Applying Jlint to Space Exploration Software

Cyrille Artho[1] and Klaus Havelund[2]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA

Abstract. Java is a very successful programming language which is also becoming widespread in embedded systems, where software correctness is critical. Jlint is a simple but highly efficient static analyzer that checks a Java program for several common errors, such as `null` pointer exceptions, and overflow errors. It also includes checks for multi-threading problems, such as deadlocks and data races. The case study described here shows the effectiveness of Jlint in finding certain faults, including multi-threading problems. Analyzing the reasons for false positives in the multi-threading warnings gives an insight into design patterns commonly used in multi-threaded code. The results show that a few analysis techniques are sufficient to avoid almost all false positives. These techniques include investigating all possible callers and a few code idioms. Verifying the correct application of these patterns is still crucial, because their correct usage is not trivial.

1 Introduction

Java is becoming more widespread in the area of embedded systems, both as a scaled-down “Micro Edition” [20] or by having real-time extensions [6, 5]. In such systems, software cannot always be replaced on a running system. Failures may have expensive or even catastrophic consequences. These costs are obviously prohibitively high when a software-related problem causes the failure of a space craft [14]. Therefore an automated tool which can detect faults easily, preferably early in the lifecycle of software, can be very useful. One tool that allows fault detection easily, even in incomplete systems, is Jlint. Among similar tools geared towards Java, it is one of the most suitable with respect to ease of use (no annotations required) and free availability (the tool is Open Source) [1].

1.1 The Java programming language

Java is a modern, object-oriented programming language that has had a large success in the past few years. Source code is not compiled to machine code, but to a different form, the *bytecode.* This bytecode runs in a dedicated environment, the *virtual machine.* In order to guarantee the integrity of the system, each class file containing bytecode is checked prior to execution [11, 19, 21].

The Java language allows each object to have any number of *fields,* which are attributes of each object. These may be static, i.e., shared among all instances of a certain class, or dynamic, i.e., each instance has its own fields. In contrast to that, *local variables* are thread-local and only visible within one method.

Java allows inheritance: a method of a given class may be *overridden* by a method of the same name. Similarly, fields in a subclass *shadow* those with the same name in the superclass. In general, these mechanisms work well for small code examples but may be dangerous in larger projects. Methods overriding other methods must ensure they do not violate invariants of the superclass. Similar problems occur with variable shadowing. The programmer is not always aware that a variable with the same name already exists on a different level, such as the superclass.

In order to prevent incorrect programs from corrupting the system, Java's virtual machine has various safety mechanisms built in. Each variable access is guarded against manipulating memory outside the allocated area. In particular, pointers must not be `null` when dereferenced, and array indices must be in a valid range. If these properties are violated, an *exception* is thrown indicating a programming error. This is a highly undesirable behavior in most cases. Ideally, such errors should be prevented by static analysis, rather than caught at run-time.

Furthermore, Java offers mechanisms to write multi-threaded programs. The two key mechanisms are locking primitives, using the `synchronized` keyword, and inter-thread synchronization with the `wait` and `notify` methods. Incorrect lock usage using too many locks may lead to *deadlocks*. For example, if two threads each wait on a lock held by the other thread, both threads cannot continue their execution. On the other hand, if a value is accessed with insufficient lock protection, *data races* may occur: two threads may access the same value concurrently, and the results of the operations are no longer deterministic.

Java's message passing mechanisms for threads also is a source of problems. A call to `wait` allows a thread to suspend until a condition becomes true, which must be signaled by `notify` by another thread. When calling `wait` the calling thread must ensure that it owns the lock it waits on, and also release any other locks before the call. Otherwise, remaining locks held are unavailable to other threads, which may in turn block when trying to obtain them. This can prevent them from calling `notify` which would allow the waiting thread to release its lock. This situation is also a deadlock.

1.2 Related work

Much effort has gone into fault-finding in Java programs, single-threaded and multi-threaded. The approaches can be separated into *static checkers*, which check a program at compile-time and try to approximate its run-time behavior, and *dynamic checkers*, which try to catch and analyze anomalies during program execution.

