering or executing code in another MSO's private memory. There are no public API methods to get references to arbitrary objects or to the current allocation context.¹ Traversing the scoped memory stack in SCJ (see Section 3.5) is done with the following static methods:

- `enterPrivateMemory(long size, Runnable logic)` to move into an inner nested private scope. This method creates a new nested private memory of size `size` (in the scope of the caller) or enters an already existent nested private memory, and executes the runnable `logic`. It can only be called from a private memory and will always add a new level into the scope stack.
- `executeInOuterAreaOf(Object obj, Runnable logic)` to execute the `logic` runnable in the memory area where the object passed as argument is allocated. This method is used to move into an arbitrary outer level in the scope stack.
- `executeInOuterArea(Runnable logic)` to execute the runnable `logic` in the immediately outer nested scoped memory in the scope stack.

Moving data between scopes requires a creative way of using these SCJ memory API methods. In the remaining of this section we present several scoped memory usage patterns that aim at helping in the development of SCJ applications. The focus is on how to pass arguments into and return objects from methods that change the allocation context to a nested memory.

4.1.1 The Basic Pattern

One can consider the use of an initial private memory executing the MSO code to be the most basic use case of the SCJ memory model. In this basic pattern

¹Without the possibility to obtain a reference to a memory area, the `newInstance()` and `newArray()` methods defined in the `javax.realtime.AllocationContext` interface cannot be used to create objects in explicit memory areas.

the event handler will not return results or need arguments as parameters (e.g. no feedback is expected when sending control signals to actuators). In this case, it is not necessary to preserve results for the next time the memory area is activated.

Implementation of this pattern is straightforward: a managed event handler overrides the `handleAsyncEvent()` method. Temporary objects are allocated in the initial private memory and the memory area is recycled at the end of the release. This pattern is called *Scoped Run Loop Pattern* in [94] but the difference between the SCJ pattern and the RTSJ pattern is that the memory area does not need to be explicitly created and entered, as this is all handled by the SCJ infrastructure.

4.1.2 Loop Pattern

Memory in embedded applications is usually a scarce resource. Moreover, SCJ private memories cannot be resized after their creation. Therefore, applications should carefully use the available scoped-memory space. One way to recycle memory is to use nested private scopes. A nested private scope can be entered and exited several times during a release. The object representing the nested private scope is reused in the implementation to avoid generating garbage in the initial private memory [114]. This nested private memory is entered by creating a `Runnable` object and change to the allocation context is done with `enterPrivateMemory()`.

An example scenario that benefits from temporary allocation within a nested private memory is a loop body that has no dependency on variables declared within the method and produces no results that need to be visible following execution of the loop body. This is shown in Figures 4.1a and 4.1b.

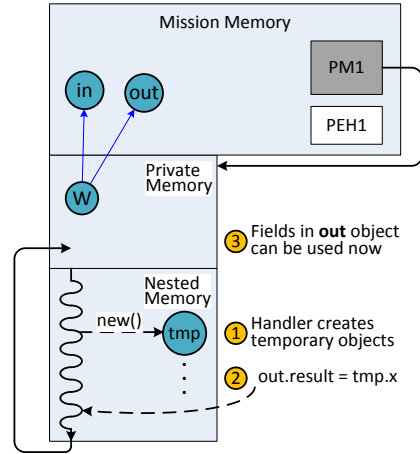
In this example, all the objects created by the `Runner` object while executing its `run()` method in the nested memory will be collected when the `enterPrivateMemory()` method returns. One can imagine the instance of the `Runner` class to be applying an algorithm (e.g. encryption, FFT, etc.) to a block of static data that should be transmitted periodically. The algorithm can then generate garbage as a result of temporal computations which will be collected at the end of every release.

Two important issues need to be considered. First, avoidance of illegal references and second, the possibility of introducing memory leaks as a result of allocating objects in an outer memory (e.g. immortal or mission memory) on every iteration of the loop. Furthermore, it is important to emphasize that this additional


```

1 public class Worker implements Runnable {
2
3   AuxIn in; AuxOut out;
4
5   public Runner(AuxIn in, AuxOut out) {
6     this.in = in;
7     this.out = out;
8   }
9
10  public void run() {
11    // Use this.in, work and generate garbage
12    T tmp = new T();
13
14    // assign primitive values to a field in
15    // the auxiliary AuxOut object
16    out.result = tmp.x;
17  }
18 }
19
20 class MyHandler extends PeriodicEventHandler {
21
22   public void handleAsyncEvent() {
23     // allocate input and output parameters
24     // fill input arguments
25     Runnable w = new Worker(in, out);
26     ManagedMemory.enterPrivateMemory(256, w);
27     // now we can use out.result
28   }
29 }

```



(a) Code sample to pass and return primitive values into a scope.

(b) Graphical representation of the execute with primitive return pattern.

Figure 4.2: Execute with primitive return value.

auxiliary objects are passed in the constructor of that class. Upon return, the caller copies the value from the Runnable's field. A drawback when using this approach is that only primitive values can be copied as the return value. Figures 4.2a and 4.2b show an example.

4.1.4 Returning a Newly Allocated Object

It might be the case that while executing a handler in a nested private memory, we need to create objects that have to be used later. References to objects created in inner scopes cannot be passed to outer contexts, as they will be reclaimed

```

1  class Runner implements Runnable {
2      RetObject rObj;
3
4      public void run() {
5
6          // Do some work
7          ...
8          ManagedMemory.executeInOuterArea(new Runnable() {
9              @Override
10             public void run() {rObj = new RetObject();}
11         });
12     }
13 };
14
15 class MyHandler extends PeriodicEventHandler {
16
17     public void handleAsyncEvent(){
18
19         Runner r = new Runner();
20         ManagedMemory.enterPrivateMemory(256,r);
21
22         // Use returned object and fields
23         r.rObj ...
24     }
25 }

```

Figure 4.3: Example of the use of `executeInOuterArea(Runnable logic)` to create objects in outer nested scopes.

when leaving the inner scope. The key point here is that objects that will be used after the inner scope is left, need to be created in an outer scope. SCJ's API offers two methods to move into outer levels in the scope stack and create the required object directly after a change of allocation context. Figure 4.3 shows an example of how to use nested memory scopes with return objects together with `executeInOuterArea(Runnable logic)`. The return object is allocated in the immediately outer scope of the caller and can be accessed through the reference that is part of the `Runner` instance. Note that any object created in the context of the `run()` method of line 10 will be created in an outer nested scope. Therefore, temporary objects should not be created inside this method as opposed to the `run()` method of line 12 in Figure 4.2a where temporary objects are allowed.

4.1.5 Scoped Methods

One can create an abstraction to hide the complexities of parameter passing, returning results and switching between memory areas by combining the patterns described above. Ideally, a scoped method will be an expression of the form `ret = f(params)`, can be executed in a specific memory area, has input parameters, and can return values or references to objects.

For an SCJ application developer, executing code in a specific memory area is achieved through the use of `enterPrivateMemory()` to go into a nested private memory or by using any of the two versions of `executeInOuterArea*` to move into an outer nested scope. Since both methods take a `Runnable` object as argument, the active part of a scoped method should be coded within the `run()` method of a `Runnable` object.

In Figure 4.4 we show an example of a scoped method that executes in a nested private memory. To provide the functionality of parameter passing and returning results, we use a parameter object (lines 1 – 7) whose fields contain the method arguments, a field to store the reference of a new object so it can be accessed when the method returns, and an arbitrary object to provide the memory area where the returning object shall be allocated.

4.1.6 Runnable Factory

In a large application, creating all the code as in Figure 4.4 for many scoped methods might be a cumbersome task. To hide such complexity, one can use a *runnable factory* whose methods have a `Runnable` return type. The required code by a particular method is implemented in the `run` method of this returned `Runnable`. The factory object itself can be instantiated in the handler's private memory or any other shared memory. Parameters can be passed when calling the factory method. Figure 4.5 illustrates the concept with an example.

The example shows an *auxiliary object*, `auxObjIn`, used to pass arguments and return values by means of the `readTemperature()` factory method. This object has a field used to store a reference to an *arbitrary object*, `resArbObj`, that should be returned. In this case, the memory area where the result object is to be saved, is the memory area where the auxiliary object was allocated. The drawback with this particular implementation is that the memory area where the returned object is allocated will be restricted to the memory area of the auxiliary object.

```

1  public class ParamObject {
2
3      int arg0;           // Method parameters
4      ReturnObject retObject; // Reference to returned object
5      Object destScope;   // Returned object is allocated in the scope of the destScope object
6
7  }
8
9  public class Method implements Runnable {
10     ParamObject params;
11
12     Method(ParamObject params){ this.params = params;}
13
14     public void run() {
15
16         // Use parameters, do work, create garbage
17         ...
18         // Change context, create return object
19         ManagedMemory.executeInOuterAreaOf(params.destScope, new Runnable() {
20             public void run() {
21                 ReturnObject rObject = new ReturnObject();
22
23                 // Update return object fields
24                 params.retObject = rObject;
25             }
26         });
27     }
28 }
29
30 class MyHandler extends PeriodicEventHandler {
31
32     Object mem = new Object(); // Allocated in mission memory when PEH is created
33
34     public void handleAsyncEvent(){
35
36         // Created in the scope where handler executes
37         ParamObject pObj = new ParamObject();
38
39         // Assign fields to the pObj
40         pObj.destScope = mem;
41         ...
42
43         // This object simulates the method with parameters that returns an object
44         Method myMethod = new Method(pObj);
45         ManagedMemory.enterPrivateMemory(256, myMethod);
46
47         // Now the returned object can be used
48         ...
49     }
50 }

```

Figure 4.4: Scoped method with parameters and a return object.

```

1  public class RunnableFactory implements IRunnable{
2
3      @Override
4      public Runnable readTemperature(final int i , final AuxObj auxObjIn) {
5
6          // This runnable will be allocated in the context of the caller , care should
7          // be taken when the factory is in any of the shared memory areas
8          return new Runnable() {
9
10             @Override
11             public void run() {
12                 // Do work, here we can use input parameters
13                 ...
14                 // The log() method is an example of a method used by all the methods
15                 // in the factory (a shared functionality in the application)
16                 log ();
17
18                 // Change execution context needs another runnable
19                 ManagedMemory.executeInOuterAreaOf(auxObjIn, new Runnable(){
20
21                     @Override
22                     public void run() {
23                         ArbObj resArbObj = new ArbObj();
24                         resArbObj.a = 50;
25                         auxObjIn.arbObj = resArbObj;
26                     }
27                 });
28             }
29         }
30     }
31
32     @Override
33     public Runnable otherFactoryMethod() {
34         ...
35     }
36 }
37
38 class MyHandler extends PeriodicEventHandler {
39
40     public void handleAsyncEvent() {
41
42         RunnableFactory factory = new RunnableFactory();
43         AuxObj auxObj = new AuxObj();
44
45         ManagedMemory.enterPrivateMemory(256, factory.readTemperature(5, auxObj));
46
47         ManagedMemory.enterPrivateMemory(512, factory.otherFactoryMethod());
48     }
49 }

```

Figure 4.5: Runnable factory.

4.1.7 Producer/Consumer

Control systems are often composed of producer and consumer processes that run in their own thread of control. The exchange of information between a producer and a consumer can involve data structures or objects, rather than primitive types. In [94] and in [34] solutions are proposed for RTSJ. Such solutions involve the use of portal objects (not part of SCJ), shared scopes, or the introduction of *safe* violations to the reference assignment rules.

Communication between handlers goes through objects located in the shared memory areas, mission and/or immortal memory. Objects in those areas are not reclaimed until termination of the mission or the JVM respectively. Thus we need to reuse objects in those shared memory areas. In [44], a solution is devised using a memory pool of immortal objects. An alternative is the use of a pool of objects in mission memory. In this way the pool of objects will be collected when the mission finishes.

4.2 Reusable Libraries: Issues and Solutions

The standard Java class library (JCL) was not developed to be used in the SCJ memory model. The JCL is based on a system where objects are allocated on the heap and are automatically de-allocated by a GC. Furthermore, objects can refer to each other unrestrictedly. These two assumptions are no longer valid in SCJ as its memory model eliminates the heap and the GC. In addition, as objects may have different lifetimes, scope allocations restrict how objects can refer to each other. As a result, the different programming idioms and patterns used in the JCL can lead to memory leaks and illegal reference assignments.

In Sections 4.2.1 to 4.2.6 we present a more detailed study of the programming patterns and idioms present in three of the most commonly used JCLs: `java.io`, `java.lang` and `java.util`. We also present possible solutions for its safe use within SCJ's scoped memory model. The outlined solutions are then used for our own implementation of a total of five scope-safe representative classes of the mentioned packages.

4.2.1 Lazy Initialization

This pattern delays the initialization of a field that contains a reference to an object until the object pointed by that field is used for the first time. This

pattern is used for two purposes: (1) to save memory in case the object is never needed, and (2) to break circularities in class initialization [21]. The problem with this pattern is that the object referred to by the lazily initialized field will be created in the scope of the first handler that uses it. This scope and the scope where the object with the lazy initialized field was allocated may not be the same.

As an example, consider the `keySet()` and `values()` methods in the `AbstractMap` class of the `java.util` package. These methods provide different views of the objects contained in a particular map implementation such as `HashMap` or `TreeMap`; they return a `Set` and a `Collection` object respectively. To ensure referential integrity, a call to these methods should be done from a scope that encloses the scope of an `AbstractMap` subclass instance.

One can also view the singleton pattern as a different version of lazy initialization. The creation of singleton objects is prone to breaking referential integrity in SCJ. The singleton pattern under the scoped memory model of RTSJ has been analyzed in [34]. The solution, which can also be applied to SCJ, consists of explicitly allocating the singleton instance in immortal memory by using the available immortal memory API methods, e.g. with the `newInstance()` method.

Illegal references can be avoided by creating the lazy object either in immortal memory or in the same scope as the object containing the lazy initialized field. One possible solution is to execute the object creation code in class initializers, which will execute in immortal memory, as it is done in [17]. This approach works for objects that should be accessed during the whole VM lifetime, such as a `Properties` object. For objects that are only used by specific missions, another approach is to create the lazy initialized object when the instance of the class containing the field is created, i.e., as part of the object's constructor. Of course this does not mean that the lazy initialized object will be used but we are on the safe side of referential integrity. This is the approach followed in the implementation of our libraries.

Another solution can be to change the allocation context to the memory area where the object with the lazy initialized field is allocated. The lazy object can then be safely created.

4.2.2 Dynamic Resizing

When a structure grows beyond its current capacity, it needs a size adjustment to accommodate new elements. Resizing involves creation of a new and larger array to accommodate the previous elements and the new ones. The old array

is de-referenced leading to a memory leak. The new array, created in the scope of the caller, may eventually be referenced from an object in a different scope, thereby potentially creating an illegal reference assignment. This situation is illustrated in Figure 4.6a, where a method adds an object to a full collection. The objects in the figure are annotated with the scope where they are allocated: MM stands for the shared mission memory and PM for a private memory.

This method is called from a private memory, PM, while the object to be added lives in mission memory, MM. In Figure 4.6a, the container array ends in a scope that will be inaccessible to other handlers, even though it is perfectly legal for all handlers to access the elements of the array. This is referred to as polluted containers in [39].

An example of this situation is found in the `Vector` class. When an element is added, the `ensureCapacity` method is used to resize the collection if necessary. In case there is no resizing, the element to be added must be allocated in the same scope or in an outer nested scope from where the `Vector` object is allocated.

