Assessment of the Java Programming Language for Use in High Integrity Systems*

Jagun Kwon, Andy Wellings, and Steve King
Department of Computer Science
University of York, UK
{jagun, andy, king}@cs.york.ac.uk

ABSTRACT

This paper sets a goal of investigating the use of Java in the development of high integrity systems. Important requirements of programming languages for the development of high integrity software are first surveyed. Based on these, we develop several criteria that are used for the following assessment of Java. A summary of the assessment is provided before we go on to review a few existing subsets of the language.

1. Introduction

High integrity systems are generally complex and crucial systems that come into their existence as we realise new problem domains and build some form of system in an attempt to protect or benefit related objects and human beings. Examples include space shuttles, nuclear power plants and medical instruments, and they typically have high development and maintenance costs due to the customised nature of their components. There exist many general and sector-specific standards produced to assist in building such important systems.

Within high integrity systems, there has been a growing trend to use software, because it provides [Leveson1986, Leveson1991, Parnas+1990, Bowen+1998]

- improved functionality
- increased flexibility in design and implementation
- reduced production cost

* This work has been funded by the EPSRC under award number GR/M94113.

• enhanced management of complexity in application areas.

Over the recent years, Java has proved to be an appropriate vehicle for a diverse range of applications including web based intranets and embedded systems. Its relatively simple linguistic semantics, the adoption of well-understood approaches to managing software complexity, and support for concurrency seem to have contributed towards its popularity. Initially designed with embedded systems in mind, Java's main goal was to provide engineers with a reliable and cost-effective platform-independent environment. The burden of learning a new language is kept to the minimum for existing C and C++ programmers, while helping them to discover errors earlier by means of strong type checking, array-bound checking, null-pointer checking, and so on [Gosling+2000]. Further, its support for concurrency, i.e. multi-threading and synchronisation mechanisms, together with the use of portable code or the *bytecode* opens up a huge number of possibilities for many other applications, including high integrity systems.

However, despite all these valuable features, Java has been criticised for its unpredictable performance as well as some security concerns [Appel1999, Azevedo+1999, Amme+2001]. The automatic garbage collection and dynamic class loading mechanisms are often considered problematic, especially under time or performance-critical situations. Moreover, a number of security bugs in the Java virtual machine have been discovered since its first appearance, especially in the bytecode verifiers and Just-in-Time (JIT) compilers [Gong1999, Appel1999]. These fears make Java and its associated technology simply unsuitable for the development of high integrity systems.

Upon the realisation of such primary drawbacks of Java, many researchers and scientists have attempted to improve the situation, particularly in search of predictable real-time performance. For instance, *the Real-Time Specification for Java* [Bollella+2000b, Bollella+2000a] and its reference implementations (e.g. [TimeSys2002]) have proved that Java can be a capable framework for concurrent real-time applications. The specification attempts to minimise any modification to the original language semantics and defines many additional classes that should be implemented in a supporting virtual machine. This, however, ironically leads to a language and run-time system that are complex to implement and have high

overheads at run-time. Software produced in that framework is also difficult to analyse with all the luxurious features, such as the asynchronous transfer of control (ATC) and dynamic class loader.

Bearing in mind the positive developments and drawbacks of Java, this paper investigates the use of the language in the development of high integrity systems. We first survey important requirements of programming language for the development of high integrity software. Based on those requirements gathered from several relevant standards and guidelines, we develop 23 criteria that are used for the following assessment of Java. A summary of the assessment is provided before we move on to review a few existing subsets of the language.

2. Requirements of Programming Language

A study by Bentley [Bentley1999] summarises some of the well-known requirements of programming language for the development of high integrity systems including works by [Carré+1990], [Cullyer+1991], [USDoD1978], [USDoD1990] and [Hutcheon+1992]. It carries out an assessment on Java against all the requirements, producing a series of comprehensive rationales. A subset of the language is also proposed, but only sequential features are included.

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

2.1. Summary of the requirements used by Bentley [Bentley1999]

• Carré et al [Carré+1990] identify six factors that can have an influence on a programming language's suitability for use in high integrity systems. These factors are summarised by [Storey1996] as

- *Logical soundness*: is there a sound, unambiguous definition of the language?
- *Simplicity of definition*: are there simple, formal definitions of the various language features? Complexity in these definitions results in complexity within compilers and other support tools, which can lead to errors.

- *Expressive power*: can program features be expressed easily and efficiently?
- *Security and integrity*: can violations of the language definitions be detected before execution?
- *Verifiability*: does the language support verification, that is, proving that the code produced is consistent with its specification?
- *Bounded space and time requirements*: can it be shown that time and memory constraints will not be exceeded?

• Cullyer et al [Cullyer+1991] define a checklist of eleven factors to help establish if a language has appropriate characteristics. The factors or questions to ask are

- *Wild jumps*: can it be shown that the program cannot jump to an arbitrary memory location?
- *Overwrites*: are there language features that prevent an arbitrary memory location being overwritten?
- *Semantics*: are the semantics of the language defined sufficiently for the translation process needed for static code analysis?
- *Model of maths*: is there is a rigorous model of both integer and floating point arithmetic?
- *Operational arithmetic*: are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?
- *Data typing*: are the means of data typing strong enough to prevent misuse of variables?
- *Exception handling*: if the software detects a malfunction at runtime, do mechanisms exist to facilitate recovery?
- *Safe subsets*: does a subset of the language exist which is defined to have properties that satisfy these requirements more adequately than the full language?
- *Exhaustion of memory*: are there facilities to guard against running out of memory at runtime?
- *Separate compilation*: are facilities available for separate compilation of modules, with type checking across the module boundaries?
- *Well understood*: will the designers and programmers understand the language sufficiently to write safety-critical software?