Several static analysis tools exist that examine a program for faults such as `null` pointer dereferences or data races. The ESC/Java [9] tool is, like Jlint, also based on static analysis, or more generally on theorem proving. It, however, requires annotation of the program. While it is more precise than Jlint, it is not nearly as fast and requires a large effort from the user to fully exploit the power of this tool [9].

Dynamic tools have the advantage of having more precise information available in the execution trace. The Eraser algorithm [22], which has been implemented in the Visual Threads tool [12] to analyze C and C++ programs, is such an algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races.

The Java PathExplorer tool (JPaX) [16] performs deadlock analysis and the Eraser data race analysis on Java programs. It furthermore recently has been extended with the high-level data race detection algorithm described in [3]. This algorithm analyzes how collections of variables are accessed by multiple threads.

More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just models thereof). This includes the Java PathFinder system (JPF) developed by NASA [15, 24], and similar systems [10, 8, 17, 4, 23]. Such systems, however, suffer from the state space explosion problem. In [13] we describe an extension of Java PathFinder which performs data race analysis (and deadlock analysis) in simulation mode, whereafter the model checker is used to demonstrate whether the data race (deadlock) warnings are real or not.

This paper focuses on applying Jlint [2] to the software for detecting errors statically. Jlint uses static analysis and abstract interpretation to find difficult errors at compile-time. A similar case study with Jlint has been made before, applying it to large projects [2]. The difference to this case study is that the other case study had scalability in mind. Jlint had been applied to packages containing several hundred thousand lines of code, generating hundreds of warning messages. Because of this, the warnings had been evaluated selectively, omitting some hard-to-check deadlock warnings. In this case study, an effort was made to analyze every single warning and also see what kinds of design patterns cause false positives.[1]

1.3 Outline

This text is organized as follows: Section 2 describes Jlint and how it was used for this project. Sections 3 and 4 show the results of applying Jlint to space exploration program code. Design patterns which are common among these two projects are analyzed in Section 5. Section 6 summarizes the results and concludes.

2 Jlint

2.1 Tool description

Jlint checks Java code and finds bugs, inconsistencies and synchronization problems by performing a data flow analysis, abstract interpretation, and building the lock graph. It issues warnings about potential problems. These warnings do not imply that an actual error exists. This makes Jlint unsound as a program prover. Moreover, Jlint can also miss errors, making it incomplete. The reason for this is that the goal was to make Jlint practical, scalable, and possible to implement it in a short time.

[1] Design patterns commonly denote compositions of objects in software. In this paper, the notion of composition is different. It includes lock patterns and sometimes only applies to a small part of the program. In that context, we also use the term “code idiom”.

Typical warnings about possible faults issued by Jlint are `null` pointer dereferences, array bounds overflows, and value overflows. The latter may occur if one multiplies two 32 bit integer values without converting them to 64 bit first.

Many warnings that Jlint issues are code guidelines: A local variable should never have the same name as a field of the same class or a superclass. When a method of a given name is overridden, all its variants should be overridden, in order to guarantee a consistent behavior of the subclass.

Jlint also includes many analyses for multi-threaded programs. Some of Jlint's warnings for multi-threaded programs are overly cautious. For instance, possible data race warnings for method calls or variable accesses do not necessarily imply a data race. The reason for such false positives are both difficulties inherent to static analysis, such as pointer aliasing across method calls, and limitations in Jlint itself, where its algorithms could be refined with known techniques.

Jlint works in two passes: a first pass, where all methods are analyzed in a modular way, and a second pass with the deadlock analysis. In the first pass, each method is analyzed with abstract interpretation. The abstraction used for numbers includes their maximal possible range and, for integers, bit masks that apply to them. For pointers, the abstraction records whether a pointer is possibly `null` or not, distinguishing the special `this` pointer, values loaded from fields, and new instance references created by the object constructor. The data flow analysis merges possible program states at branch targets but only executes loops once.