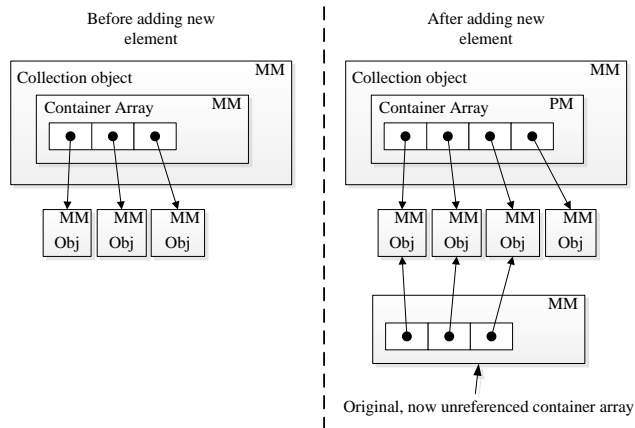
A similar situation occurs with some of the methods in the `StringBuffer` and `StringBuilder` classes. Concatenation and append operations may need resizing while character replacement operations will always create new character arrays.

The fundamental problem here is that for every expansion of a data structure, a new storage element (usually an array of objects) is created in the context of the caller while the previous storage element gets dereferenced, producing a memory leak.

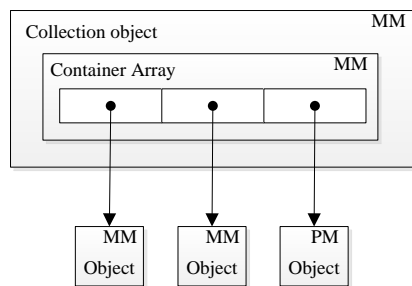
One option to avoid structures to dynamically resize is to limit the maximum amount of elements the structure can hold. This seems too restrictive, but it is likely that most data structures for hard real-time systems are big enough to hold all of the intended elements.

Another option is to change the allocation context before the expansion to guarantee that the storage element is created in the same memory area as the data structure. However, memory leaks created by de-referencing the old storage element cannot be avoided.

The approach we use in our libraries is to limit the maximum amount of elements a structure can hold and to recycle objects from a pool of objects. Objects are allocated from the pool when they are needed and returned to the pool when removed from the data structure. In this way we avoid the risk of creating the storage element in another region and the memory leaks associated with



(a) Polluted container



(b) Polluted object

Figure 4.6: Effects of using shared objects from different scopes. Scoped memories from where objects are allocated are represented as MM for mission memory and PM for private memory

removing or replacing elements. The drawback is the overhead introduced by the additional functionality needed to do the pool object management, i.e., get an object from a pool and to reset its state when removing the object from the structure.

4.2.3 Objects Used in Mixed Contexts

Modification of JCL objects shared between handlers requires special care because most of the methods may create new objects in the scope of the caller. Consider for instance the following two examples:

- The `addElement(Object obj)` method of the `Vector` class. The already existing object to be added to the collection should reside in the same or in an outer nested scope as the `Vector` object. As long as the addition of the new element does not exceed the capacity of the collection, the method can be called from within any scope.
- The `add(E e)` method of the `LinkedList` class. This method will create a wrapper object. This wrapper object is used to store book keeping information (e.g. references to the next and previous elements in the list) and a reference to the actual element to be added. There are two requirements in this case: 1) call the method from the same memory, or an outer nested scoped memory, as the one in which the `LinkedList` object is allocated, and 2) to ensure the element is added in the same scope or an outer nested scope from where the method is called. Neglecting to do this may result in contamination of the container array with an object allocated in a scope that is inaccessible to other handlers. This situation is referred to as polluted objects in [39] and is illustrated in Figure 4.6b.

This is perhaps one of the most difficult issues to address in library code since there is no easy way to ensure that caller-allocated results or arguments to methods reside in appropriate scopes.

Illegal references come from field or array stores, either to new objects or to objects referenced by arguments passed to methods. Arguments must reside in the same scope as the `this` argument,² or in an outer scope (e.g. for setter methods). One option for ensuring that library code enforces this requirement is to provide dynamic guards (see the memory annotations appendix of [77]). A dynamic guard is a conditional statement used to test that the parenting

²`this` is a reference to the object whose method is being called.

relationship of the scopes in which arguments reside is appropriate to avoid illegal references.

Another option is to change the allocation context to execute a method (or part of it) in the scope of the `this` argument. This is particularly useful if the method requires the creation of auxiliary objects as in a `HashMap` or a `LinkedList`, where additional objects are created to store bookkeeping information. For our libraries we reuse objects from pools and require that instances of library classes are created in the same or an inner nested scope as the pool.

4.2.4 Iterators

It is common to use an iterator pattern in collection classes to traverse and access elements of a container. This pattern is used in base classes such as `AbstractMap` and `AbstractList`. Iterator patterns require the allocation of `Iterator` objects that may lead to memory leaks if used within the shared memory areas (e.g. during initialization phase) or if used repeatedly. Iterators also require additional synchronization considerations by the application developer.

One option for the use of this pattern is to pre-allocate single iterator objects when collection objects are created, as in [52]. However, we consider that not much is gained by pre-allocating a single iterator object per collection, because this solution only works on single-threaded applications. As described by the authors of [52], if a handler requests use of the iterator object after another handler has gained access to the single iterator object, then the iterator object's state is reset causing runtime errors.

The use of iterators may become problematic if used multiple times. The reason is that to use the iterator pattern we need to obtain an `Iterator` object.³ In this case, a better idea is to use iterators within nested private memory areas in the same way as with the loop pattern presented in Section 4.1.2.

4.2.5 Loop Bounds

Although not related to scope-safety, the methods in software libraries intended for real-time systems must have predictable execution times. Unbounded loops are a concern for real-time systems as worst-case execution time (WCET) anal-

³One can also reuse iterator objects however obtaining a new `Iterator` object seems to be the common practice.

ysis tools cannot automatically extract loop bounds. For correct analysis, loops must be annotated manually with bounds.

Most of the programming idioms, used for loops in the JCL, are not friendly for our current WCET analysis tool, WCA [116]. The exit condition in loops may depend on boolean flags or on a value that the data flow analysis in the tool is not able to propagate (e.g. internal manipulation of an array size). The most common case were loops in which the stop condition is a boolean check for a **null** element. In our libraries, loops are limited to a maximum number specified as an argument in the instance constructor. This argument can be propagated by the data flow analysis of the WCA tool provided it is not modified inside the library code. To enforce this restriction, such arguments are declared **final**.

4.2.6 Exceptions

Throwing exceptions may involve the creation of an exception object in the current allocation context. Even if the scope has been sized to include the effects of any thrown exception (that is, not only considering the normal execution path), there is still the risk of ending up with illegal assignments if the exception is thrown in an inner scope and propagated to be handled in an outer scope. Furthermore, it is desirable to avoid exception propagation as this may introduce program paths that are complicated to analyze [77].

Safe exception handling in SCJ requires the observation of the following rule:

- Propagation of exceptions to a scope different from the one in which it was originally allocated causes a **ThrowBoundaryError** exception. In this way, scoped memory errors such as illegal reference assignments are avoided [77].

According to SCJ's specification, there are no special requirements for the allocation of exception objects. These objects can be created in the current scope with the **new** keyword; in a different scope after a change of allocation context; or they can be pre-allocated. Our libraries pre-allocate exceptions in immortal memory and when necessary and possible, library exceptions are created with a constant string message describing the cause of the error. Unnecessary memory allocations that come from concatenation of strings are thus eliminated.

4.3 Reusable Libraries: Implementation

Real-time and safety-critical systems are typically less dynamic and more restricted than non real-time or non safety-critical systems. Such characteristics allow for certain simplifications and modifications to be made in Java's library code in order to achieve the characteristics listed below, which are required for SCJ applications:

- **Maintain referential integrity.** Referential integrity concerns the avoidance of throwing illegal assignment exceptions. It is important that SCJ library classes are aware of the scoped memory where method arguments, returned results, and objects allocated in methods reside.
- **Predictable memory consumption.** The size of a scoped memory area has to be provided when the area is created. It is therefore important to know how much memory will be allocated in the specific scoped memory. Libraries with predictable memory consumption help to size scoped memory areas in such a way that allocation demands can be met at all times during the execution of a program.
- **Predictable worst-case execution time.** Predictable execution time in library code is essential for calculating the WCET of an application. In turn, WCET values are used as input for the schedulability analysis.

The SCJ specification provides a list of class libraries for safety-critical applications defined with respect to the JDK 1.6. This set of core libraries is kept as small as possible by restricting the use of certain methods and fields. The goal is to reduce size and complexity to decrease the certification effort of safety-critical applications. A summary of these classes is presented in Table 4.1.

The first part of Table 4.1 (rows 1 to 11), lists classes from the `java.io` package and the second part (rows 12–37), from the `java.lang` package. This subset of classes is our starting point either to modify or to create new safe classes, such as the classes for the `java.util` package where only the `Iterator` interface is provided in the SCJ specification. In the relation to JDK 1.6 column, *Same* means that the definition of the corresponding class in SCJ is the same as in JDK 1.6. *Restricted* means that the corresponding class in SCJ is allowed to use only a reduced set of the methods and fields of JDK's 1.6 definition. The *reusability type* column refers to whether or not event handlers can safely share instances of this unmodified class. The next section provides more details.

Table 4.1: List of library classes allowed by SCJ. Exception classes are not shown.

No.	Class Name	Relation to JDK 1.6	Reusability type
1	Closeable	Same	—
2	DataInput	Same	—
3	DataOutput	Same	—
4	Flushable	Same	—
5	Serializable	Same	—
6	DataInputStream	Same	Instance unsafe
7	DataOutputStream	Same	Instance unsafe
8	FilterOutputStream	Same	Instance safe
9	InputStream	Same	Instance safe
10	OutputStream	Same	Instance safe
11	PrintStream	Same	Instance unsafe
12	Appendable	Same	—
13	CharSequence	Same	—
14	Comparable	Same	—
15	Runnable	Same	—
16	Boolean	Same	Instance safe
17	Byte	Same	Instance safe
18	Character	Restricted	Instance safe
19	Class	Restricted	Instance unsafe
20	Double	Same	Instance safe
21	Enum	Restricted	Instance safe
22	Float	Same	Instance safe
23	Integer	Same	Instance safe
24	Long	Same	Instance safe
25	Math	Same	Instance unsafe
26	Number	Same	Instance safe
27	Object	Restricted	Instance safe
28	Short	Same	Instance safe
29	StackTraceElement	Same	Instance safe
30	StrictMath	Same	Instance unsafe
31	String	Restricted	Instance safe
32	StringBuilder	Restricted	Instance unsafe
33	System	Restricted	Instance unsafe
34	Thread	Restricted	Instance unsafe
35	Thread.Uncaught- ExceptionHandler	Same definition	—
36	Throwable	Restricted	Instance unsafe
37	Void	Same	Instance safe

4.3.1 Analysis of Standard Java Class Libraries

As a first step towards our SCJ libraries, an analysis of the classes defined in JSR-302 (Table 4.1), using OpenJDK's (version 6) source code, was performed to:

- Classify a standard implementation of the classes allowed by JSR-302 according to the taxonomy described in [41]. This classification is performed to provide an estimate of the degree of reusability of unmodified classes.
- Locate the points where memory allocations take place. In order to provide bounds on memory consumption it is important to know how memory is being used and, whenever possible, to provide rules, restrictions, or modifications that prevent unbounded memory allocations.

In [41], classes are cataloged according to whether or not they can be considered as no-heap safe. No-heap safe means that a particular class can be used concurrently by both *heap* and *no-heap* threads without the risk of storing references to heap-allocated objects.

To adapt this classification to SCJ, we abandon the concept of *heap* threads, as SCJ does not allow the use of heap memory. In addition, what is referred to as code executed by *no-heap* threads in [41] translates into periodic or aperiodic event handlers. Our classification is for scope safety (i.e. without the risk of generating illegal references) and has the following two categories:

- **Instance safe:** A class instance can be shared by different event handlers or allocated in a handler's private memory and used in a nested private memory without the risk of generating illegal references. Few classes are expected to fall into this category, as they need to have only final reference fields.
- **Instance unsafe:** Event handlers cannot safely share instances of this class.

The results of this classification are shown in Table 4.1. The following rules were used for classification:

1. A class is instance safe if all of its reference fields are declared as final.

2. A class with non-final reference fields assigned only at class initialization (execution of its `<clinit>` method) can be considered to be instance safe.
3. A class with non-final reference fields or with methods that perform array reference assignments are instance unsafe.
4. A class inherits its superclass classification. For example, if class B extends class A and class A is instance unsafe, then class B is also instance unsafe. An unsafe class B can, however, have a safe class A as parent.

During the analysis of the JCL classes, we noted all allocation places and reference assignments. Once a problematic part in the code was located, we proceeded to implement our solutions, which combine techniques such as restricting the size of different structures, changing between scopes, running specific code in nested scopes, and recycling of objects by memory pooling.

In the following sections, we describe the implementation of five representative classes of the standard Java libraries. The implemented classes were adapted for use in safety-critical Java in accordance with the requirements of having referential integrity, predictable memory consumption, and predictable worst-case execution time; and the solutions outlined in Sections 4.2.1 to 4.2.6. Of the five classes, three are defined in the safety-critical Java specification (**AbstractStringBuilder**, **StringBuilder**, and **DataInputStream**) while the other two (**Vector** and **HashMap**) are not. Nevertheless, we consider them to be important for the development of reusable software components.

4.3.2 AbstractStringBuilder and StringBuilder

Within these two classes, memory consumption is related to the size of the character array backing those types of objects. To provide bounds on memory consumption, we limit the maximum number of characters any of those classes can hold. That is, the size of the character array is limited to the initial size set at object creation. This decision can be supported by considering that safety-critical programs typically do not incur in extensive text processing or file manipulations [77].

Limiting the maximum number of characters also has the following two benefits: 1) Resizing operations are not needed and 2) we can have bounds on methods that iterate over the character elements (through annotations, see Section 4.2.5). For methods such as `append(String str)` creating a new character array will be necessary if the resulting string exceeds the initial size. However, as resizing operations are not allowed, an exception is thrown.

4.3.3 `DataInputStream`

The `java.io` package contains classes to perform input and output operations in Java. We focus on the `DataInputStream` class because the additional classes in this package defined in JSR-302 (see Table 4.1) are only wrapper classes.

Memory allocations within classes in this package come from re-sizable arrays that are used for temporary processing or to perform buffered reads and writes. Methods that perform temporary processing can be executed inside nested scopes. In this way, array resizing is allowed if needed and any temporary array will be collected when leaving the nested scope. As an example, Figure 4.7 shows our modified version of the `readUTF(DataInput in)` method (lines 4–11) from the `DataInputStream` class. This method reads a representation of a character string encoded in modified UTF-8 format⁴ and uses two arrays of up to 65,535 bytes for temporary processing. The `readUtfHelper` inner class encapsulates in its `run()` method (lines 22–25) the code of the original `readUTF` method and executes it in a nested private memory. The modified version needs an additional parameter to set the size of the nested scope because the memory consumption of objects is implementation dependent. The `run()` method also handles the additional scope change needed to return the resulting string object into the context of the caller (lines 29–34). The scope change is made using the SCJ's `executeInOuterArea` method, which moves the current allocation context one level up in the scope stack.

Resizing operations can also be avoided by using working arrays and buffers of size equal in size to the worst-case expected length. The drawback of this approach is that arrays that are only needed for a few methods will be created for every instance and will most likely be poorly utilized.