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

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

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

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

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

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

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

• The work by York Software Engineering, British Aerospace and the U.K. Ministry of Defence [Hutcheon+1992] is specific to safety-critical applications with emphasis on the military requirements of the INTERIM Defence Standard 00-55 [UKMoD1991]. Two levels of requirements are defined (i.e., one to represent mandatory and the other optional features that a language should have) and subsequently used to assess Ada9X in [Hutcheon+1992]. The level one, mandatory requirements are

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

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

2.2. Additional Requirements

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

• The Ada95 Trustworthiness Study [Craigen+1995, Saaltink+1996, Saaltink+1997] is broad and analytical in that it defines a concrete framework for language analysis based on important standards, evaluates each language feature of Ada95 against the framework, and produces comprehensive guidance for the use of Ada95 in the development of high integrity systems. It first identifies four main themes for analysis, which are *predictability*, *analysability*, *traceability*, and *engineering*, and these themes lead to the development of ten analytical categories and ratings in each category, as shown below.

Categories	Ratings	Examples
Run-time Support Needed	1. No or minor run-time support. 2. Some run-time support. 3. Significant run-time support.	1. Scalar types 2. Simple tasking 3. Asynchronous Transfer of Control
Functional Predictability	1. Exact, which gives only one outcome. 2. Bounded, which gives only a small set of possible outcomes, which could be a few possible results or results within a small range. 3. Unpredictable, for all other cases.	1. Discriminants 2. Access types 3. Generalized access types
Timing Predictability	1. Tightly bounded, where the time-to-execute can be expressed in terms of a formula over the data, number of iterations, etc. 2. Loosely bounded, where a maximum time to execute can be determined, but actual execution times are usually much better. 3. Unpredictable, where we do not know how to predict the time bound.	1. Type conversion of numerated types 2. Arrays 3. Case statements
Space Usage Predictability	1. Exact, where we can develop a formula to determine exact memory usage. This requires implementation information on use of temporaries, stack, etc. 2. Worst-case analysis, where we can bound the space used, both immediately and over time. 3. Unpredictable.	1. Loop statements 2. Subprogram declarations 3. Task units and task objects
Formal Definition	1. Existing definition (for Ada83 and no changes to Ada95). 2. Potentially definable, where a definition exists in Ada83 but changes mean a re-definition is needed, or definitions exist in other languages. 3. Unknown.	1. Exception handlers 2. Static expressions and subtypes 3. Return statement for functions
Integrity and Security Issues	1. Enhances, for syntax and language rules that forbid or guard against violations. 2. Neutral. 3. Hinders, for a construct that facilitates the violation of integrity or access protections.	1. Package specifications and declarations 2. Object renaming declarations 3. Abort of a task
Reliability and Engineering Support	1. Enhances reliability (with explanation). 2. Neutral. 3. Problematic (with explanation).	1. Membership tests 2. Signed integer types 3. Derived types and classes
Robustness	1. Enhances (contributes to robustness). 2. Neutral (no effect on robustness). 3. Hinders (deleteriously affects robustness).	1. Predefined exceptions and language-defined checks

		2. Concatenate operator 3. Task and entry attributes
Static Analysis	1. Tractable analysis. 2. Hard/Intractable analysis. 3. Unknown.	1. Enumeration representation clauses 2. Generic instantiation – subprograms 3. Unchecked type conversions
Dynamic Analysis	1. Tractable analysis. 2. Hard/Intractable analysis. 3. Unknown.	1. Static expressions and subtypes 2. Dispatching subprograms 3. Generic formal objects

Figure 1. Ten analytical categories and ratings of the Ada95 Trustworthiness Study [Craigen+1995]

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

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

Figure 2. Verification Techniques employed in the assessment of Ada95 [ISO/IEC DTR 15942]

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

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

· have available validated compilers.

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

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

Top-level attributes	**Intermediate attributes**	**Base attributes**
Reliability	1. Predictability of memory utilisation	· Minimising dynamic memory allocation · Minimising memory paging and swapping
	2. Predictability of control flow	· Maximising structure · Minimising control flow complexity · Initialising variables before use · Single entry and exit points for subprograms · Minimising interface ambiguities · Use of data typing · Accounting for precision and accuracy · Order or precedence of arithmetic, logical, and functional operators · Avoiding functions or procedures with side effects · Separating assignment from evaluation · Proper handling of program instrumentation · Controlling class library size · Minimising use of dynamic binding · Controlling operator overloading
	3. Predictability of timing	· Minimising the use of tasking · Minimising the use of interrupt driven processing
Robustness	1. Controlling use of diversity	· Controlling internal diversity · Controlling external diversity
	2. Controlling use of exception handling	· Handling of exceptions locally · Preserving external control flow · Handling of exceptions uniformly

	3. Checking input and output	· Input data checking · Output data checking
Traceability	1. Readability	See Maintainability
	2. Controlling use of built-in functions	None
	3. Controlling use of compiled libraries	None
Maintainability	1. Readability	· Conforming to indentation guidelines · Using descriptive identifier names · Commenting and internal documentation · Limiting subprogram size · Minimising mixed language programming · Minimising obscure or subtle programming constructs · Minimising dispersion or related elements · Minimising use of literals
	2. Data abstraction	· Minimising the use of global variables · Minimising the complexity of the interface defining allowable operations
	3. Functional cohesiveness	· Single purpose function and procedures · Single purpose variables
	4. Malleability	· Isolation of alterable functions
	5. Portability	· Minimising the use of built-in functions · Minimising the use of compiled libraries · Minimising dynamic binding · Minimising tasking · Minimising asynchronous constructs (interrupts) · Isolation of non-standard constructs

Figure 3. Generic Safe Programming Attributes [NUREG/CR-6463]

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

· formally defined syntax and semantics

· a formal means of relating the code to the formal design

· block structured

· strongly typed

· run-time type and array bound checking

· conformance to an international standard

· use of a validated compiler

· well-understood

· exception handling

· extensive tool support, and tools that are trusted/validated.