Most properties, such as possible value overflows, are analyzed in this first pass. The lock graph is also built in this first pass and checked for deadlocks in a second pass. A possible refinement would be to defer some data race analyses to the second pass, where global information, such as if a field is read-only, can be made available.

2.2 Warning review process

Jlint gives fairly descriptive warnings for each problem found. The context given is limited to the class in which the error occurs, the line number, and fields used or methods called. This is always sufficient to find the source of simple warnings, which concern *sequential properties* such as `null` pointer dereferences. These warnings are easy to review and were considered in a first review phase. The other warnings, concerning multi-threading problems, take much more time to consider, and were evaluated in a second phase.

The review process essentially checks whether the problems described in the warnings can actually occur at run-time. In simple cases, warnings may be ruled out given the algorithmic properties of the program. Complex cases include reviewing callers to the method in question.

Data race and deadlock warnings fall in the latter category. They require constructing a part of the call graph including locks owned by callers when a method is called. If it can be ensured that all calls to non-synchronized, shared methods are made only through methods that already employ lock protection, then there cannot be a data race.[2]

[2] Methods that access a shared field are also considered "shared" in this context. The lock used for ensuring mutual exclusion must be the same lock for all calls.

This review process can be rather time-consuming. Many warnings occur in similar contexts, so warnings referring to the same problem can usually be easily confirmed as duplicates. This part of the review process was not yet automated in any way but could be automated to a large extent with known techniques. Both cases studies were made without prior knowledge of the program code. It can be assumed that the time to review the warnings is shorter for the author of the code, especially when reviewing data race or deadlock warnings.

During the review process, Jlint's warnings were categorized to see whether they refer to the same problem. Such situations constitute calls to the same method from different callers, the same variable used in different contexts, or the same design pattern applied throughout the class. In a separate count, counting the number of distinct problems rather than individual warnings, all such cases were counted once. Note that the review activity was often interrupted by other activities such as writing this paper. We believe this reduced the overall time required because manual code reviews require much attention, and cannot be carried out in one run without a degradation of the concentration required.

3 First case study: Rover code

The first case study is a software module, called the Executive, for controlling the movement of the planetary wheeled rover K9, developed at NASA Ames Research Center. The run time for analyzing the code with Jlint was 0.10 seconds on a PowerPC G4 with a clock frequency of 500 MHz.

3.1 Description of the Rover project

K9 is a hardware platform for experimenting with rover technology for exploration of the Martian surface. The Executive is a software module for controlling the rover, and is essentially an interpreter of plans, where a plan is a special form of a program. Plans are constructed from high-level constructs, such as sequential composition and conditionals, but no while loops. The effect of while loops is achieved by assuming that plans are generated on the fly during rover operation as environment conditions change. The lowest level nodes of a plan are tasks to be directly executed by the rover hardware. A node in a plan can be further constrained by a set of conditions, which when failing during execution, cause the Executive to abort the execution of the subsequent sibling nodes, unless specified otherwise through options. Examples of conditions are pre-conditions and post-conditions, as well as invariants to be maintained during the execution of the node. The examined Executive consists of 7,300 lines of Java code. This code was extracted by a colleague from the original rover code, written in 35,000 lines of C++. The code is highly multi-threaded, and hence provides a risk for concurrency errors. The Java version of the code was extracted as part of a different project, the purpose of which was to compare various formal methods, such as model checking, static analysis, runtime analysis, and simple testing [7]. The code contained a number seeded of errors.

Type	Warnings	Problems found	Correct warnings	False positives	Time [min.]
`null` pointer	5	1	4	1	10
Integer overflow	2	2	2	0	5
`equals` overridden but not `hashCode`	2	1	2	0	1
String comparison as reference	1	0	0	1	1
Total: Sequential errors	10	4	8	2	17
Incorrect `wait/notify` usage	21	5	5	16	26
Data race, method call	157	5	18	139	112
Data race, field access	31	0	0	31	43
Deadlock	30	7	20	10	36
Total: Multi-threading errors	239	17	43	196	217
Total	249	21	51	198	234

Table 1. Jlint's warnings for the Rover code.