4.3.4 `Vector` and `HashMap`

These two classes are representative for the `java.util` package. The safety-critical Java specification only provides the definition for the `Iterator` interface. However, due to how useful this package is with regard to reusable software components, we decided to provide some implementation examples for this package.

The modified `Vector` class, illustrated in Figure 4.8, shows how, through the use of object pooling [64], a solution for most of the problems mentioned in Sections 4.2.1 to 4.2.6 can be provided.

⁴See <http://docs.oracle.com/javase/6/docs/api/java/io/DataInput.html#modified-utf-8> for a description of the modified UTF-8 format

```

1 public class DataInputStream ... {
2     /* Other methods of DataInputStream class */
3     ...
4     public static final String readUTF(DataInput in, long size) ... {
5         ReadUtfHelper readUtfHelper = new ReadUtfHelper();
6         readUtfHelper.in = in;
7         ManagedMemory.enterPrivateMemory(size, readUtfHelper);
8
9         /* Return String lives in the context of the caller */
10        return readUtfHelper.retString;
11    }
12 }
13
14 class ReadUtfHelper implements Runnable {
15     String retString;
16     DataInput in;
17
18     @Override
19     public void run() {
20         try {
21             /* Begin of original code of readUTF method */
22             int utflen = in.readUnsignedShort();
23             byte[] byterr = new byte[utflen];
24             final char[] chararr = new char[utflen];
25             ...
26             /* End of original code of readUTF method */
27
28             /* Return a String object in the scope of the caller */
29             ManagedMemory.executeInOuterArea(new Runnable() {
30                 @Override
31                 public void run() {
32                     /* count is the length of the char array */
33                     retString = new String(chararr, 0, count);
34                 }
35             });
36         } catch (IOException e) { ... }
37     }
38 }

```

Figure 4.7: Example of a method modified to run in a nested private scope.

Our classes are restricted to store only elements belonging to a pool of pre-allocated objects. When elements are removed or replaced, they are returned to their corresponding pool, their state is reset, and they are marked as available for reuse. To add an element, one must first obtain a free object from the pool of pre-allocated objects and then add it to the **Vector**.

An **ObjectPool** instance is created with a fixed number of objects. The number of objects is passed as a parameter in the constructor (if omitted, a default value is used). The elements belonging to the pool are created when the **ObjectPool**

is instantiated and a `PoolObjectFactory` provides a strategy to define how they will be created (through the `createObject()` method). Retrieving a free object from the pool is done by calling the `getPoolObject()` method which in turn will call an initialization hook, the `initiaize()` pool object's method, from the returned free object. When returning an object to the pool, the `releasePoolObject()` method calls the termination hook method, `reset()`, of the object being returned. It is important to note that the `ObjectPool` as well as the `Vector` can only have elements of the `PoolObject` type or subtype. This restriction is enforced through generics. Method `getPool()` returns a reference to the pool an object belongs to (it is possible that a `Vector` contains elements from different pools).

To bound the use of memory, we note that memory allocations in the `Vector` class result from resizing its internal storage element (an array of objects), throwing new exceptions, and from the use of iterators (create a new `Iterator` object). Resizing is avoided by fixing the size of the internal storage element. The fixed-sized storage element together with the fixed-size pool of objects implies that the maximum number of elements that the collection will store must be known in advance. This value will be application-specific and can be calculated according to the application requirements through static analysis. Creating new exception objects is avoided by pre-allocating them in immortal memory during class initialization.

For the modified version of the `HashMap` class, we also used pools of objects. However, with the `HashMap` class we need a double object-pool management, one for the objects representing entries in the bucket list (implementations of `Map.Entry`) and one for the `Map` objects that are to be added to the hash map.

4.3.5 Comparison with JCL

Tables 4.2 and 4.3 show a comparison between our modified and the original classes of the JDK 6 implementation. Tables 4.2 compares the lines of code (LoC), number of fields (NoF), number of methods (NoM), and number of constructors (NoC). In each category the numbers to the left of the "/" symbol correspond to the original JDK implementation while the number to the right correspond to our implementation. The numbers in parenthesis represent the number of methods belonging to the public API of the class. Table 4.3 shows the number of modified methods and the number of additional methods. The number in parenthesis indicates how many of the modified methods belong to the public API.

None of the original interfaces implemented by the modified classes were changed.

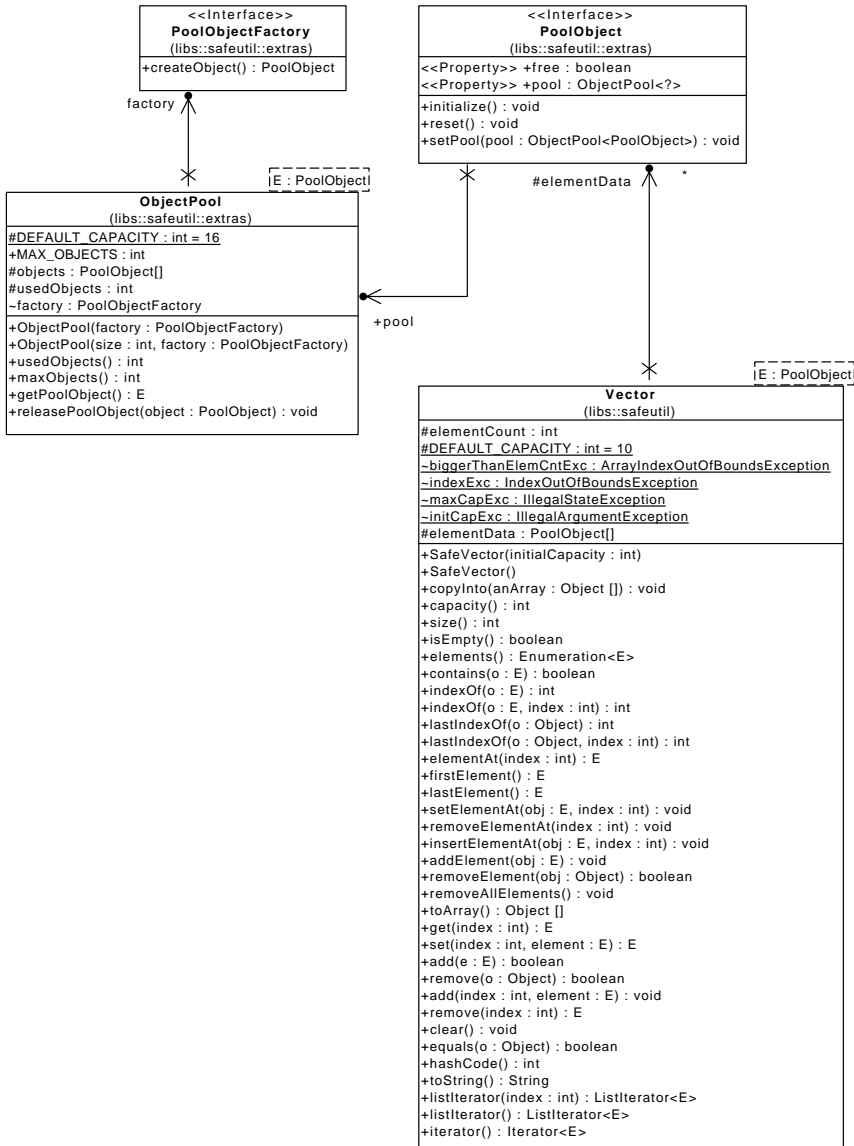


Figure 4.8: Class hierarchy of the modified Vector class.

One of the goals is to keep compatibility as close as possible to standard Java applications. However, a number of interfaces implemented by the original JDK 6 classes are not allowed by SCJ. For example, the `Cloneable` interface is not allowed because of its weak definition. Therefore, the `clone()` method will throw a `CloneNotSupportedException`. The `Serializable` interface is left as part of the implemented classes for compatibility with standard Java, but its inclusion or exclusion has no effects on a SCJ application.

Some of the constructors had to be eliminated due to their underlying algorithm. For example, the `Vector(Collection<? extends E> c)` is omitted because the size of the final array element created cannot be guaranteed. This is because the constructor relies on the size of the `Collection` parameter, which can be modified while the constructor is still executing. As a result of modifying the collection, array objects of varying size can be created and it is not known which one will be returned to be used as the storage element for the `Vector`. From this it follows that methods operating on collections are also eliminated. Constructors that require parameters for resizing are also omitted, as well as methods for resizing and ensuring capacity. A similar restriction regarding constructors and resizing methods applies for `HashMap`.

The reduction in the number of methods and lines of code is a consequence of two things: (1) the elimination of methods that are no longer needed (e.g. resizing methods) and (2) the reduced number of methods allowed by the SCJ profile. The increase in the number of fields in the implemented classes is a consequence of the pre-allocated exceptions included as class variables.

Additional methods in the modified classes are the result of needing specific SCJ functionality, such as a scope change in `DataStream`, or because exceptions that may be thrown by the class are preallocated in immortal memory and therefore introduce a static initializer method.

4.3.6 Testing

To check that the implemented classes are functionally correct, a set of test cases were developed using the standard JUnit Java framework. The test cases allow us to check the majority of methods, except for those that involve parts of the SCJ API. For methods including parts of the SCJ API, the Java processor JOP [113] our SCJ implementation was used.

To test that there are no reference assignment errors, we used the private memory analyzer tool described in [37] together with the reference assignment check facility of JOP. Memory consumption is checked by measuring the amount of

Table 4.2: Comparison of the implemented classes with JDK's implementation. In each category the numbers to the left of the "/" symbol correspond to the original JDK implementation. LoC = Lines of code, NoF = Number of fields, NoM = Number of methods. Number of public methods is in parenthesis.

Class Name	LoC	NoF	NoM	NoC ¹
AbstractStringBuilder	447 / 237	2 / 6	52 (50) / 29 (26)	2 / 2
StringBuilder	189 / 119	1 / 1	38 (35) / 19 (18)	4 / 4
DataInputStream	212 / 108	4 / 5	18 (18) / 17 (17)	1 / 1
DataInputStream\$ReadUtfHelper ²	– / 77	– / 2	– / 1	– / 1
DataInputStream\$1 ³	– / 6	– / 0	– / 1	– / 0
Vector	322 / 228	4 / 7	48 (45) / 36 (35)	4 / 2
Vector\$1 ⁴	14 / 14	1 / 1	2 / 2	0 / 0
Vector\$Itr	33 / 33	3 / 3	4 / 4	0 / 0
Vector\$ListItr	43 / 43	0 / 0	6 / 6	1 / 1
HashMap	356 / 269	10 / 22	36 (13) / 24 (11)	4 / 2
HashMap\$Entry	48 / 57	4 / 5	8 / 9	1 / 1
HashMap\$EntryIterator	5 / 5	0 / 0	1 / 1	0 / 0
HashMap\$EntrySet	21 / 24	0 / 0	5 / 5	0 / 0
HashMap\$HashIterator	41 / 43	4 / 4	3 / 3	1 / 1
HashMap\$KeyIterator	5 / 5	0 / 0	1 / 1	0 / 0
HashMap\$KeySet	17 / 27	0 / 0	5 / 5	0 / 0
HashMap\$ValueIterator	5 / 5	0 / 0	1 / 1	0 / 0
HashMap\$Values	14 / 14	0 / 0	4 / 4	0 / 0

¹ Zero means only the default implicit constructor.

² Not in JDK6, encapsulates SCJ functionality.

³ Not in JDK6. Anonymous Runnable class.

⁴ Anonymous Enumeration class.

Table 4.3: Number of modified methods and additional methods in the modified classes. The number in parenthesis indicates how many of the modified methods belong to the public API.

Class Name	Modified methods	Additional methods
AbstractStringBuilder	11 (11)	1
StringBuilder	1 (0)	0
DataInputStream	1 (1)	1
DataInputStream\$ReadUtfHelper	0	1
DataInputStream\$1	0	1
Vector	16 (15)	2
Vector\$1	0	0
Vector\$Itr	0	0
Vector\$ListItr	0	0
HashMap	11 (5)	1
HashMap\$Entry	0	1
HashMap\$EntryIterator	0	0
HashMap\$EntrySet	1	0
HashMap\$HashIterator	1	0
HashMap\$KeyIterator	0	0
HashMap\$KeySet	1	0
HashMap\$ValueIterator	0	0
HashMap\$Values	0	0

memory used by the different methods in the libraries. Memory measurements are only carried out on methods that have memory allocations identified by the analysis in Section 4.3.1. WCET is tested with JOP's distribution WCET tool, WCA [116]. We check that loop bounds are correctly found. Our synthetic test-bench for this part of the testing is a SCJ application with shared data structures in mission memory that are accessed from a set of `PeriodicEventHandlers` (PEH).

As a final step, two additional, more complex applications were tested. First, the new `java.util` collection classes were used as drop-in replacements for the shared data structures in the parallel `miniCDj` benchmark [135]. The `miniCDj` benchmark is a SCJ version of the benchmark described in [65]. `miniCDj` implements an air traffic controller simulator that generates artificial radar frames containing airplane positions. The frames are processed to detect possible collisions. For the parallel version, one PEH generates the radar frames and a selectable number of `AperiodicEventHandlers` process them.

The second test uses a SCJ version of a watchdog application running on top of the Cubesat space protocol (CSP) [14]. CSP is a network-layer protocol designed at Aalborg University that is used by small space-research satellites called Cubesats. The watchdog application has one PEH that sends packets to a set of nodes and one PEH functions as a router. An interrupt service routine adds incoming packets into the router's queue. For our experiments, one of the CSP nodes was an on-board satellite computer used in commercial Cubesats. The application has three main data structures that handle packets, connections and sockets. We replaced the data structure used to handle packets with our `Vector` implementation. Functionality was not affected nor were any scope-related issues introduced. An interesting result was a reduction of almost 7% of the use in immortal memory; in the original implementation, the router PEH needs additional packet-managing structures.

4.3.7 Discussion

Based on our analysis and the implementation of our reusable libraries, we found certain issues that are worth discussing. We comment on the issues of setting loop bounds for library code, idioms and patterns for WCET analysis, application developer considerations while using library code for SCJ development, certification issues related to library code, and the stability of our modified classes between releases of the OpenJDK.

4.3.7.1 Loop Bounds for Library Code

Because library code is intended to be a working and proven solution for the development of reusable software components, it is not possible to set the maximum number of iterations of loops in library code as a fixed constant, as can be done for specific applications. For example, a loop that iterates over the elements of a collection will need that the maximum number of iterations be the size of the collection, which will not be the same for different instances. We therefore need to rely on the ability of analysis tools to automatically extract such bounds. The importance of finding the maximum number of iterations is that such value is needed for WCET/WCMEM⁵ analysis.

To automatically find loop bounds with our WCET tool, some loop exit conditions had to be changed. For example the loop below iterates over all the elements of a linked list (one linked list per element in the `tab[]` array) and stops once the last element is found (i.e., the `next` pointer of an element is `null`).

```
for(Entry e = tab[i]; e != null; e = e.next)
```

The modified version of the loop is:

```
Entry e = tab[i];
for (int j = 0; j < entries.length; j++){
    if(e == null) break;
    ...
    e = e.next;
}
```

In the original code we don't know how many elements the list has and therefore we need the boolean test (test for a `null` element) as the loop stop condition. In the modified code, the `entries` element is a fixed-size pool of entries. The reason for this change is that, even if the maximum number of elements in the original list was known, our WCET tool currently cannot automatically find bounds on loops whose exit condition depends on testing boolean variables. In the modified loop, the worst-case iteration count, `entries.length`, can be propagated in the data flow analysis of the WCET tool and identified as the loop bound. The `break` statement is necessary to avoid iterations being performed when they are not necessary.