The reports also contain some guidelines derived from relevant literature, such as [Cullyer+1991] and [Carré+1990], and these largely overlap other requirements described previously. The use of safer subsets is also emphasised.

3. Assessment Criteria

Now, since some of the requirements introduced above are redundant and ambiguous we inclusively categorise them into relevant assessment criteria along with appropriate references. However, it is important to note that this collection of criteria is neither complete[1] nor authoritative, but it attempts to amalgamate many different requirements into a balanced and informative framework for the assessment of programming languages. As in [Hutcheon+1992] we propose two levels of criteria, namely *Mandatory requirements* (Level 1) and *Desirable requirements* (Level 2).

3.1. Level 1 – Mandatory requirements

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

L1.1. Syntactical / Semantic Requirements

L.1.1.1	**Type safety / Strong typing rules**
References	[USDoD1978], [Cullyer+1991], [USDoD1990], [Hutcheon+1992], [Craigen+1995], [ISO/IEC DTR 15942], [NUREG/CR-6463], [MISRA1995f]
Rationale	Strongly typed languages help reduce errors in programs at compile-time. Moreover, type safety is often considered to be sufficient for ensuring the minimum nontrivial level of program safety, i.e. control

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

	flow safety, memory safety, and stack safety [Kozen1999]. Thus it is strongly encouraged to use a type safe or strongly typed language, enhancing the integrity and security of software.
Specifics	Implicit type conversions must not be allowed. All data types should be statically analysable before program execution. Explicit type conversion rules should be clearly stated in the language standard or definition. There should be some ways to avoid access types or pointers.
Ratings	1. Strongly typed / Statically analysable. 2. Strongly typed, but some types are analysable only at run-time, mainly due to the use of polymorphism in the language. 3. Not strongly typed and implicit type conversions are allowed.

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

L.1.1.3	**Modularity / Structures**
References	[USDoD1978], [Cullyer+1991], [Hutcheon+1992], [Craigen+1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	It must be straightforward to code and maintain programs in a high integrity programming language, so that the complexity of software becomes manageable. This is often achieved by means of visibility control (or scopes), functions, and objects in many modern languages, in which the integrity and security of software are generally improved.
Specifics	There should be sound mechanisms to structure and modularise program code both syntactically (in some form of determinable blocks or scopes) and semantically with clear interfaces. There should be no wild/unbounded jumps between different modules. Separate compilation of modules should be possible.
Ratings	1. The language provides rich and precise means of structuring programs, and programs can be maintained in terms of modules or objects. 2. Such mechanisms are provided, but not cost-effective or efficient.

	3. There is no reasonable approach.

L.1.1.4	**Formal semantics / International standards**
References	[USDoD1978], [Cullyer+1991], [USDoD1990], [Hutcheon+1992], [Craigen+1995], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	A standardised language benefits the development of compilers and tools, and user training. Verification techniques can also be applied to a language with formally defined semantics.
Specifics	There should be a (international) standard definition of the language. There should be formally defined semantics of the language, or at least a subset of the language.
Ratings	1. An internationally standardised formal definition exists. 2. The language or high integrity subset of it can be formally defined. 3. Unknown.

L.1.1.5	**Well-understood**
References	[Cullyer+1991], [USDoD1978], [Hutcheon+1992], [USDoD1990], [MISRA1995f]
Rationale	A language with well-understood semantics and syntaxes will help to produce quality software, often cost-effectively.
Specifics	The language should be simple, well understood, easy to adopt, and easy to implement.
Ratings	1. The language is well understood, and there are many trained developers and designers. 2. The language is well understood only by a limited number of people. 3. Unknown.

L.1.1.6	**Support for domain specific or embedded applications**
References	[USDoD1978], [Hutcheon+1992]
Rationale	High integrity systems are often embedded systems that need to interface or control physical resources or (non-standard) peripheral devices. Therefore, a programming language designed with such applications in mind should be used.
Specifics	Robust mechanisms for controlling memory, I/O devices or other hardware are required.
Ratings	1. The language naturally supports embedded applications. 2. There is a limited support, but external libraries or language extensions can be utilised. 3. No support provided or Unknown.

L.1.1.7	**Concurrency / Parallel processing**
References	[USDoD1978], [Hutcheon+1992]
Rationale	Although concurrency is one of the main sources of complication in program analysis and verification (classified as only a desirable – not mandatory - feature in [Hutcheon+1992]), it is invaluable in modelling

	or capturing real-world problems. Thus, we believe this has to be an essential requirement for modern high integrity language.
Specifics	The following features should be included: Language-level support for multitasking or multithreading. Control over scheduling policy. Straightforward communication and synchronisation mechanism(s), plus facility to bound blocking.
Ratings	1. All the above specifics are satisfied. 2. Only limited support is provided at the language-level, but external libraries or run-time systems can be utilised. 3. No reasonable support provided or Unknown.