3.2 Jlint evaluation

Jlint issues 249 warnings when checking the Rover code. Table 1 summarizes Jlint's output. The first two columns show each type of problem and how many warnings Jlint generated for them. The third, forth and fifth column show the result of the manual source code analysis: how many actual, distinct faults, or at least serious problems, in the code were found, how many warnings described such actual faults, and how many were considered to be false positives. The last column shows the time spent on code review. In the first phase, focusing on sequential properties, ten warnings were reviewed, while the second phase had 239 warnings to be reviewed.

Sequential errors: Among the problems found are two integer overflows, where two 32-bit integers were multiplied to produce a 64 bit result. However, integer conversion took place *after* the 32 bit multiplication, where an overflow may occur.

Two other warnings referred to one problem, where `equals` was overridden, but not `hashCode`. This is dangerous because the modified `equals` method may return true for comparing two objects even though their `hashCode` differs, which is forbidden [21].

A noteworthy false positive concerned two strings that were compared as references. This was correct in that context because one of the strings was always known to be `null`.

Multi-threading errors: The number of deadlock and data race warnings given by Jlint was almost prohibitive. Yet, for answering the question why the false positives were generated, all warnings were investigated. All warnings were relatively easy to analyze. In most cases, possible callers were within the same class. Only for the most complex class, the call graph was large, making analysis more difficult.[3]

A surprisingly high number of multi-threading warnings were of type "Method `'<this>.wait|notify|notifyAll'` is called without synchronizing on `'<this>'`."

[3] The portion of the call graph to be investigated for this was up to eight methods deep.

After discounting dead code and false positives, one scenario remained: A lock was obtained conditionally, although it should be obtained in *all* cases, as required by the Java semantics for `wait` and `notify`. In the Rover code, this reflects a global switch in the original C++ program that would allow testing the program without locking, eliminating possible deadlocks at the cost of introducing data races. Java does not allow this, so the Java version of the program always needs to be run with locking enabled.

All data race warnings about shared field accesses were false positives. Reasons for false positives include the use of thread-local copies [18] or a thread-safe container class. In one case, only one thread instance that could access the shared field is ever generated. Evaluating data races for method calls was even more difficult and time-consuming. The errors found referred to cases where a read-only pattern, was broken by certain methods, creating potential data races. Because of their high number, the distribution of method data race warnings is noteworthy. A few classes which embody parallelized algorithms incurred the largest number of warnings, which were also the hardest to review. Classes encapsulating data are usually much simpler. Because some of these were heavily used in the program, a few of them were also responsible for a large number of warnings. However, these warnings were usually much easier to review.

The 30 deadlock warnings all referred to the same two classes. There were two sets of warnings, the first set containing ten, the second one 20 warnings. The first ten warnings, all of them false positives, showed incomplete synchronization loops in the lock graph. The next 20 warnings, referring to seven methods, showed the same ten warnings with another edge in the lock graph, from the callee class back to the caller. Such a synchronization loop includes two sequences of different lock acquisitions in reverse order. This makes a cyclic deadlock possible. Therefore these warnings referred to actual faults in the code.

Results: In only 15 minutes, four faults could be found by looking at the ten warnings referring to sequential properties. While reviewing the multi-threading warnings was time-consuming due to the complex interactions in the code, it helped to highlight the critical parts of the code. The effort was justifiable for a project of this complexity.

3.3 Comparison to other projects

In an internal case study at NASA Ames [7], several other tools were applied to the Rover code base, detecting 38 errors. Among these errors were 18 seeded faults. Interestingly, most of these errors found were not those detected by Jlint. Almost all the seeded bugs concerned algorithmic problems or hard-to-find deadlocks, which Jlint was not capable of finding. However, Jlint in turn detected a lot of faults which were not found by any other tool. Table 2 compares Jlint to the other case studies. In that table, missed faults include both sequential and multi-threading properties.