⁵Worst Case Memory consumption

An alternative approach for bounding loops in library code is the use of standard Java annotations to pass symbolic information about loop bounds, as proposed in [53], where loops can be bounded with annotations of the form:

```
@LoopBound(max=elementCount)
for (int i = 0; i < elementCount; i++){...}
```

where the maximum number of iterations, `elementCount` in the code above, is obtained from an annotation attached to the declaration of the class instance implementing the method with the loop. However, such non-standard annotations require the use of a modified Java compiler and therefore this option is not implemented in our libraries.

Annotations for loop bounds in `String` and `StringBuilder` objects are more difficult to handle with such type of annotations because those objects can be created in different ways e.g. explicitly with the `new` keyword, with the `toString()` method, when declaring constant strings, as a result of concatenation with the “+” operator, etc. In these cases there is no easy way to propagate information about the internal character array size to internal methods containing loops. In our libraries, we rely in creating strings with a fixed maximum size (see Section 4.3.2) and propagating such value in the data flow analysis.

4.3.7.2 Programming Idioms and Patterns for WCET Analysis

Another issue found when performing WCET analysis of library code was the use of overridden implementations of `Object.equals()`. The problem here is that the call graph generated during the analysis contains cycles, and the WCA tool is not able to handle this type of recursion. Cycles in the call graph were observed in classes that use the `java.util.Map.Entry` inner class. Testing for equality between two entries requires testing for equality between the pair of key-value mappings contained in the entry. The key and value objects will call their own implementation of the `equals()` method. One possible solution could be to restrict the types of objects that can be stored as key-value mappings, and to implement a different form of equality that avoids the use of an overridden form of `Object.equals`. This is, however, too restrictive on the types of key-value objects that can be used in a map.

The delegation pattern also generates cycles in the call graphs. For example, the methods of the `AbstractList.Sublist` inner class make calls to the “real” implementation of a `List` passed as argument to the constructor. To avoid

cycles, the annotation system proposed in [60] can be used to tighten the number of possible receiver types of a method invocation.

The solution to this issue will require a stronger analysis and to modify our WCET tool which was not done for the development of this thesis.

4.3.7.3 Application Developer Considerations

Many of the problems outlined in Section 4.2 can be solved by modifications of the Java class libraries source code. Such modifications can minimize, but cannot completely eliminate the occurrence of scoped memory protocol errors in a complete application. For example, it is the responsibility of the application developer to use caller allocated results or arguments to methods correctly, so as to avoid illegal assignments. We have presented patterns for SCJ memory usage in Section 4.1. Furthermore, a typing system based on annotations, such as the one described in [89], can be useful in this case. This typing system adds extra information about the scope of the different elements of an application, which can later be retrieved to perform static analysis. In addition, which restrictions exist for passed arguments and returned results should be considered when overriding library method implementations.

4.3.7.4 Certification Issues in Library Code

As a final remark, it is important to note that the use of reusable components and libraries for the development of safety-critical systems presents additional challenges, e.g., unused code from a library introduces code that is not traceable to requirements (dead and/or deactivated code). Certification standards, such as DO-178C [6], expect that code not associated to requirements is either eliminated or that requirements for the code are developed [1, 2, 3, 4, 45]. Moreover, dead and deactivated code will appear as noncovered code during structural coverage analysis and if detected in a late phase of the certification process may require complete re-verification of the system [102]. Therefore, the benefits of re-usability should overcome a potential increased certification effort and costs.

Another type of issue is the encapsulation of data, as this complicates robustness testing [1], i.e., “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [61]. Robustness tests will not be able to access library-private data.

4.3.7.5 Stability Between Releases of the JDK

By comparing the source code between JDK6 and JDK7 of the modified classes in this work, we found very few changes. The changes concentrate in methods to ensure capacity, the use of the enhanced for-loop and generics. Those minimal changes made our libraries relatively stable between JDK's releases.

4.4 Summary

In this chapter we have analyzed possible use patterns of the scoped memory as defined in safety-critical Java aiming at helping in the development of applications. We have also analyzed, identified, and proposed solutions to common problematic patterns and idioms present in three of the most used Java libraries: `java.lang`, `java.util`, and `java.io`.

Temporary storage offered by nested private memories can be exploited with the use of scoped loops which are entered more than once per handler release. If results are to be preserved for the next loop iteration, a simple approach is to update global objects in immortal memory. However, the use of objects in immortal memory is limiting and not thread safe because static fields are only allowed to refer to other immortal objects and multiple threads may overwrite each other's parameters. A different approach is the use of auxiliary objects to return results and pass arguments from and into the `Runnable` object required by the different static methods available to traverse the scope stack, i.e. `enterPrivateMemory()`, `executeInOuterArea()`, and `executeInOuterAreaOf()`. Returning objects from private or nested private memory areas is accomplished by an allocation context change and by saving the reference to the newly allocated object in a field of an auxiliary object. The complexities of creating auxiliary objects, passing references and saving results can be hidden by means of encapsulated methods.

Problematic patterns and idioms present in library code arise from the absence of a GC and restrictions on how objects can refer to each other. Safety-critical systems are more restricted and less dynamic than non safety-critical systems. These characteristics allows for restrictions and simplifications to be made for the implementation of reusable libraries for SCJ. We have adapted representative sample library classes from those allowed in SCJ as a first step towards the development of reusable libraries. The provided classes have predictable memory consumption, are WCET analyzable, and maintain referential integrity between objects created and used internally by the classes.

Evaluation

In this chapter we present an evaluation of our SCJ's implementation. We evaluate two main categories: (1) performance and timeliness and (2) compliance to the SCJ specification. For the performance and timeliness category we used the miniCDj benchmark,¹ an application-based benchmark, and we developed specific benchmarking methodologies where we evaluate (1) the accuracy of periods (i.e. release jitter), (2) linear-time memory allocation, (3) aperiodic event handling execution according to priorities, and aperiodic event queue overflow policy and size, (4) dispatch latency for interrupts, (4) context switch preemption latency, and (5) correct priority inversion avoidance when executing synchronized methods. In the compliance category we test how good our implementation adheres to the SCJ profile by using an early work on a technology compatibility kit (TCK) for SCJ developed at Purdue University.

Our test platform is the Altera DE-2 70 evaluation board with a Cyclone II FPGA and 2 MB of external SRAM with access latency of 3 clock cycles. JOP runs in the FPGA at 60 MHz and is configured with 4 KB method cache with 32 blocks and a stack cache of 512 KB.

This chapter is based on the published paper: “*An Evaluation of Safety-Critical Java on a Java Processor*” [104].

¹The miniCDj benchmark is a reduced version of the CDx benchmark described in [65] and adapted for SCJ.

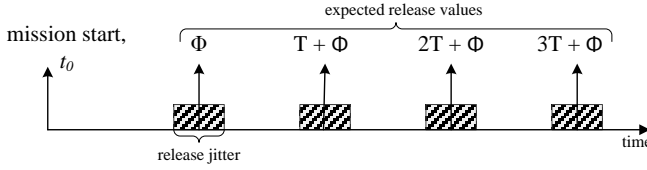


Figure 5.1: Precision of periods

5.1 Microbenchmarks

In this section we present the micro benchmarks developed to test our SCJ implementation. Each test is organized as an independent mission, where we use the mission memory to store test-specific configuration data. Test results and statistics are computed and displayed during mission cleanup phase. For the tests described in Sections 5.1.1, 5.1.2, 5.1.4, 5.1.5 we used JOP's cycle counter for accurate timing measurements. For the tests of Sections 5.1.3 and 5.1.6 SCJ's clock API was used, as we do not need a high degree of accuracy.

5.1.1 Accuracy of Periods

The majority of the computational load in real-time systems comes from periodic activities (e.g. sampling sensor data and applying control laws) [30]. Support for periodic activities is provided in SCJ via the `PeriodicEventHandler` class where an application developer adds its own functionality by overriding the `handleAsyncEvent()` method and provides a *start* time and a *period*. The *start* value represents an offset measured from the start of the mission until the first release of the PEH (Φ in Figure 5.1).

Ideally, the j -th release of a PEH will happen at integer multiples of the PEH's period plus the initial offset, as shown in Figure 5.1. However, in practice this is not the case as there are many factors that might delay the release of a PEH and therefore introduce release jitter. Some of those factors are inherent to the platform where the set of PEHs execute such as the granularity of the clock used to trigger the periodic events and the time it takes for the platform to notice those events [77, 29]. There are also factors that do not depend on the platform but on the particular set of tasks (in this case PEHs) that is being executed because executing higher priority tasks at the release time of lower priority tasks will delay the execution of the latter.

Table 5.1: Measured PEH start time deviations

Period	Max (us)	Min (us)	Avg (us)	Stdev (us)	Jitter (us)
5	68.8667	66.5333	68.6312	0.1587	2.3333
10	69.1667	66.6000	69.0245	0.1586	2.5667
30	69.6167	67.3500	69.6141	0.0725	2.2667
50	69.3667	67.0333	69.2006	0.1774	2.3333
100	69.2000	66.6500	69.1972	0.0811	2.5500
150	68.9000	66.6667	68.7582	0.1537	2.2333
300	69.5667	66.8833	69.2897	0.1985	2.6833
500	68.8667	66.5333	68.7248	0.1551	2.3333

To have an idea on the accuracy of successive periodic releases, in this part of the evaluation we measured the component of the release jitter that is inherent to our system. Our test setup uses 8 different periodic tasks, and each of this tasks is included in a single mission (a single PEH per mission). For each mission, the period of its PEH is set to 5, 10, 30, 50, 100, 150, 300, and 500 ms respectively. The PEH executes no logic in its `handleAsyncEvent()` method and it is rescheduled by the SCJ framework to be release at the next ideal release time (see Figure 5.1) by calling the `waitForNextPeriod()` method. For our experiments, we set the initial delay (offset from mission startup) to 0 (i.e., $\Phi = 0$ in Figure 5.1) and proceed as follows:

1. We first obtain the mission start time, t_0
2. We measure the actual release time of the j -th instance of a PEH, t_r^j , equal to the time at which its `handleAsyncEvent()` method is executed
3. We calculate the time interval between the actual j -th PEH release time and the start of the mission, $t_r^j - t_0$
4. We calculate the deviation of the start time for the j -th PEH release, Δ_j , as $\Delta_j = (t_r^j - t_0) - jT$

Table 5.1 shows the values of Δ_j measured for 1,000 releases of the single PEH. The release jitter is thus equal to the maximum deviation of the start times (difference between the minimum offset and the maximum offset) among all releases of the periodic task [30]. In this case, the release jitter has a maximum value of 2.68 μ s.

The values in Table 5.1 represent the time it takes for the hardware to generate an interrupt (in this case the timer interrupt), the interrupt dispatch latency,

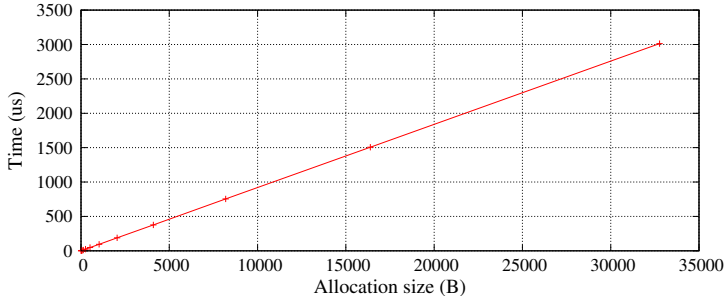


Figure 5.2: Linear time memory allocation time

and the scheduling/dispatching time. This is indeed the case, as it will be shown latter in sections 5.1.4 and 5.1.5.

5.1.2 Linear-time Memory Allocation Time

SCJ requires that mission and private scoped memories be linear-time memory areas, i.e., memory regions where the allocation time is proportional to the size of the allocated data. Immortal memory however is not required to be of linear-time type [77]. If allocations in immortal memory are restricted to the initialization phase then whether or not the immortal memory is of linear-time type is not relevant. However, in mission phase, allocating data in immortal memory can affect the timeliness of the system as allocations may have variable execution times. In our implementation, all the SCJ memory areas are derived from a single system class, called `Memory`, that provides linear-time allocations.

To test the linearity in memory allocation times ideally we would like to measure the time it takes to perform the pure memory allocation, that is, without considering the time to execute an object's constructor method. The problem is that we can only allocate memory as new objects are created and even if we know the size of the objects being created (possibly obtained with the `SizeEstimator` class), measuring the time it takes for object allocation will also include the execution time of the constructor. To measure the linearity of memory allocations we use a different approach which consists on measuring the time it takes to allocate a variable number of bytes of an array of integers. The number of bytes allocated was varied from 8 to 32 K and each measurement for every allocation size was repeated 1,000 times.

The results of this test are shown in Table 5.2 and Figure 5.2. Figure 5.2 shows

Table 5.2: Measured scoped memory allocation times. The allocation size is in bytes, the system clock is of 60 MHz, and the timer used for measurements has a 1 clock cycle resolution.

Size	Cycles	Time (us)
8	56	0.9333
16	98	1.6333
32	184	3.0667
64	360	6.0000
128	712	11.8667
256	1416	23.6000
512	2824	47.0667
1,024	5768	96.1333
2,048	11296	188.2667
4,096	22592	376.5333
8,192	45184	753.0667
16,384	90368	1506.1333
32,768	180736	3012.2667

a plot of the measured allocation times registered where it can be seen the linearity in the allocation times. The linearity in allocation times comes from only requiring a pointer to be bumped while performing memory allocations (the *allocPtr* in Figure 3.9). From Table 5.2 we see that there is no variation in the measurements, a desirable characteristic in real-time systems where consistency is the main concern.

5.1.3 Aperiodic Event Handling

Real-time systems also need to respond to events (internal or external) that occur at random points in time. Handling this type of events has to be done as they occur, without disturbing the system's main application logic, which is usually executed by periodic activities [132]. Unanticipated events can be handled by *aperiodic* or *sporadic* tasks. An *aperiodic* task has either soft or no deadlines while *sporadic* tasks have hard deadlines and a minimum inter-arrival time [76].

SCJ does not provide support for sporadic tasks, as there is no detection of minimum inter-arrival time violations. Only aperiodic tasks can be implemented through the `AperiodicEventHandler` (AEH) or `AperiodicLongEventHandler` (ALEH) classes. Both AEHs and ALEHs must have a priority and, in the absence of shared resources, they will not interfere with the execution of higher

priority PEHs. However, PEHs with lower priorities can miss their deadlines due to a higher priority AEH or ALEH with a large execution time. As opposed to RTSJ, where there is a configurable-size queue of events to service bursts of events, in SCJ the queue is of fixed size and equals to one with a queue overflow policy set to REPLACE, i.e. to overwrite pending event releases.