L1.2. Application of verification techniques / Predictability

L.1.2.1	**Functional predictability**
References	[Hutcheon+1992], [Craigen+1995], [ISO/IEC DTR 15942], [NUREG/CR-6463], [MISRA1995f]
Rationale	High integrity software must be proven to be predictable in terms of its functional behaviours.
Specifics	All or most of the following analysis techniques should be applicable. Control flow analysis Data flow analysis Information flow analysis Symbolic execution Formal code verification
Ratings	1. All techniques in the above specifics or feasible alternatives can be utilised. 2. Not all techniques can be utilised due to the complex features of the language, but sub-setting the language may improve such analyses. 3. Unknown or there is no cost-effective way of utilising such analysis techniques.

L.1.2.2	**Temporal predictability / Timing analysis**
References	[Hutcheon+1992], [Craigen+1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	In addition to the functional predictability, timely behaviours of such software and systems must also be guaranteed.
Specifics	Worst Case Execution Time (WCET) of each process must be obtainable, so that schedulibility analysis can be performed.
Ratings	1. Tightly bounded execution time(s) can be obtained. 2. Loosely bounded execution time(s) can be obtained. 3. Unpredictable or there is no known way to obtain WCET.

L.1.2.3	**Resource usage analysis**
References	[Cullyer+1991], [Hutcheon+1992], [Craigen+1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	It is important to identify what resources are needed and how they are

	utilised, so that errors such as stack overflow may not occur, and system implementations may be kept economical.
Specifics	The following properties should be analysable. Memory (or heap) usage Stack usage Any other resources to be utilised in the application area.
Ratings	1. Exact prediction of the above specifics is possible. 2. Worst-case analysis is possible, but not practical. 3. Unpredictable.

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

L.1.3.1	**Certified language translators / Run-time environments**
References	[USDoD1978], [Hutcheon+1992], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	There must be a high level of assurance in language processors, especially compilers.
Specifics	A formally certified compiler by an authoritative or trusted body should be used. Low-level code should be traceable in accordance with source code. Run-time environments should also be certified if used.
Ratings	1. There exist one or more certified language translators, and they are formally proven to be flawless. 2. Language translators may contain several known errors or malfunctions that are well documented, but they will not affect the development of high integrity software. 3. Unknown.

L.1.3.2	**Run-time support / Environment issues**
References	[USDoD1978], [Hutcheon+1992], [USDoD1990], [Craigen+1995], [ISO/IEC DTR 15942], [NUREG/CR-6463]
Rationale	Libraries (or any additional code) or run-time support may make it complex to perform some analyses, such as WCET and control flow analyses. Hence, all such additional code should be predictable and analysable in terms of safety and timeliness. Minimising implementation dependencies is also encouraged.
Specifics	All the behaviours of additional code should be well understood. All timing information of the underlying run-time system and libraries should be known and accurate.
Ratings	1. There exists concrete information on the functional and temporal behaviours of all libraries and run-time system. 2. Only worst-case analysis is possible. 3. Unknown.

3.2. Level 2 – Desirable Requirements

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

L2.1. Syntactical / Semantic Requirements

L.2.1.1	**Exception Handling / Failure behaviour**
References	[USDoD1978], [Cullyer+1991], [Hutcheon+1992], [Craigen+1995], [NUREG/CR-6463], [MISRA1995f]
Rationale	Handling errors while a high integrity system is operating is sometimes seen as undesirable on account of additional overheads and unpredictable behaviours. However, if any sort of error can occur, then the system should gracefully degrade, or recover after some corrections.
Specifics	Robust and analysable run-time error detection and handling mechanism should exist. Failure behaviours should be programmable.

L.2.1.2	**Model of Mathematics**
References	[Cullyer+1991], [USDoD1978]
Rationale	As often required in some high integrity systems, the language should have a rigorous model of maths defined in the language standard.
Specifics	A model of both integer and floating point arithmetic should be defined within the language standard. Procedures for checking if operational arithmetic at run-time is correct should exist.

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

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

L.2.1.5	**Coding style guidelines**
References	[Hutcheon+1992], [NUREG/CR-6463], [MISRA1995f]
Rationale	Coding style guidelines may help reduce the gap between well-established Software Engineering principles and the actual practice of programming in a particular language.
Specifics	None.

L.2.1.6	**Support for abstraction and information hiding**
References	[ISO/IEC DTR 15942], [Hutcheon+1992], [USDoD1990], [MISRA1995f]
Rationale	Employing abstraction or information hiding techniques (e.g. object orientation) can greatly decrease software complexity. Thus they are beneficial in program design, development, and maintenance.
Specifics	None.

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

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

L.2.2.1	**Certified (static/dynamic) analysis tools**
References	[Hutcheon+1992], [ISO/IEC DTR 15942], [MISRA1995f]
Rationale	In order to gain more confidence in high integrity software it is imperative to use certified analysis tools, which may check for errors, such as, race conditions and deadlocks.
Specifics	None.

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

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

L.2.2.4	**Code portability**
References	[Hutcheon+1992], [NUREG/CR-6463]
Rationale	Since there exists a diverse range of code-executing platforms, it is often considered beneficial to have a portable program representation, so that all necessary analyses may be applied once for all.
Specifics	None.

4. Assessment of the Java programming language

Now, Java is assessed against the criteria developed in the previous section. A summary is provided at the end.

4.1. Assessment of Java against Level 1

L.1.1. Syntactical / Semantic Requirements

L.1.1.1. Type safety / Strong typing rules

Java is a strongly typed language. For all primitive types, implicit type conversions are not allowed (all possible conversions are stated in the language specification), and programs are analysable before running them. But for dynamic reference types, it is not always straightforward to statically analyse code, but is generally possible only at run-time because of the use of, for example, inherited interfaces and local classes within different scopes.