For this part of the evaluation, we test that: (1) high priority PEHs do not miss their deadlines, (2) the event queue size is equal to one, and (3) the queue overflow policy is set to replace. Our setup is as follows: we generate the two task sets of PEHs shown in Table 5.3 with a processor utilization of 69% and 88%. The 69% is chosen as it is the theoretical utilization limit to guarantee schedulability under rate monotonic [75] and the 88% value represents the average case bound for rate monotonic [73]. Schedulability of the task sets was verified with the TIMES tool [13]. We then increase the total load of the system by generating a burst of events that are to be served by a single ALEH. An ALEHs is used instead of an AEH because we can piggy-back a payload of type `long` to the servicing of an event request and then use this payload to identify which event is being serviced. The priority of the ALEH is chosen to be a value between the priorities of the PEHs with the purpose of dividing the periodic task sets in two parts: one that can potentially miss deadlines ($Prio(PEH_i) < Prio(ALEH)$) and one that must not miss deadlines ($Prio(PEH_i) \geq Prio(ALEH)$). The ALEH priority was set to be $Prio(PEH_3) > Prio(ALEH) > Prio(PEH_4)$.

The events are generated using a PEH (the “ST” task in 5.3) that fires the ALEH at random times. The arrival times of events follows a Poisson distribution with a mean arrival time of 200 ms ($\lambda = 5$). The Poisson distribution is chosen because it can accurately model the arrival of random events in time due to its no-memory property [92].² Ten different sets of random events generated from the mentioned distribution were used in our experiments.

The service time of each event, i.e. the execution time of the ALEH, varies from 15 ms to 200 ms in order to increase the aperiodic load of the system. The aperiodic load is a function of the number of serviced events, N , and the ALEH execution time, C , and is calculated as $NC/\Delta T$, where ΔT is the total running time of the experiment, in this case 50,000 ms.

Figures 5.3 and 5.4 show the results of our experiments. In Figures 5.3a and 5.4a we see the increase in the total aperiodic load as the event service time increases. In Figures 5.3b and 5.4b we see how the total number of events serviced decreases as the event service time increases. This reduction is a consequence of events overwriting each other if they arrive too close or if they arrive while still servicing a previous event as only the most recent event received will

²The arrival of one event has no influence on the arrival/non-arrival of a latter event

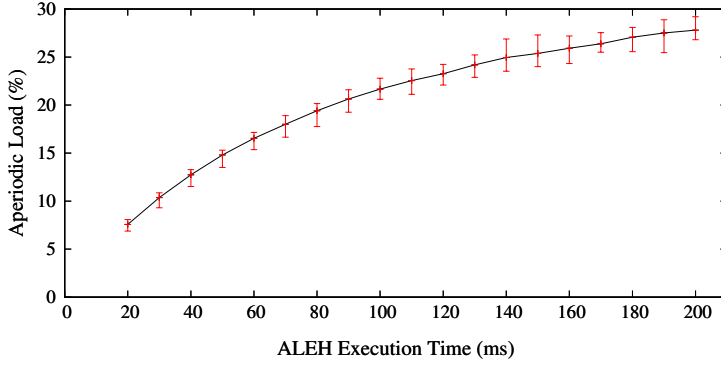
Table 5.3: Task sets used for the aperiodic event handling tests. The tasks are listed in descending priority order, deadlines are equal to periods, and the “ST” task is the PEH that fires the ALEH and runs at the highest priority

(a) 69% utilization task set				(b) 88% utilization task set			
Name	C (ms)	T (ms)	U	Name	C (ms)	T (ms)	U
ST	4.174	10	0.4174	ST	4.170	10	0.4170
PEH_0	5.000	200	0.0250	PEH_0	5.000	150	0.0333
PEH_1	5.000	250	0.0200	PEH_1	5.000	155	0.0323
PEH_2	15.000	300	0.0500	PEH_2	15.000	300	0.0500
PEH_3	15.000	500	0.0300	PEH_3	25.000	500	0.0500
PEH_4	20.000	750	0.0267	PEH_4	25.000	550	0.0455
PEH_5	20.000	1125	0.0178	PEH_5	30.000	1000	0.0300
PEH_6	25.000	1250	0.0200	PEH_6	50.000	1250	0.0400
PEH_7	45.000	2000	0.0225	PEH_7	110.000	1500	0.0733
PEH_8	50.000	2500	0.0200	PEH_8	110.000	1750	0.0629
PEH_9	100.000	2525	0.0396	PEH_9	100.000	2000	0.0500
Total Utilization			0.6890	Total Utilization			0.8843

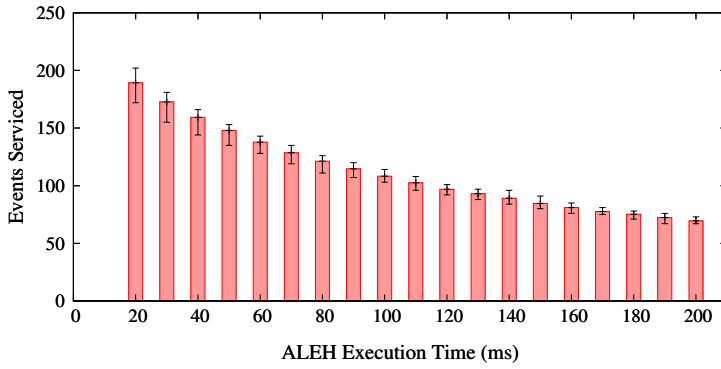
be serviced. We checked that only the most recently arrived event is serviced by using the `long` parameter of the released ALEH to identify which event is being serviced. In this way we see that our implementation has a REPLACE policy for the overflowing of an event queue and that the size of this queue is equal to one. In JOP this queue is implemented as a one-element array. As we are using a probability distribution to generate our events, the total number of events released is variable with a maximum of 259 and a minimum of 217. Therefore, Figures 5.3a, 5.3b, 5.4a, 5.4b show the average values (solid lines or bars) together with the maximum and minimum values.

Figures 5.3c and 5.4c show the number of deadlines missed normalized to the total expected releases per PEH ($T/\Delta T$). Deadlines start to be missed as the aperiodic load reaches around 10% and 8% for the 69% and 88% periodic loads respectively. High priority PEHs ($Prio(PEH_i) \geq Prio(ALEH)$) never miss a deadline and therefore are not included in figures 5.3c and 5.4c. Furthermore, the lowest priority PEHs are the most affected by the increase in the aperiodic load as they are the first to miss their deadlines and the ones that miss the most deadlines. In these figures only average values are shown.

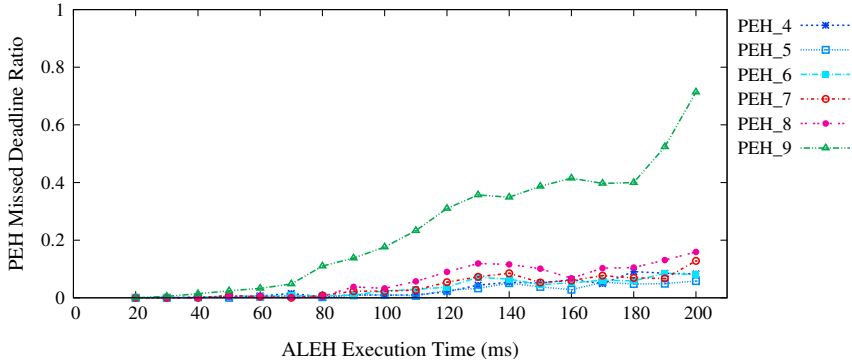
We ran a similar experiment with several ALEHs running at the lowest priority. The results showed that PEHs do not miss any deadline and that the aperiodic load stays below 12% and 31%. As the number of available low priority ALEHs



(a) Aperiodic load as a function of the ALEH execution time

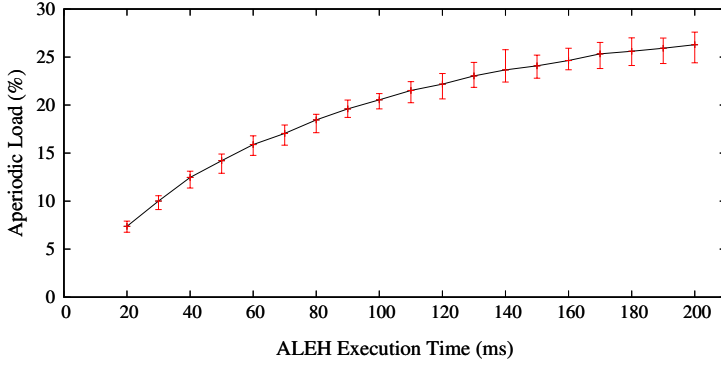


(b) Number of events serviced as a function of the ALEH execution time

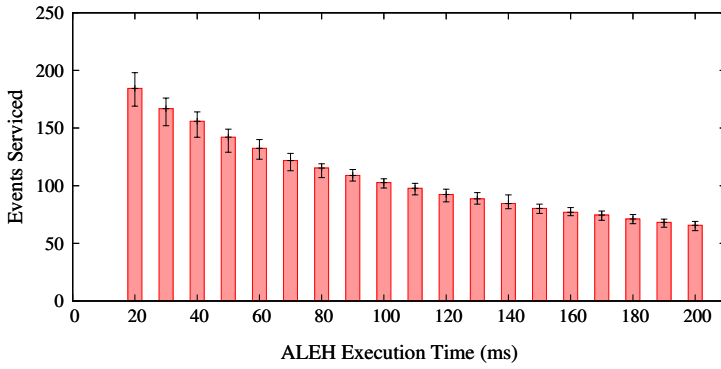


(c) Fraction of missed deadlines per PEH as a function of the ALEH execution time

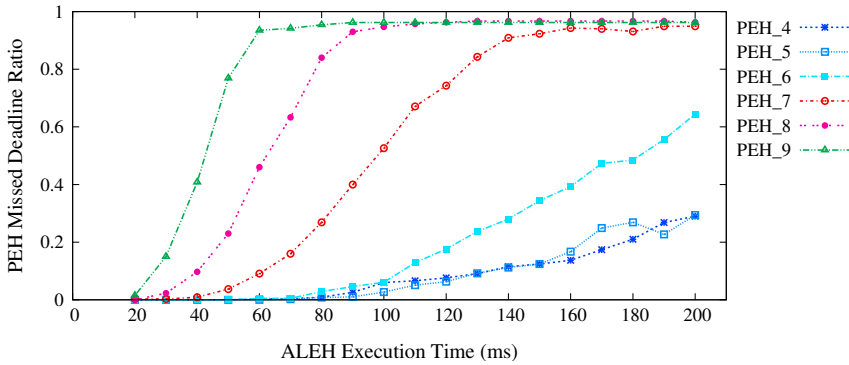
Figure 5.3: Aperiodic load (a), events serviced (b), and PEH's missed deadlines (c) as a function of the ALEH execution time for the 69% periodic load experiment.



(a) Aperiodic load as a function of the ALEH execution time

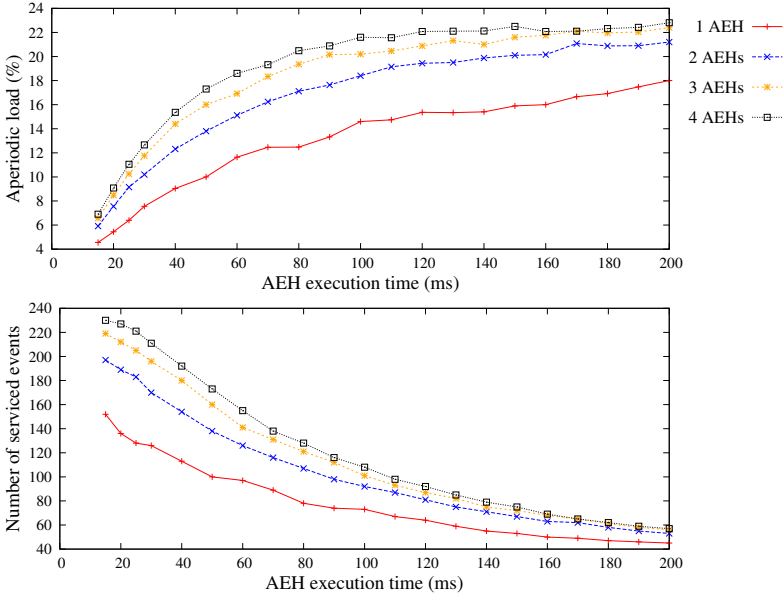


(b) Number of events serviced as a function of the ALEH execution time

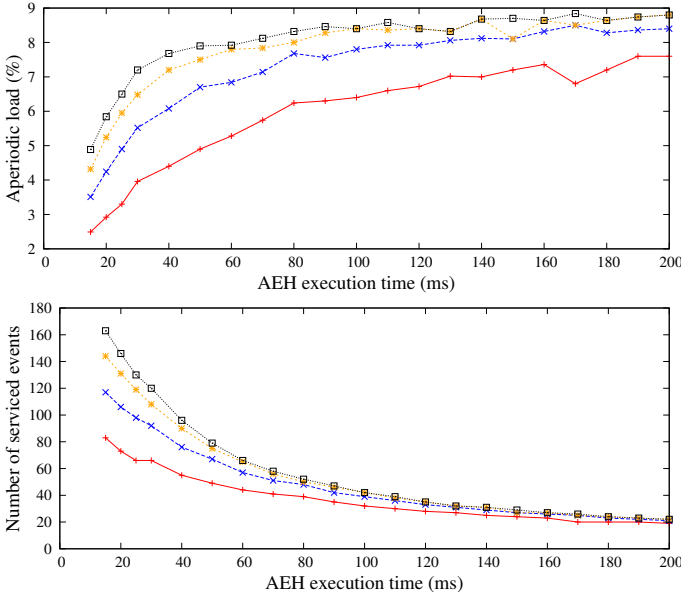


(c) Fraction of missed deadlines per PEH as a function of the ALEH execution time

Figure 5.4: Aperiodic load (a), events serviced (b), and PEH's missed deadlines (c) as a function of the ALEH execution time for the 88% periodic load experiment.



(a) Aperiodic load and serviced events for a 69% periodic load.



(b) Aperiodic load and serviced events for an 88% periodic load.

Figure 5.5: Aperiodic load and number of serviced events as a function of the AEH execution time for a periodic load of 69% (top) and 88% (bottom). The priority of the AEHs is lower than the priority of the PEHs.

increase, more events can be serviced, giving an intuition on how to dimension the system in terms of the number of aperiodic event handlers required to serve aperiodic events with a known arrival pattern. The results are shown in Figure 5.5a and Figure 5.5b for the 69% and 88% periodic load respectively.

5.1.4 Dispatch Latency for Interrupts

SCJ supports the notion of first level interrupt handlers in Java through the `ManagedInterruptServiceRoutine` (MISR) class. Similar to other MSOs, MISR objects have to be registered to a specific mission and the interrupt service routine (ISR) code is implemented by overriding the `handle()` method of RTSJ's `InterruptServiceRoutine` class, which in SCJ is more restricted than its RTSJ counterpart. In general, in Java an ISR can be implemented as a handler or as an event [114]. The handler approach uses a method invoked by the hardware while the event approach uses a form of asynchronous event to fire an AEH or unblock a managed thread. The advantages and disadvantages of both methods are described in more detail in [114].

Interrupt handlers are executed at hardware priorities, i.e., priorities that are higher than those of ordinary schedulable objects, and can delay the completion of schedulable objects. It is therefore important that interrupts are dispatched as fast as possible. An ISR can be preempted (or interrupted) only by a higher priority interrupt, assuming that the ISR disables only lower or equal priority interrupts. Nested interrupts can make the system more responsive to external events but can increase the blocking times due to interrupts.

In this part of our evaluation we measure the interrupt dispatch latency when using the SCJ framework and compare it to the dispatch latency when JOP's API is used. For that we measure the time between the generation of an interrupt and its delivery.³ How interrupts are generated is implementation dependent, making it difficult to measure the time of the interrupt generation in a way that can be ported between SCJ implementations. Measuring the time when the interrupt is delivered is however non-implementation dependent and is equal to the time when the `handle()` method of the `InterruptServiceRoutine` is executed.

In our test setup we used JOP's system device to generate interrupts triggered by software. The interrupt generation time is therefore the time just before triggering the interrupt. The interrupt delivery time will be the time when the

³Generation is the hardware mechanism that makes the interrupt available to the Java program while delivery is the action that invokes an ISR [77]

Table 5.4: Number of interrupts serviced when measuring the interrupt dispatch latency.

(a) Using the SCJ framework		(b) Using JOP's API	
Mission No.	Number of Interrupts	Iteration No.	Number of Interrupts
1	788	1	813
2	834	2	701
3	820	3	917
4	800	4	304
5	804	5	204
6	822	6	391
7	798	7	705
8	832	8	903
9	811	9	294
10	823	10	190

first statement of the `handle()` method of the ISR is executed. Interrupts will be triggered by a PEH when the SCJ framework is used or by the main thread when JOP's API is used.

When the SCJ framework was used, we run a total of 10 test iterations. Each test iteration is structured as an SCJ mission with 10 PEHs from where interrupts were triggered at random times and from different PEHs. Similarly a total of 10 test iterations were executed with JOP's API. An iteration in this case consists of randomly triggering interrupts from within the body of a loop in the main thread. A total of 10 loop iterations were run.

The number of interrupts triggered on each test iteration when using the SCJ framework and JOP's API are shown Table 5.4a and 5.4b respectively. Table 5.5 compares the measured dispatch latencies when using SCJ's MISR and when using a plain runnable invoked by the hardware as the ISR. The overhead when using the SCJ implementation comes from invoking four methods as opposed to only one for the non-SCJ version. The additional methods result from an extra indirection (invoking the `InterruptServiceRoutine`'s `handle()` method) and having to execute the ISR in a private memory. Entering a private memory requires two methods: (1) its `enter()` method, invoked by the SCJ framework, and (2) the `run()` method of the runnable object that will execute in the private memory.

Table 5.5: Measured interrupt dispatch latency.

	Cycles	Time (us)
SCJ	1462	24.3667
Non-SCJ	266	4.4333

Table 5.6: Measured context switch latency times

Period	Max (us)	Min (us)	Avg (us)	Stdev (us)
5	65.6167	58.1500	64.8901	0.4701
10	65.7000	58.1500	65.2281	0.6212
30	66.0667	58.1500	65.1334	0.6791
50	66.0667	58.1500	64.4568	0.4831
100	65.7000	58.1500	65.2053	0.6060
150	66.0667	58.1500	64.8920	0.4710
300	66.0667	58.1500	65.2309	0.6206
500	66.0667	58.1500	65.0801	0.6872

5.1.5 Context Switch Latency

Context switch latency is a source of overhead that can affect the performance of the system. To measure the context switch delay in our system we used two PEHs, one with high priority, PEH_H , and the other with a lower priority, PEH_L . The start times of both PEHs are selected in a way that makes PEH_L begin its execution before PEH_H . PEH_L has a long period, long enough to execute only one release during the test, and an infinite loop that constantly updates a variable with the current time, t_L . PEH_H has a shorter period and reads the current time, t_H , as the first statement of its `handleAsyncEvent()` method. Because PEH_H will always preempt PEH_L , the context switch latency will be equal to the difference between t_H and t_L .

The results of our measurements are shown in Table 5.6 In this test we used 10 high-priority PEHs with different periods and 1,000 context switches for each test were registered. The minimum value is obtained when t_L is updated just before the context switch while the maximum value is obtained when the context switch happens just before updating t_L .

Table 5.7: Task set for synchronization test

Priority	Offset (ms)	Critical section (ms)	WCET (ms)
HIGH	400	100	300
MEDIUM	300	0	400
LOW	0	500	600

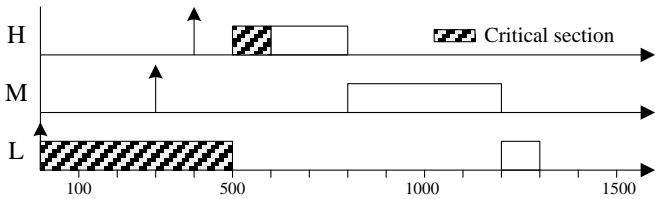


Figure 5.6: Expected execution pattern of the task set used to test the correct synchronization implementation (Table 5.7)

5.1.6 Synchronization

Contention for shared resources among managed schedulable objects can result in priority inversions. To avoid priority inversions, SCJ requires the use of the priority ceiling emulation (PCE) protocol to control access to shared resources.

To test a correct PCE implementation, we use the task set of Table 5.7 that consists of three PEHs with low, medium, and high priorities. The offset column is the time of the PEH's first release, the critical section column is the amount of time that the PEH requires a shared resource, measured from the time the PEH gets the shared resource. The WCET column is the total execution time of the PEH. The low and high priority PEHs require access to a shared object located in mission memory and its lock is first acquired by the low priority PEH at time $t = 0$. Then, at time $t = 400$, the high priority PEH will try to get the lock to the shared object. With this setup we will force a potential priority inversion in the task set. We expect that with a correct PCE implementation the execution pattern to be as shown in Figure 5.6. The measured release delay and response times for all PEHs are shown in Table 5.8.

JOP's implementation of the PCE protocol is done by globally disabling all interrupts within synchronized methods. As the scheduler is a first level interrupt handler fired by the programmable timer interrupt, with interrupts globally disabled there will be no scheduling decisions, i.e., once a PEH has acquired a lock

Table 5.8: Measured timing values of synchronization task set

PEH	Release delay (ms)	Resp. Time (ms)
HIGH	501.840	403.565
MEDIUM	802.397	907.115
LOW	0	1308.939

it cannot be preempted or interrupted. To provoke a priority inversion we need the medium priority PEH to be able to preempt the low priority PEH once it has acquired the lock to the shared object. However, this is not possible if no scheduling decisions are made. The object's monitor ceiling is therefore effectively equal to the maximum priority of all the possible managed schedulable objects that could ever acquire the lock of the shared object.

5.1.7 Discussion

By organizing each micro benchmark as a single mission we are also testing compliance to the mission-based programming model of SCJ. We are implicitly testing the mission life cycle by: (1) changing missions and executing them in a predetermined sequence, (2) resizing the mission memory before mission initialization, (3) providing mission termination mechanisms, (4) using nested private memories at mission cleanup, and (5) delaying the start of tasks with the `start` parameter of the `ReleaseParameters` class.

In order to make our micro benchmarks portable across SCJ implementations we developed them using the available public SCJ API features. However, when we used SCJ's clock API for timing measurements an overhead is introduced due to additional method calls and normalization operations. Moreover, saving such measurements in *clock objects* allocated in shared memory areas increases the complexity. The use of the clock API is kept for portability however there are some tests where it is necessary to increase the accuracy of measurements. For the tests described in Sections 5.1.1, 5.1.2, 5.1.4, 5.1.5 we used JOP's cycle counter instead.

One of the SCJ's features that complicates portability is the correct sizing of memory resources of MEHs, in particular for mission sequencers. Contrary to all other MEHs, mission sequencers do not have a private scope and they execute either in the memory of its enclosing mission (or in immortal memory if it is the top level sequencer) or, during mission initialization, in the memory of the mission returned by its `getNextMission()` method. Memory requirements are

specified by the sequencer's `StorageParameters` that has two arguments whose value limit the amount of immortal and mission memory that the sequencer can use: `maxImmortal` and `maxMissionMemory` respectively. The problem is that, in addition to the explicit memory requirements set by the application developer for the mission being initialized, an implementation may need to allocate additional data structures and objects in immortal and/or mission memory, e.g. structures to register the handlers. This amount of memory required by the implementation is unknown to the application developer meaning that knowledge of the mission's memory requirements alone is not enough to correctly set the `maxImmortal` and `maxMissionMemory` parameters.

A similar issue applies for the `sizes` array, also an argument of `StorageParameters` as this array is completely implementation specific and most likely implementations will use this array in different forms or even ignore it.

5.2 SCJ's TCK and miniCDj

As SCJ is still in an early draft phase, currently there is no official TCK available that can be used to check our implementation against the specification. There is however a reduced functionality TCK done at Purdue University that we used to check the portions of the SCJ implementation that this TCK exercises. The TCK is organized as a collection of SCJ applications and each of them evaluate different features of the SCJ specification. It is however not possible to apply all the tests of the TCK to our implementation because some tests rely either on outdated or L2-specific features. Other tests require access from user space to non-public methods of the SCJ specification such as testing that private memories are entered by only one thread. This test is done by *explicitly creating* private memories and entering them with the `enter()` method. To run these type of tests we need access to implementation details of the `javax.safetycritical` package. We can either use a delegate class in the `javax.safetycritical` package to access such methods or make them temporarily public. We used the latter option to exercise such type of tests.

The TCK tests successfully executed are summarized in Table 5.9. The reader is referred [136] for the details of the tests. Every subdivision in the table represents what is being tested on a single test case of the TCK.

Test 1 requires that all missions can be executed and that their executions do not overlap, that is, each mission is executed in a predefined order. Test 2 is satisfied by design, because there is no heap in JOP when scopes are used and there are no daemon activities in JOP (which typically depend on an operating

Table 5.9: TCK tests implementation and successfully passed

No.	Test description
1	Missions can be executed sequentially
2	AEH, PEH, NHRT are no-heap and non-daemon
3	The default ceiling for locks is <code>PriorityScheduler.instance().getMaxPriority()</code>
4	Nested calls from one synchronized method to another are allowed
5	The only scheduler is the preemptive priority-based scheduler with at least 28 priorities. There is no support for changing base priorities.
6	Test classes <code>CyclicExecutive</code> and <code>CyclicSchedule</code> to make sure all frames in a mission are issued sequentially according to the order of their creations
7	Only one server thread of control shall be provided by the VM at L0
8	The handlers shall be executed non preemptively on Level 0
9	Synchronized code is not allowed to self suspend on Level 0 and 1. An <code>IllegalMonitorStateException</code> is thrown if this constraint is violated
10	Priority Ceiling Emulation supported on Level 1+
11	Full preemptively scheduling shall be supported Level 1+
12	A preempted schedulable object must be placed at the front of the run queue for its active priority level
13	Each application uses a global mission scope in the place of immortal memory to hold global objects used during a mission
14	In initialization phase and mission phase all objects are allocated in mission memory
15	Each schedulable object has its private scoped memory
16	Object creation in mission scoped memory or immortal memory during the mission phase is allowed
17	MissionMemory is ScopedMemory
18	PrivateMemory is based on LTMemory
19	Deep nested private memory is supported
20	A given private scoped memory can only be entered by a single thread at any given time
21	Mission memory is resizable
22	The real-time clock should be monotonic and non-decreasing

system).

Tests 3 and 5 are also satisfied by design as we are in control of the default priority ceiling and the maximum priority values. Test 4 basically tests a correct implementation of synchronized methods. Test 6 is required for a correct cyclic executive implementation. Satisfying tests 7 and 8 is a consequence of how we implement L0. We are sure that only one thread will be executing, the main thread (see Section 3.3.1) and therefore no preemption can occur. Test 10 is also satisfied by the particular implementation of JOP's synchronized methods (see Section 5.1.6).

Level 1 applications will use JOP's scheduler, which is a fixed priority preemptive scheduler and therefore test 11 is satisfied. In JOP there are no independent priority queues for tasks with the same priority. Another way to see this is that there is a single one-element queue for each priority thus test 12 will be satisfied.

Test 13 is only testing that given a shared object in mission memory it will be accessible to all MSOs that can get a reference to it. Test 14 requires that the default allocation context when using the `new` keyword is mission memory, that is, it is testing that the infrastructure is able to change to mission memory once entering any of those phases. To satisfy test 15 it is required that the allocation context of any MSO is a private memory (i.e., neither immortal nor mission memories). Tests 17 and 18 are testing a correct memory hierarchy implementation.

Test 20 is satisfied also by design, as we do not have any method to explicitly create or enter private memory areas. We can only execute several MSOs in the shared memories. Test 21 is trivially satisfied, a mission memory either can or cannot be resized. Test 22 is checking a correct normalization of time values.

It is also important to mention that from the TCK, a maximum of 66 error messages can be obtained as the test cases fail. From those 66 errors, up to 35 apply to our implementation⁴ and the rest are either for L2 features, to test the Java native interface, or refer to features no longer available. Of those applicable failures none was observed.

To test our implementation with a complete application, we have used the miniCDj benchmark. We used six planes, 1,000 frames, and the period of the detector was set to 1,000 ms. No frame overruns were detected and values for the detector's execution time are shown in Table 5.10.

The variation in the computation time is because collision detection depends

⁴Each test in Table 5.9 can produce more than one error message.

Table 5.10: miniCDj benchmark execution time measurements

Min (ms)	Max (ms)	Avg (ms)	Stdev (ms)
422.94	744.05	568.03	75.72

on both, the current and the previous frames, thus generating a variable size space to search for potential collisions. The measured release delay is less than 3 microseconds thus it is not shown in Table 5.10.

To see if parts of our SCJ implementation may affect the execution times, we also run this test with scope checks enabled and disabled, and using both, JOP's microsecond counter and the SCJ clock API to drive the cyclic schedule. None of these showed to add considerable overhead. The main source of execution time comes from the large number of floating point operations that in JOP are done in software.

5.3 Summary

In this chapter we have presented an evaluation of the SCJ implementation on a Java processor. We have developed a series of micro benchmarks organized as independent missions in order to test timeliness and performance of the system. We have tested the accuracy of periods, linear-time memory allocations, aperiodic event handling, interrupt dispatch latency, context switch latency, and synchronization. In addition, we have used an application-oriented benchmark, the miniCDj benchmark, to test in a non-isolated way our implementation. We have also used a reduced version of Purdue's SCJ technology compatibility kit to show how well our implementation adheres to the SCJ specification.

Conclusions

In this thesis we have examined the use of the safety-critical Java profile for embedded systems through an implementation of the safety-critical specification for Java, JSR-302, on a time-predictable Java processor. In this work we have reported in our experiences while implementing the profile, explored hardware extensions for time-critical operations, and examined the impact of SCJ's scoped memory model on the development of applications and reusable libraries. Our implementation is open source and can be downloaded from JOP's main repository.

6.1 Main Results

The SCJ specification is reaching its maturity and with this implementation we have proved its feasibility on an embedded platform. Our implementation is kept completely in the Java space and even structures typically implemented in the target platform's native language (e.g., C, or assembly) or with some operating system support are implemented in Java (e.g., a task scheduler).

The two major issues we faced during this implementation were (1) the need to share package private information and (2) the use of the scoped memory model. Sharing private information between packages becomes problematic when such

information has to be hidden from the application developer, thus changing accessibility modifiers to public is not an option. Our solution to this issue uses a singleton delegation mechanism that avoids the use of reflection or other unsafe mechanisms such as those used by Sun¹ on its `SharedSecrets` class. The major drawback of this solution is the overhead introduced by an additional indirection when methods in a different package are accessed through an intermediate class.

The scoped memory model is perhaps the most difficult feature to use as both, SCJ applications and the SCJ infrastructure are affected by an extra dimension in the design space: that of conserving referential integrity. Implementing the SCJ infrastructure on top of this scoped memory model requires extra care as illegal references can be introduced. For example, in our implementation, the scheduler object and thread structures are allocated in mission memory to avoid illegal reference assignments between the scheduler and the MSOs that run under the scheduler's control. However, there are parts in our implementation where we were unable to avoid such illegal assignments. In those cases, we need to guarantee by design that such violations can be considered as safe, i.e., there is no risk of ending with dangling pointers. Formally proving that such violations are indeed safe will be the recommended approach however this is beyond the context of this work.