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

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

Side effects can occur in Java if expressions contain embedded assignments, sub-operators, and method invocations. Many side effects, however, can be eliminated via the use of a code checker or analyser, and a subset of Java. Operator precedence levels are defined in the specification [Gosling+2000], but the large number is at times seen undesirable as it becomes more difficult for programmers to learn [Bentley1999, USDoD1978]. All types in Java have default initial values, but compilers issue warnings if any variables are used before initialisation. It should also be noted that some returned values of a method can be *quietly discarded* without any warning [Gosling+2000], i.e. when there is no assignment expression for a method call that returns a value.

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

L.1.1.3. Modularity / Structures

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

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

L.1.1.4. Formal semantics / International standards

There are no stable standards for Java although the language specification [Gosling+2000] serves as an informal standard for the time being. There exist some formal semantics of Java, for example, in Action semantics [Watt+2000, Brown+1999], in Denotational Semantics [Alves-Foss+1999b], and in other BNF-like notations [Alves-Foss+1999a]; most of which are based on parts of the language. Drossopoulou and Eisenbach [Drossopoulou+1999] have also defined a series of subsets of Java and proved their type soundness.

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

L.1.1.5. Well-understood

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

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

L.1.1.6. Support for domain specific or embedded applications

One of the main application areas for which Java was first developed was embedded systems. In pure Java, however, it is not possible to control underlying hardware without appropriate native methods implemented in different languages. Even then, it is still difficult to implement systems with rigorous safety and real-time requirements, thanks mostly to the overheads incurred by the garbage collection mechanism, and virtual machines *per se*. There has been much research on scheduling the garbage collector and improving the efficiency of code transformation, even though it has not proven particularly effective so far. In the recent years, the Real-Time Specification for Java [Bollella+2000a] and Real-Time Core Extensions [Jconsortium2000] have been defined, so that real-time applications will certainly benefit from reference implementations of such specifications.

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

L.1.1.7. Concurrency / Parallel processing

Java supports concurrent execution of multiple threads, as well as some key synchronisation mechanisms, for example, the monitor and synchronized blocks/methods. Programmers can also allocate a priority to threads, which nevertheless is not of any significant value, as they have no control over scheduling mechanisms implemented in the virtual machine and underlying kernel. Recently, two of the specifications for real-time Java, i.e. one from Sun Microsystems [Bollella+2000a] and the other J Consortium [JConsortium2000], state various features that real-time systems require, especially with regard to scheduling, memory management, synchronisation, time, and exceptions.

Rating: 1. All the above specifics are satisfied.

L.1.2. Application of verification techniques / Predictability

L.1.2.1. Functional predictability

Due to the recent development of sophisticated analysis algorithms and tools it is now possible, to some extent, to analyse Java programs in terms of control and data flow. Nevertheless, some complex features of Java, such as the exception handing mechanism and monitors, are still not considered, or at least are immaturely handled. Formal verification is even harder for Java as there is no complete formal semantics. However, a constant progress is made in this area, and especially *Model-checking*

technology is proving strong in the verification of Java programs. For example, the Java PathFinder 2 [Brat+2000a] developed by the NASA can detect race conditions, deadlocks, and violations of user-specified assertions.

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

L.1.2.2. Temporal predictability / Timing analysis

It is well known that with all the sometimes-superfluous features like the garbage collector and virtual machine support, it is hard to obtain tight execution-time bounds for Java threads, and such timing analyses are all dependent on eventual target architectures and base operating systems (if utilised). Some techniques, however, have been suggested (e.g. [Bate+2000, Puschner+2001b]), and the release of the specifications for real-time Java will certainly improve the current situation.

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

L.1.2.3. Resource usage analysis

On account of the presence of the background garbage collector, it is generally difficult to predict how much memory space will be in use at a given moment in time, or even deducing the worst case can become impractical (and dependent on which garbage collection algorithms are employed). However, subsets of Java or of the Real-Time Specification for Java [Bollella+2000a], such as [Puschner+2001a] in which garbage collection is excluded, will ease this sort of analysis.

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

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

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

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

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

L.1.3.2. Run-time support / Environment issues

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

Rating: 3. Unknown.

4.2. Assessment of Java against Level 2

L.2.1. Syntactical / Semantic Requirements

L.2.1.1. Exception Handling / Failure behaviour

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

L.2.1.2. Model of Mathematics

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

L.2.1.3. Support for User documentation

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

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

Subtypes and enumeration types are not supported in Java, but may possibly be emulated with additional overheads.

L.2.1.5. Coding style guidelines

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

L.2.1.6. Support for abstraction and information hiding

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

L.2.1.7. Assertion checking

There is not a specific language construct for assertion checking in Java, even though it can be modelled. There is currently a specification request for a simple assertion facility, a.k.a. JSR41, which can be found at http://www.jcp.org/jsr/detail/41.jsp.

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

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

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

L.2.2.2. Interface to other languages

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

L.2.2.3. Code optimisation

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

Optimisation will also make the complexity of compiler and tool validation more difficult.