An example of a safe violation that can be guaranteed by construction is the case of the scheduler object itself, which is attached to an array of schedulers allocated in immortal memory. The scheduler is required only during mission execution meaning that after a mission termination the scheduler object can be safely collected as the application leaves the mission memory. A new scheduler is attached to execute the next mission in the transition to the mission execution phase of the next mission.

As the mission sequencer runs only at the initialization of the first mission or in between missions, we have simplified the mission sequencer and instead of executing it under the control of the scheduler, we run it on the main thread of the system. This simplification is not restricted to our implementation and can also be used by different SCJ implementations though only for L0 and L1. An RTSJ-based implementation will however require that the sequencer thread is executed in a `NoHeapRealtimeThread` as the main thread by default runs in the heap.

SCJ requires the use of three memory areas free of garbage collector interference: immortal memory, mission memory, and private memory. However, all three memory areas can be implemented with a single class [114]. Our implementation uses a single system memory class and to be compliant with the SCJ

¹Now part of Oracle Inc.

specification, the SCJ memory classes are implemented by delegating functionality to this system class. Scoped memories are sized according to their local allocation size and total backing store size requirements. To avoid fragmentation, each scoped memory is physically nested within its parent e.g., mission memory is an inner memory of immortal memory, private memories are inner memories of mission memory, etc.

The use of the scoped memory model and the absence of a garbage collector requires that reference assignment operations be checked in order to avoid making references from shorter-lived objects to longer-lived objects. However, the restricted memory model that SCJ provides greatly simplifies the reference assignment checks. In SCJ, the hierarchy of nested scopes grows in a linear stack and each scope is located at a unique nesting level within this stack thus allowing for simpler illegal reference assignment checks. Assignment checks can be performed by simply comparing the nesting level of the scoped memories where the objects involved in a reference assignment were created. We have presented three approaches to perform these checks, two of them can be implemented in any JVM and use the auxiliary object information to store the scope nesting level. The third approach is specific to our JOP implementation and uses simple hardware extensions to reduce the execution time of the reference assignment checks. Our hardware-based scope checks are performed as part of the execution of the reference assignment bytecodes. Those bytecodes are executed in the memory management unit of JOP. We showed that our hardware implementation adds a very small overhead to the memory management unit, around a 4% increase in logic resources. The performance gain is however strongly dependent on the application and on how frequent reference assignments are.

The execution, concurrency, and memory models constitute the core of SCJ. Within our implementation we have also provided support for other features that have been left unimplemented in previous works [96, 121]. Those other features are the raw memory interface, based on RTSJ's 1.1 raw memory API; the necessary framework to use managed interrupts; and managed long event handlers. We believe that such features are especially important in the context of embedded systems.

We have presented two ways of gaining access to raw memory areas through the use of the **RawMemory** API: (1) using native methods and (2) using hardware objects. Native methods are most likely what a JVM will use and in JOP native methods are mapped to special bytecodes implemented as microcode sequences. The use of hardware objects is specific, but not restricted to JOP. The use of native methods provides the most flexible option however the use of hardware objects is beneficial if we want to avoid native methods in application code as such methods break Java's type safety.

First level interrupt handlers use SCJ's `ManagedInterruptServiceRoutine` class. Using `ManagedInterruptServiceRoutine` objects introduces additional delay in the interrupt processing due to the extra indirection of executing the `handle()` method and because a private memory needs to be entered to execute the ISR code. In contrast, the delay of servicing an interrupt is smaller when using a plain method invoked by the hardware as ISR.

In addition to our implementation of the SCJ profile, we have also analyzed use patterns of the scoped memory aiming at helping in the development of applications. The scoped memory use patterns we provide are intended to be a working solution for a recurring problem: how to reuse the scoped memory available and how to move data between scopes. Returning objects from private or nested private memory areas is accomplished by an allocation context change and by saving the reference to the newly allocated object in a field of an auxiliary object. The complexities of creating auxiliary objects, passing references and saving results can be hidden by means of encapsulated methods.

We have also analyzed, identified, and proposed solutions to common problematic patterns and idioms present in three of the most used Java libraries: `java.lang`, `java.util`, and `java.io`. Problematic patterns and idioms present in library code arise from the absence of a GC and restrictions on how objects can refer to each other. However, the more conservative nature of safety-critical systems allows for certain restrictions and simplifications to be made for the implementation of scope-safe reusable libraries. We have modified representative classes of different standard Java packages that illustrate different ways to solve problematic patterns and idioms. This work can be considered as a first step towards the development of reusable libraries for SCJ. The provided classes have predictable memory consumption, are WCET analyzable, and maintain referential integrity between objects created and used internally by the classes. Our starting point for our scope-safe classes was the OpenJDK 6 source and our goal was to minimize the required changes in order to keep compatibility with standard Java applications.

To evaluate our implementation we developed a series of micro benchmarks organized as independent missions, used an application-oriented benchmark, and used Purdue's SCJ's technology compatibility kit. With our micro benchmarks we showed that: (1) our implementation has a small release jitter for periodic activities, (2) has linear time memory allocation times, (3) aperiodic event handling execution is done according to priorities and event queue overflow policy is set to replace, (4) preemption latencies have small variation, (5) a correct priority inversion mechanism for synchronized methods is implemented.

The application benchmark we used was the miniCDj benchmark. This benchmark simulates a collision detector implemented as L0 application. Maximum,

minimum, and average values of the collision detector execution times are reported by the benchmark. Rather than testing performance, our main goal while using the miniCDj was to successfully execute a third-party application in our framework and therefore fully exercise our L0 implementation.

With Purdue's SCJ technology compatibility kit we showed how well our implementation adheres to the SCJ specification. From the TCK, a total of 22 tests can be applied to our implementation. Those 22 tests can report 35 failures. We successfully ran every test with no errors reported. The TCK also contains tests that we cannot use because they either test L2 features, the Java native interface, or refer to features no longer available in the current version of the specification.

6.2 Lessons Learned

During the development of this thesis we have gained a good insight into SCJ's proposal for the use of Java in software for safety-critical systems. SCJ follows the same approach as the MISRA C [16] and Ravenscar-Ada [28] variants of the C and Ada programming languages respectively. This approach consists on providing a subset of a language that excludes features that affect timing predictability and complicate safety certification.

One of the strengths of SCJ is that its simpler subset can simplify the task of developing analysis tools. Even though the SCJ specification incorporates many annotations intended to implement static analysis tools [126], recent research has shown that it is indeed feasible to develop analysis tools that do not rely on such annotations [37, 14, 81]. Moreover, a simplified subset of a language is easier to express in a formal way and therefore it is expected that implementing formal analysis tools for SCJ to be a feasible task. There is indeed research in this direction that uses the Java modeling language (JML) [49, 100, 122] to formally specify and verify properties of SCJ (the next section elaborates on this topic). From these arguments it follows that two of the generic challenges for safety-critical systems presented at the end of Section 2.2, namely those related to formal verification and development time and effort can be addressed with the new technology proposal of SCJ.

From an application developer perspective, SCJ at levels 0 and 1 provides a familiar programming model for real-time systems developers. However, one of SCJ's weak points is the absence of a garbage collector, which has made standard Java so popular. In our experience, the use of the scoped memory model initially leads to constant cycles of testing and debugging as we needed to keep

track of where objects are allocated in order to avoid illegal reference assignments. Correct use of the scoped memory model is indeed the most difficult part when developing SCJ applications. This change of mindset is most likely the biggest barrier that Java developers may face. However after getting used to the scoped memory model, development of SCJ applications become easier and more intuitive.

The development of the SCJ profile has followed a top-down approach, starting with the full Java language, enhancing several of its ill-defined areas (e.g., thread scheduling and dispatching) through the RTSJ, and finally restricting what methods and classes can be used, and how a developer can structure an application. Unfortunately, this top-down approach makes SCJ dependent on RTSJ and forces developers to know and possibly rely on another technology, namely the RTSJ. The consequences are that (1) the issues described in Chapter 3 regarding package crossing must be considered, (2) most RTSJ classes become greatly restricted and possibly empty, and (3) complex implementation features of RTSJ will also be part of an SCJ implementation that runs on top of the RTSJ. For safety-critical applications that need to be certified this dependency will increase the effort and cost and there will be a lot of dead and deactivated code. A simpler profile that follows a bottom-up approach would be desirable in order to avoid inheriting unnecessary complexity (see for example [98, 118, 23]).

A factor that limits the development of reusable software in SCJ comes from the restrictions that the scoped memory imposes to library code. Libraries in Java offer a working and tested solution to ease the development of reusable software. However, in SCJ, safe use of library code is restricted to the boundaries of scoped memories as any temporary or auxiliary object created by the library code can be safely allocated and referenced only from within the boundaries of scoped memories. The use of library code becomes more difficult when crossing scope boundaries (i.e., calling methods of library classes from different scopes) either as a result of the developer's intent or because the library classes make calls to other library classes (which is very common). If it is the intent of the developer to use library code in mixed scoped memory contexts, then the developer has to be aware of the memory allocation behavior of the code. Such behavior is not documented in standard library code and therefore, library code developed for SCJ must document, in addition to its functional behavior, its memory allocation behavior in terms of temporary objects and the expected allocation context of arguments and returned objects.

Special attention has to be paid on analysis of library code to ensure that space and time requirements are met. This task is not trivial, as existing library code uses the full potential of the Java language which can introduce programming patterns difficult to analyze, loops where automatic bound detection is difficult,

and reflective calls. There is also the alternative of developing reusable libraries from scratch instead of trying to use or modify existing libraries. Given the less dynamic nature of safety-critical software a balance between functionality, scope-safety, and predictability in terms of memory consumption and execution time can be achieved.

Despite the limitations mentioned, SCJ is a good choice for the development of embedded safety-critical systems as it brings the possibility to implement hard real-time systems using a safer language (strongly typed without pointer arithmetic). Once the developer is familiar with the use of the scoped memory model, development of SCJ applications becomes easier.

6.3 Future Work

During the development of this work we have covered many practical aspects on the use of SCJ for embedded systems. However, there are many issues not covered in this work which can provide a good basis for future work. We present in this section some possible research directions or issues that we would like to address in the future.

SCJ Level 2

To the best of our knowledge, there is no SCJ Level 2 implementation. All previous work has focused in levels 0 and 1 of the specification. L2 proposes a more dynamic programming model which is interesting from a practical point of view.

We would need to extend our implementation and add support for: (1) nested sequencers and missions, (2) managed threads, (3) the use of `Object.wait` and `Object.notify`. To implement nested sequencers and missions we need to remove our simplification of a sequencer implemented in the main thread and make it a full managed schedulable object. We also need to modify our concept of mission execution as currently we do not allow managed schedulable objects to be added to the scheduler at the mission execution phase.

With these additional features, our SCJ implementation will be a good research platform to test new features and proposed changes to the SCJ specification such as those proposed in [133] e.g., deadlines on mission sequencers. It would

also be interesting to see the challenges of verifying L2 applications and the level of acceptance that the L2 model can have in the real-time community.

Technology Compatibility Kit and Use Cases

Any Java specification request is composed of the specification itself, a reference implementation, and a technology compatibility kit. In this work we have used an early work done at Purdue University regarding a SCJ TCK [136] and while we could adapt it to the current version and use most of its tests, a complete and updated version of the TCK is necessary. As the SCJ specification is still evolving, such TCK will also have to be in a continuous update process.

Another issue related to testing SCJ implementations is the lack of use cases. Developing real-life use cases is an important and necessary step.

Verification Tools

The work of this thesis was focused on an implementation of the SCJ profile. The next step will be to focus on the development and integration of verification tools as verification is a key process on the certification of safety-critical systems. Here we describe what type of tools may be required from a certification perspective and we give some pointers into previous work.

As SCJ is intended to provide a reduced set of the Java language amenable to certification, it should be easier to apply or develop tools that help in this certification process. Safety standards such as DO-178C require verification activities such as analysis, testing, and verification of testing. The analysis process is involved with providing repeatable evidence of correctness [102] and it requires, amongst others, WCET analysis and memory consumption analysis. In this regard, it is reported by the authors of [14], a worst-case memory consumption analysis tool, that using SCJ's scopes, which use a backing store model to avoid memory fragmentation, simplify the object layout and therefore the memory consumption analysis.

Testing in the framework of DO-178C is *requirements-based* however, DO-178C allows formal methods to be used as alternatives to testing [6] and to this end requirements should be expressed in a formal language or as a logic function contract. Some work related to the second approach has been done in [?] where the authors derive conformance test suites for SCJ from formal specifications expressed in JML [72]. Integration testing, i.e., "the process of putting software

components together” [102], also suggested by DO-178C, can benefit from the use of JML. Work has been done in the context of SCJ in [49] where the author integrates functional and timing constraints for SCJ programs in SafeJML, an extension to JML, with the purpose of statically reasoning about timing properties in an application from timing properties of several independent components.

Verification of testing is involved with structural coverage analysis, i.e., with identifying code that was not exercised during testing. Modified condition/decision (MC/DC) coverage is one of the requirements in this category. Work has been done on automatic test generation for high MC/DC coverage for the RTSJ in [11]. Though not for SCJ, this work uses properties specified in JML for the test case generation and as SCJ is based on RTSJ, the main ideas can also be applied to SCJ.

Certifiable Garbage Collector

One of the main problems faced while developing both infrastructure code and applications is that of having to know the allocation context of objects. Trying to keep track of this information quickly becomes cumbersome and error prone and leads to long cycles of testing and debugging. Java’s GC is one of its main strengths but is also its weak point for hard real-time systems due to the risk of interfering tasks with stringent time constraints.

Work has been done to develop garbage collectors suitable for hard-real time systems and while it can be proved that a GC task can be scheduled on hard-real time systems given that allocation rates of the mutator task are known (see e.g., [119]), safety-critical systems will need a *certified* GC, i.e., a GC that can be proved in a formal way to be correct. Efforts towards a certified GC [55, 84, 86, 129] have focused on correctly specifying the collector-mutator interface in order to avoid implementers either on the collector or mutator side to violate intended invariants [83].

Certifying a garbage collector is a big task but can be a one time investment but the benefits are worth the effort. With a GC, development of software becomes easier and opens the possibility to reuse existing libraries and components without major changes.

Bibliography

- [1] Handbook for Object-Oriented Technology in Aviation (OOTiA). Technical Report Volume 1, Federal Aviation Administration, October 2004.
- [2] Handbook for Object-Oriented Technology in Aviation (OOTiA). Technical Report Volume 2, Federal Aviation Administration, October 2004.
- [3] Handbook for Object-Oriented Technology in Aviation (OOTiA). Technical Report Volume 3, Federal Aviation Administration, October 2004.
- [4] Handbook for Object-Oriented Technology in Aviation (OOTiA). Technical Report Volume 4, Federal Aviation Administration, October 2004.
- [5] HIJA Safety Critical Java Proposal. Technical report, May 2006.
- [6] DO-333 Formal Methods Supplement to DO-178C and DO-278A. Technical report, RTCA & EUROCAE, December 2011.
- [7] Ajile Systems. <http://www.ajile.com/index.php>, January 2014.
- [8] JamVM. <http://jamvm.sourceforge.net/>, January 2014.
- [9] The picoJava-II Source Readme. <http://www1.pldworld.com/@xilinx/html/pds/HDL/picoJava-II/README.html>, June 2014.
- [10] Xenomai: Real-Time Framework for Linux. <http://www.xenomai.org/>, January 2014.
- [11] Wolfgang Ahrendt, Wojciech Mostowski, and Gabriele Paganelli. Real-time java api specifications for high coverage test generation. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 145–154, 2012.