L.2.2.4. Code portability

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

4.3. Summary of Assessment

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

Strengths

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

Weaknesses

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

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

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

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

Strengths

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

Weaknesses

- Overhead and complexity of exception handling mechanism (L.2.1.1)
- The utilisation of the standard IEEE 754 arithmetic semantics can be overhead, and no exception is generated for particular operations (L.2.1.2)
- No support for subtypes and enumeration types (L.2.1.4)
- No coding guidelines for high integrity applications (L.2.1.5)
- No assertion checking facility (L.2.1.8)
- Shortage of certified analysis tools (L.2.2.1)
- Difficulty in interfacing to programs written in other languages (L.2.2.2)
- Complexity of analysing optimised code (L.2.2.3)

5. Review of Subsets

Burns et al. [Burns+1998] suggest that a restricted programming model or profile can help produce efficient and predictable systems by removing language features with high overheads, and complex and erroneous semantics. Along these lines, there have

been a few subsets or profiles for Java suggested in the literature[2]. We review them in the following.

5.1. Sequential subset of Java by [Bentley1999]

As mentioned early in this paper, Bentley [Bentley1999] defines a subset of Java after assessing the language. The subset consists of 21 rules that are effectively derived from [Hutcheon+1992], [MISRA1998] and his assessment. All the rules are categorised into six groups, as shown below with a summary of rules for each group.

• **Rules Concerned With Verification**

Multithreading is not allowed as it may cause significant difficulties in analysing programs, due mainly to the thread synchronisation mechanisms. In addition, methods and constructors shall not be overloaded.

• **Rules Concerned With Comments**

Comments shall not be nested.

• **Rules Concerned With Predictability**

Variables or objects must be statically initialised (by constructors of appropriate classes), so that no default values are expected. All constraints, such as, those used in *for*-loops, must be static. This will greatly ease various analyses, for example, memory requirement and timing analysis. The *continue* and *break* statements shall not be used, except to terminate the cases of a switch statement, for which a break statement is required for every non-empty case clause. Plus, all switch statements should contain a final default clause. The *return* statement should only appear as the last statement of a method. Further, methods must not have any side effects and not be recursively invoked. The result of a method should never be an unconstrained array type object.

• **Rules Concerned With Constants**

[2] In fact, there are subsets of Java defined for other purposes than for use in high integrity systems. For example, in [Drossopoulou+1999] the authors define a series of subsets in order to prove the type soundness of them.

Octal constants (other than zero) shall not be used. Because numbers beginning with zero are treated as octal values in Java, it is easy to make a mistake, e.g. inserting zero before a decimal constant.

• **Rules Concerned With Identifiers**

All identifier names must be unique.

• **Rules Concerned With Operators**

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

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

5.2. Profile for high integrity Real-Time Java programs [Puschner+2001a]

Puschner and Wellings [Puschner+2001a] suggest a *Ravenscar*-like profile for the Real-Time Specification for Java [Bollella+2000a], and the following is a brief summary of each of the key areas.

• **Threading Model**

There are two execution phases, i.e. *initialisation* and *mission* phases. In the initialisation phase, all necessary threads, event handlers, and memory objects are created in a non time-critical manner. No threads will be allowed to start until the top-priority thread with *main()* method finishes its execution. In the mission phase, threads may not change their own or other thread's priority except when forced by the underlying implementation of the priority ceiling protocol. Sporadic or event-

triggered activities are implemented as event handlers, and only one handler is allowed per event. All periodic threads must be an instance of *NoHeapRealtimeThread* class and need to invoke *waitForNextPeriod* method to delay execution until the start of their next periods. Asynchronous Transfer of Control (ATC), overrun and deadline-miss handlers, and delay statements are not supported by the profile; nor is dynamic class loading during the mission phase.

• Concurrency

The *synchronized* methods and blocks are the key mechanism for mutual exclusion to shared resources in Java, and the priority ceiling protocol should be implemented in the run-time system in order to avoid deadlocks. For similar reasons, *wait*, *notify*, and *notifyall* are not supported, avoiding any queue management.

• Memory Management and Raw Memory Access

The heap-based garbage collection mechanism of Java is not supported due to its long-debated unpredictability at run-time. Instead, only immortal memory and linear-time scoped memory are supported as defined in the RTSJ. Immortal memory is used by default to create objects during the initialisation phase, but is not allowed for further object creation afterwards. In addition to this, all other memory objects must only be created in the initialisation phase. The RTSJ classes for raw memory access are also supported, so that device drivers, memory-mapped I/O, and other low-level functions can be programmed.

• Time and Clock

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

The profile is primarily focused on leaving out complex features of the RTSJ. However, little attention is paid to the Java's sequential language constructs (unlike [Bentley1999]) and object-orientation features that can be problematic in performing various static analyses.

5.3. High integrity profile by the J Consortium

A sub-committee has been formed within the Real-Time Java Working Group of the J Consortium to produce a high integrity profile based on the Real-Time Core Extensions [JConsortium2000]. The profile has not been released yet, but according to Dobbing [Dobbing2001] it will resemble the Ravenscar profile for Ada95 [Burns+1998]. It consists of four main themes: partitioning, memory management, concurrency, and error recovery, respectively. Up-coming information will be found at http://www.j-consortium.org/hip/index.shtml.

• Partitioning

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

• Memory Management

The automatic garbage collection is not supported, nor is any memory compaction mechanism. The use of general heap memory is also not allowed. There are three memory allocation strategies, which are

- stack allocation for method local objects that are automatically reclaimed
- fixed size “allocation contexts” for local objects in each thread
- global allocation at initialisation time for immortal objects.

• Concurrency

Three types of priority-based tasks are supported, namely, periodic, sporadic, and interrupt tasks. In addition to these, the profile defines a subclass of the basic *CoreTask* that must explicitly be started by another thread. All threads are created at program start-up, e.g. as part of the initialisation code for classes, and it is not allowed to declare a thread class as an inner class, so that there is no requirement for any implicit *join* interface.