- [12] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and W. Scott Harrison. The MILS architecture for high-assurance embedded systems. *International journal of embedded systems*, 2(3):239–247, 2006.
- [13] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *In Proc. of FORMATS’03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.
- [14] Jeppe L. Andersen, Mikkel Todberg, Andreas E. Dalsgaard, and René Rydhof Hansen. Worst-case memory consumption analysis for SCJ. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 2–10. ACM Press, 2013.
- [15] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7(1):1–49, 2007.
- [16] Motor Industry Software Reliability Association. *MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems*. 2013.
- [17] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, page 249–258. ACM, 2007.
- [18] T.P. Baker and A. Shaw. The cyclic executive model and ada. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 120–129, Dec 1988.
- [19] Edward G. Benowitz and Albert F. Niessner. A patterns catalog for RTSJ software designs. In *On the Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, page 497–507. Springer, 2003.
- [20] Peter Bishop and Robin Bloomfield. A Methodology for Safety Case Development. In *Proceedings of the Sixth Safety-Critical Systems Symposium*, Birmingham, 1998. Springer.
- [21] Joshua Bloch. *Effective Java*. Addison-Wesley, Upper Saddle River, NJ, 2008.
- [22] Robin Bloomfield and Peter Bishop. Safety and assurance cases: Past, present and possible future—an Adelard perspective. In *Making Systems Safer*, page 51–67. Springer, 2010.

- [23] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. A predictable Java profile: rationale and implementations. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, page 150–159. ACM, 2009.
- [24] G. Bollella. *The real-time specification for Java*. Addison-Wesley, 2000.
- [25] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real time specification for Java. In *Companion of the 18th annual ACM SIG-PLAN conference on Object-oriented programming, systems, languages, and applications*, page 361–369. ACM, 2003.
- [26] Greg Bollella, James Gosling, Benjamin M. Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. The Real-Time Specification for Java 1.0.2. Technical report, 2006.
- [27] Eric J Bruno and Greg Bollella. *Real-time Java programming with Java RTS*. Prentice Hall, Upper Saddle River, NJ, 2009.
- [28] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *ACM SIGAda Ada Letters*, 24(2):1–74, 2004.
- [29] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, 2009.
- [30] Giorgio C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Real-time systems series. Springer, New York, 3rd ed edition, 2011.
- [31] Lisa J. Carnahan and M. Ruark. Requirements for Real-Time Extensions to the Java Platform: Report From the Requirements Group for Real-Time Extensions for the Java Platform. Technical report, National Institute of Standards and Technology, September 1999.
- [32] Adele-Louise Carter. Safety-critical versus security-critical software. In *System Safety 2010, 5th IET International Conference on*, page 1–6, 2010.
- [33] J. E. Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.
- [34] Angelo Corsaro and Corrado Santoro. Design patterns for RTSJ application development. In *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, page 394–405. Springer, 2004.

- [35] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, page 90–100. IEEE, 2002.
- [36] W.J. Cullyer and B.A. Wickmann. The choice of computer languages for use in safety-critical systems. *Software Engineering Journal*, 6(2):51–58, March 1991.
- [37] Andreas E. Dalsgaard, René Rydhof Hansen, and Martin Schoeberl. Private memory allocation analysis for safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 9–17, 2012.
- [38] Jean-Marie Dautelle. Javolution. <http://javolution.org/>, September 2012.
- [39] M.H. Dawson. Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 241–247, May 2008.
- [40] Michael Dawson. Real-time Java, Part 6: Simplifying real-time Java development. <http://www.ibm.com/developerworks/java/library/j-rtj6/>, July 2007. Last accessed on July 6, 2012.
- [41] Peter Dibble. No-Heap Safe Classes, August 2004.
- [42] Peter Dibble. *Real-time Java platform programming*. Sun Microsystems Press, Santa Clara, Calif., 2008.
- [43] Brian P. Doherty. A real-time benchmark for Java. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 35–46. ACM, 2007.
- [44] Daniel L. Dvorak and William K. Reinholtz. Hard real-time: C++ versus RTSJ. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, page 268–274. ACM, 2004.
- [45] Michael R. Elliott and Peter Heller. Object-oriented software considerations in airborne systems and equipment certification. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, page 85–96. ACM, 2010.

- [46] Huáscar Espinoza, Alejandra Ruiz, Mehrdad Sabetzadeh, and Paolo Panaroni. Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In *Software Certification (WoSoCER), 2011 First International Workshop on*, page 1–6, 2011.
- [47] David Flanagan. *Java in a nutshell*. O'Reilly, Beijing, 5th edition, March 2005.
- [48] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, October 1995.
- [49] Ghaith Haddad. *Specification and Runtime Checking of Timing Constraints in Safety Critical Java*. PhD thesis, University of Central Florida, 2012.
- [50] Wolfgang A. Halang and Alexander D. Stoyenko. Comparative evaluation of high-level real-time programming languages. *Real-Time Systems*, 2(4):365–382, 1990.
- [51] Wolfgang A. Halang and Janusz Zalewski. Programming languages for use in safety-related applications. *Annual Reviews in Control*, 27(1):39–45, January 2003.
- [52] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. Toward libraries for real-time java. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 458–462, May 2008.
- [53] Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A Modular Worst-case Execution Time Analysis Tool for Java Processors. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 47–57. IEEE, April 2008.
- [54] Joel Hasbrouck and Gideon Saar. Low-Latency Trading. Technical Report No. 35-2010, Chicago, May 2013.
- [55] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In *ACM SIGPLAN Notices*, volume 44, page 441–453. ACM, 2009.
- [56] M. Hecht, S. Graff, SoHaR Incorporated, W. Green, H. Hecht, U. S. Nuclear Regulatory Commission Office of Nuclear Regulatory Research Division of Systems Technology, S. Koch, D. Lin, A. Tai, and D. Wendelboe. *Review Guidelines for Software Languages for Use in Nuclear Power Plant Safety Systems: Final Report*. U.S. Nuclear Regulatory Commission, 1996.

- [57] M. Teresa Higuera-Toledano. About 15 years of real-time java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 34–43, 2012.
- [58] M. Teresa Higuera-Toledano and Miguel A. de Miguel-Cabello. Dynamic detection of access errors and illegal references in RTSJ. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, RTAS '02, Washington, DC, USA, 2002. IEEE Computer Society.
- [59] M.T. Higuera-Toledano. Hardware-based solution detecting illegal references in real-time Java. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 229–337, July 2003.
- [60] EY-S. Hu, Guillem Bernat, and Andy Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. In *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, page 109–116. IEEE, 2002.
- [61] IEEE. *IEEE standard glossary of software engineering terminology*. Institute of Electrical and Electronics Engineers, New York, N.Y, 1990.
- [62] International J Consortium Specification. Real-Time Core Extensions, Draft 1.0.14. Technical report, September 2000.
- [63] ISO. *ISO/IEC TR 15942:2000 Programming languages Guide for the Use of the Ada Programming Language in High Integrity Systems*. ISO, 2000.
- [64] Prashant Jain and Michael Kircher. Pattern Oriented Software Architecture: Patterns for Resource Management. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, page 41–41, 2007.
- [65] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CDx: a family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, page 41–50. ACM, 2009.
- [66] John C. Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, page 547–550, 2002.
- [67] Stephan E. Korsholm. HVM lean Java for small devices, 2014.
- [68] Jagun Kwon. *Ravenscar-Java: Java Technology for High-Integrity Real-Time Systems*. PhD thesis, University of York, 2006.

- [69] Jagun Kwon and Andy Wellings. Memory management based on method invocation in RTSJ. In *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, page 333–345. Springer, 2004.
- [70] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: a high-integrity profile for real-time Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):681–713, April 2005.
- [71] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley, Reading, Mass., 1999.
- [72] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, and Werner Dietl. JML reference manual, 2008.
- [73] J. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. pages 166–171, December 1989.
- [74] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 1 edition, February 2013.
- [75] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
- [76] Jane W. S. Liu. *Real-Time systems*. Prentice Hall, Upper Saddle River, NJ, 2000.
- [77] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Hentties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. *Safety-Critical Java Technology Specification, Public draft*. 2013. v 0.94.
- [78] P. Mader, P.L. Jones, Y. Zhang, and J. Cleland-Huang. Strategic Traceability for Safety-Critical Projects. *Software, IEEE*, 30(3):58–66, June 2013.
- [79] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.
- [80] Rajib Mall. *Real-time systems theory and practice*. Dorling Kindersley, New Delhi, India, 2008.
- [81] Chris Marriott and Ana Cavalcanti. Scj: Memory-safety checking without annotations. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 465–480. Springer International Publishing, 2014.

- [82] James Mc Enery, David Hickey, and Menouer Boubekour. Empirical evaluation of two main-stream RTSJ implementations. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 47–54. ACM, 2007.
- [83] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ACM Sigplan Notices*, volume 45, page 273–284. ACM, 2010.
- [84] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. *ACM SIGPLAN Notices*, 42(6):468–479, 2007.
- [85] Y. Moy, E. Ledinot, H. Delseny, Vi. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *Software, IEEE*, 30(3):50–57, June 2013.
- [86] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, Computer Laboratory, Trinity College, 2008.
- [87] R. Nelson. Certification Processes for Safety-Critical and Mission- Critical Aerospace Software. Technical report, NASA, 2003.
- [88] Kelvin Nilsen. Issues in the design and implementation of real-time Java. *Java Developer’s Journal*, 1(1):44, 1996.
- [89] Kelvin Nilsen. A type system to assure scope safety within safety-critical Java modules. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, page 97–106. ACM, 2006.
- [90] Kelvin Nilsen. Revisiting the perc real-time API. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 165–174, 2012.
- [91] Linda Northrop, Peter H Feiler, Bill Pollak, and Daniel Pipitone. *Ultra-large-scale systems: the software challenge of the future*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 2006.
- [92] Jon Ogborn, Simon Collins, and Mick Brown. Randomness at the root of things 2: Poisson sequences. *Physics Education*, 38(5):398, 2003.
- [93] Marcel Vinicius Medeiros Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York, 2005.
- [94] Filip Pizlo, Jason M. Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, page 101–110. IEEE, 2004.

- [95] Ales Plsek. *SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2009.
- [96] Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera, and Jan Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, page 95–101. ACM, 2010.
- [97] Alex Potanin, James Noble, Tian Zhao, and Jan Vitek. A high integrity profile for memory safe programming in real-time Java. In *The 3rd workshop on Java Technologies for Real-time and Embedded Systems, San Diego, CA, USA*, 2005.
- [98] P. Puschner and A. Wellings. A profile for high-integrity real-time Java programs. In *Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on*, pages 15–22. IEEE Comput. Soc, 2001.
- [99] Anders P. Ravn and Martin Schoeberl. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience*, 24(8):772–788, 2012.
- [100] Anders P. Ravn and Hans Søndergaard. A test suite for safety-critical Java using JML. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '13*, pages 80–88. ACM Press, 2013.
- [101] Jorge Real and Alfons Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.
- [102] Leanna Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178c compliance*. CRC Press/Taylor & Francis Group, Boca Raton, 2013.
- [103] Juan Rios and M. Schoeberl. Hardware Support for Safety-Critical Java Scope Checks. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 31–38, April 2012.
- [104] Juan Rios and M. Schoeberl. An Evaluation of Safety-Critical Java on a Java Processor. In *Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), 2014 IEEE/IFIP 10th Workshop on*, pages 276 – 283, June 2014.

- [105] Juan Rios and M. Schoeberl. Reusable Libraries for Safety-Critical Java. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 188–197, June 2014.
- [106] Juan Ricardo Rios, Kelvin Nilsen, and Martin Schoeberl. Patterns for safety-critical Java memory usage. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 1–8, 2012.
- [107] RTCA. DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA, 1992.
- [108] RTCA. DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA, 2011.
- [109] John Rushby. New challenges in certification for aircraft software. In *Proceedings of the ninth ACM international conference on Embedded software*, page 211–218, 2011.
- [110] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, page 93–100. IEEE, 2004.
- [111] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Technische Universität Wien, 2005.
- [112] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Virtualbookworm, Alpha Edition edition, October 2007.
- [113] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1):265–286, 2008.
- [114] Martin Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, page 47–53, 2011.
- [115] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A Hardware Abstraction Layer in Java. *ACM Transactions on Embedded Computing Systems*, 10(4):1–40, November 2011.
- [116] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Softw. Pract. Exper.*, 40(6):507–542, 2010.

- [117] Martin Schoeberl and Juan Ricardo Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 54–61, Copenhagen, Denmark, 2012. ACM.
- [118] Martin Schoeberl, B. Thomsen, A. P. Ravn, and H. Sondergaard. A profile for safety critical java. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, page 94–101. IEEE, 2007.
- [119] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 85–93. ACM, 2007.
- [120] David John Smith and Kenneth G. L Simpson. *Safety critical systems handbook a straightforward guide to functional safety, IEC 61508 (2010 edition) and related standards*. Butterworth-Heinemann, Oxford, 2010.
- [121] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 44–53, 2012.
- [122] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. A safety-critical java technology compatibility kit. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 1:1–1:9, New York, NY, USA, 2014. ACM.
- [123] Hans Søndergaard, Bent Thomsen, and Anders P. Ravn. A Ravenscar-Java profile implementation. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, page 38–47. ACM, 2006.
- [124] SPEC. SPEC JBB2005. <http://www.spec.org/jbb2005/>.
- [125] Daniel Tang, Ales Plsek, and Jan Vitek. Static checking of safety critical Java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, New York, NY, USA, 2010. ACM.
- [126] Daniel Tang, Ales Plsek, and Jan Vitek. Memory safety for safety critical java. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 235–264. Springer US, 2012.
- [127] TimeSys. RTSJ Reference Implementation. <http://www.timesys.com/java/>.

- [128] K.W. Tindell, A. Burns, and A.J. Wellings. Mode changes in priority preemptively scheduled systems. In *Real-Time Systems Symposium, 1992*, pages 100–109, Dec 1992.
- [129] Noah Torp-Smith, Lars Birkedal, and John C. Reynolds. Local reasoning about a copying garbage collector. *ACM Trans. Program. Lang. Syst.*, 30(4):1–58, 2008.
- [130] U.S. DoD. Department of Defense Requirements for High Order Computer Programming Languages "STEELMAN", 1978.
- [131] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.
- [132] Andy Wellings and MinSeong Kim. Asynchronous event handling and safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 53–62, Prague, Czech Republic, 2010. ACM.
- [133] Andy Wellings, Matt Luckcuck, and Ana Cavalcanti. Safety-critical java level 2: Motivations, example applications and issues. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '13*, pages 48–57, New York, NY, USA, 2013. ACM.
- [134] B. A. Wichmann. Requirements for programming languages in safety and security software standards. *Computer standards & interfaces*, 14(5):433–441, 1992.
- [135] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CDx. Technical report, Tech. Rep., University of York, Department of Computer Science, York, UK, 2012.
- [136] Lei Zhao, Daniel Tang, and Jan Vitek. A technology compatibility kit for safety critical Java. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 160–168. ACM, 2009.
- [137] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, page 241–251. IEEE, 2004.