Shared resources and inter-thread synchronisations are managed through *protected objects*, which rely on the underlying implementation of the Priority Ceiling Protocol.

However, no mutual exclusion locks or synchronised methods are supported in the profile as they add considerable complexity to program analyses. Further, all the asynchronous thread-to-thread operations, including *stop(), setPriority(), suspend(), resume(),* and event-driven Asynchronous Transfer of Control (ATC) mechanisms, are not permitted, nor are synchronised objects and counting semaphores.

• Error Recovery

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

Like the one proposed in [Puschner+2001a], this profile is mainly focused on sub-setting the Real-Time Core Extensions [JConsortium2000], but does not address issues on the use of problematic language constructs and object-orientation features of Java.

5.4. Formal subsets by [Drossopoulou+1999]

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

Java^{s} is a substantial subset of the Java programming language, and it includes some primitive types, interfaces, classes with instance variables and instance methods, inheritance, hiding of instance variables, overloading and overriding of instance methods, arrays, implicit pointers and the *null* value, object creation, assignment, field and array access, method call and dynamic method binding, exceptions and exception handling [Drossopoulou+1999], as shown below.

Program	::=	*Def**
Def	::=	**class** ClassId ext ClassName **impl** InterfName* {*ClassMember**}
	|	**interface** InterfId **ext** InterfName* {*InterfMember**}
ClassMember	::=	*Field* | *Method*
InterfMember	::=	*MethHeader*
Field	::=	*VarType* VarId *;*
Method	::=	*MethHeader MethBody*

MethHeader	::=	(**void** | VarType) MethId ((*VarType ParId*)*) **throws** ClassName*
MethBody	::=	{*Stmts* [**return** *Expr*]] }
Stmts	::=	(*Stmt* ;)*
Stmt	::=	**if** *Expr* **then** *Stmts* **else** *Stmts*
	|	*Var* = *Expr* | *Expr*.MethName(*Expr**) | **throw** *Expr*
	|	**try** *Stmts* (**catch** *ClassName Id Stmts*)* *finally Stmts*
	|	**try** *Stmts* (**catch** *ClassName Id Stmts*)+
Expr	::=	*Value* | *Var* | *Expr*.MethName(*Expr**)
	|	**new** ClassName() | **new** *SimpleType*([*Expr*])+ ([])* | **this**
Var	::=	Name | *Expr.*VarName | *Expr*[*Expr*]
Value	::=	*PrimValue* | *RefValue*
RefValue	::=	null
PrimValue	::=	*intValue* | *charValue* | *boolValue* | …
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | ClassName | InterfaceName
ArrayType	::=	*SimpleType*[] | *ArrayType*[]
PrimType	::=	**bool** | **char** | **int** | …

Figure 4. Java^{s} programs [Drossopoulou+1999]

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

While these subsets contain many important language constructs of Java that are often omitted in other formal subsets (e.g. exceptions), they still overlook some of Java's inherent features, such as the multithreading and synchronisation models. [Hartel+2001] surveys formal subsets and approaches aimed at improving the safety of Java programs.

6. Conclusions

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

The Java programming language and its associated environments are then assessed against the two levels of criteria, and we conclude that Java is a good general

language, yet not appropriate as a whole for the development of high integrity systems that require rigorous and predictable language features, compilation systems, and tools. However, Java may be able to qualify as a suitable vehicle in the future with the help of sub-setting the language and future developments of formal mechanisms, although none of the currently proposed subsets address all the necessary areas required for high-integrity real-time systems. There is perhaps some movement towards a standardisation through the Java 2 Platform Micro Edition (J2ME) that introduces profiles for resource constrained mobile devices. One could devise a profile for high-integrity real-time systems.

7. References

[Alves-Foss+1999a] J. Alves-Foss and D. Frincke, ***Formal Grammar for Java***, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[Alves-Foss+1999b] J. Alves-Foss and F. S. Lam, ***Dynamic Denotational Semantics of Java***, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[Amme+2001] W. Amme, N. Dalton, M. Franz, and J. Von Ronne, ***SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form***, Accepted for the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation 2001.

[Appel1999] Andrew W. Appel, ***Protection against untrusted code: The JIT compiler security hole, and what you can do about it***, http://www-106.ibm.com/developerworks/library/untrusted-code/, as of January 2001.

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

[Bate+2000] I. Bate, G. Bernat, G. Murphy, P. Puschner, ***Low-level analysis of a portable WCET analysis framework***, 6th IEEE Real-Time Computing Systems and Applications (RTCSA), 2000.

[Bentley1999] S. Bentley, ***The Utilisation of the Java Language in Safety Critical System Development***, MSc dissertation, Department of Computer Science, University of York, 1999.

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

[Bollella+2000b] G. Bollella and J. Gosling, ***The Real-Time Specification for Java***, IEEE Computer, Vol. 33, No. 6, June 2000.

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

[Brat+2000a] G. Brat, K. Havelund, S. Park, and W. Visser, ***Java PathFinder – Second Generation of a Java Model Checker***, In Proceedings of Post-CAV Workshop on Advances in Verification, Chicago, July 2000.

[Brown+1999] D. F. Brown and D. A. Watt, ***JAS: a Java Action Semantics***, in Proc. of 2nd International Workshop on Action Semantics (ed. Mosses, P.D., and Watt, D.A.), BRICS NS-99-3, University of Aarhus, Denmark, 1999.

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

[Carré+1990] B. A. Carré, T. J. Jennings, F. J. Maclennan, P. F. Farrow, and J. R. Garnsworthy, ***SPARK – The SPADE Ada Kernel***, 3rd ed, Program Validation Limited, 1990.

[Craigen+1995] D. Craigen, M. Saaltink and S. Michell, ***Ada 95 Trustworthiness Study; A Framework for Analysis***, ORA Canada, 29 November 1995.

[Cullyer+1991] W. J. Cullyer, S. J. Goodenough, and B. A. Wichmann, ***The Choice of Computer Languages for use in Safety-Critical Systems***, Software Engineering Journal, March 1991.

[Dobbing2001] B. Dobbing, ***The Ravenscar Profile for High-Integrity Java Programs?***, ACM Ada Letters, Vol. 21, Issue. 1, March 2001.

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

[Gong1999] Li Gong, ***Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation***, Addison-Wesley, 1999.

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

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

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

[ISO/IEC DTR 15942] ***Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems***, ISO/IEC DTR 15942, ISO/IEC WG9, 1999.

[JConsortium2000] J Consortium, ***International J Consortium Specification: Real-Time Core Extensions***, Revision 1.0.14, www.j-consortium.org, September 2000.

[Kozen1999] D. Kozen, ***Language Based Security***, Technical Report TR99-1751, Cornell University, 1999.

[Leveson1986] N. G. Leveson, ***Software Safety: Why, What, and How***, Computing Surveys, Vol. 18, No. 2, ACM, June 1986.

[Leveson1991] N. G. Leveson, ***Software Safety in Embedded Computer Systems***, Communications of the ACM, Vol. 34, No. 2, February 1991.

[Lindholm+1999] T. Lindholm and F. Yellin, ***The Java Virtual Machine Specification***, Second Edition, Addison-Wesley 1999.

[MISRA1994] The Motor Industry Software Reliability Association, ***Development Guidelines for Vehicle Based Software***, ISBN 0952415607, MIRA Ltd., November 1994.

[MISRA1995a] The Motor Industry Software Reliability Association, ***Report 1: Diagnostics and Integrated Vehicle Systems***, MIRA Ltd., February 1995.

[MISRA1995b] The Motor Industry Software Reliability Association, ***Report 2: Integrity***, MIRA Ltd., February 1995.

[MISRA1995c] The Motor Industry Software Reliability Association, ***Report 3: Noise, EMC and Real-Time***, MIRA Ltd., February 1995.

[MISRA1995d] The Motor Industry Software Reliability Association, ***Report 4: Software in Control Systems***, MIRA Ltd., February 1995.

[MISRA1995e] The Motor Industry Software Reliability Association, ***Report 5: Software Metrics***, MIRA Ltd., February 1995.

[MISRA1995f] The Motor Industry Software Reliability Association, ***Report 6: Verification and Validation***, MIRA Ltd., February 1995.

[MISRA1995g] The Motor Industry Software Reliability Association, ***Report 7: Subcontracting of Automotive Software***, MIRA Ltd., February 1995.

[MISRA1995h] The Motor Industry Software Reliability Association, ***Report 8: Human Factors in Software Development***, MIRA Ltd., February 1995.

[NUREG/CR-6463] H. Hetcht, M. Hecht, S. Graff, et at, ***Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems***, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997, also available at http://fermi.sohar.com/J1030/index.htm, last accessed in January 2002.

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

[PERENNIAL2001] ***JETS, A Perennial Validation Suite for the Java language***, http://www.peren.com/pages/jets_set.htm, last accessed in December 2001.

[Puschner+2001a] Puschner and A. J. Wellings, ***A Profile for High-Integrity Real-Time Java Programs***, Proceedings of ISORC 2001.

[Puschner+2001b] P. Puschner, G. Bernat, ***WCET Analysis of Reusable Portable Code***, Proceedings of the 13th Euromicro International Conference on Real-Time Systems, 2001.

[Saaltink+1997a] M. Saaltink and S. Michell, ***Ada 95 Trustworthiness Study; Analysis of Ada 95 for Critical Systems***, V2.0, ORA Canada, 27 March 1997.

[Saaltink+1997b] M. Saaltink and S. Michell, ***Ada 95 Trustworthiness Study; Guidance on the Use of Ada95 in the Development of High Integrity Systems***, V2.0, ORA Canada, 27 March 1997.

[Storey1996] N. Storey, ***Safety-Critical Computer Systems***, Addison Wesley Longman 1996.

[Sun1999] Sun Microsystems, ***Code Conventions for the Java Programming Language***, available at http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html, written in April 1999, last accessed in December 2001.

[TimeSys2002] TimeSys™, ***Products and Services: Real-Time Java***, available at http://www.timesys.com/rtj/index.html, last accessed in January 2002.

[UKMoD1991] U.K. Ministry of Defence, ***The Procurement of Safety Critical Software in Defence Equipment***, INTERIM Defence Standard 00-55 (PART 1: REQUIREMENTS)/Issue 1, 5 April 1991.

[USDoD1978] U.S. Department of Defence, ***Requirements for High Order Computer Programming Languages "STEELMAN"***, U.S. Department of Defence, 1978.

[USDoD1990] U.S. Department of Defence, ***Ada 9X Requirements***, Office of the Under Secretary for Defence Applications, Washington, D.C., December 1990.

[Watt+2000] D. A. Watt and D. F. Brown, ***Formalising the Dynamic Semantics of Java***, In Proceedings of the Third International Workshop on Action Semantics (AS2000), Recife, Brazil, May 2000.

[Wheeler1997] D. A. Wheeler, ***Ada, C, C++, and Java vs. the Steelman***, ACM Ada Letters, Vol. 17, Issue 4, July 1997.