The eleven new bugs found by Jlint were a great success, even considering that the seven deadlocks correspond to two classes where other deadlocks have been known to occur. However, Jlint reported different methods than those reported in other analyses.

Error type	#	Evaluation
Seeded faults	18	Not found by Jlint
Non-seeded faults, other than overflow	18	Not found by Jlint
Integer overflow	2	Found by both case studies
`null` pointer	**1**	**New** (i.e., only found by Jlint)
`equals` overridden but not `hashCode`	1	Translation artifact (not occurring in the C version)
Incorrect `wait`/`notify` usage	5	Debugging artifact (not executable in Java)
Data races	5	**3 new**, 2 dead code (unused methods)
Deadlocks	**7**	**new** (two classes known to be faulty involved)

Table 2. Comparison of errors found by Jlint and by other tools.

4 DS1

The second case study consisted of an attitude control system and a fault protection system for the Deep Space 1 (DS1) space craft. It took 0.17 seconds to check the entire code base on the same PowerPC G4 with a clock frequency of 500 MHz.

4.1 Description of DS1

DS1 was a technology-testing mission, which was launched October 24 1998, and which ended its primary mission in September 1999. DS1 contained and tested twelve new kinds of space-travel technologies, for example, ion propulsion and artificial intelligence for autonomous control. DS1 also contained more standard technologies, such as an attitude-control system and a fault-protection system, coded in C. The attitude-control system monitors and controls the space craft's attitude, that is, its position in 3-dimensional space. The attitude is controlled by small thrusters, which can be pointed, and fired, in different directions. The fault-protection system monitors the operation of the space craft and initiates corrective actions in case errors occur. The code examined in this case study is an 8,700-line Java version of the attitude-control system and fault-protection system, created in order to examine the potential for programming flight software in Java, as described in [5]. That effort consisted in particular of experimenting with the real-time specification for Java [6]. The original C code was re-designed in Java, using best practices in object-oriented design. The Java version used design patterns extensively, and put an emphasis on pluggable technology, relying on interfaces.

4.2 Jlint evaluation

Sequential errors: Again, a first evaluation of Jlint's warnings included only the sequential cases. Table 3 shows an overview. Eleven warnings referred to name clashes in variable names, a large risk of future programming errors. False positives resulted from either dead code, a code idiom that was poor choice but acceptable in that case, and compiler artifacts introduced by inner classes. Three warnings reported problems with overridden methods, where several versions of a method with the same name but

Type	Warnings	Problems found	Correct warnings	False positives	Time [min.]
Local variable shadows field	4	2	2	2	2
Component shadows base class	7	0	0	7	3
Incomplete method overriding	3	3	3	0	3
`equals` overridden but not `hashCode`	1	0	0	1	1
Total: Sequential errors	15	5	5	10	9
Incorrect `wait`/`notify` usage	7	0	0	7	3
`run` method not `synchronized`	5	0	0	5	0
Overriding `synchronized` methods	3	0	0	3	2
Data race, field access	1	0	0	1	7
Data race, method call	20	1	6	14	38
Deadlock	11	0	0	11	20
Total: Multi-threading errors	47	1	6	41	70
Total	62	6	11	51	79

Table 3. Jlint's warnings for the DS1 code.

different parameter lists ("signatures") were only partially overridden. This must be avoided because inconsistencies among the overridden and inherited variants are almost inevitable.

Multi-threading errors: In the second phase, the 47 multi-threading warnings were investigated. Most of them were false positives: Warnings about `run` methods which are not `synchronized` are overly conservative. Warnings about `wait`/`notify` were caused by the unsoundness of Jlint's data flow analysis. False positives for data race warnings were mostly caused by the fact that Jlint does not analyze all callers when checking methods for thread safety. If all callers synchronize on the same lock, a seemingly unsafe method becomes safe. Other reasons for false positives were the use of thread-safe container classes in such methods, the use of read-only fields, and per-thread confinement [18], which always creates a new instance as return value.

The six warnings indicating an error concerned calls to a logger method. In the logger method, there were indeed data races, even though they may not be considered to be crucial: The output of different formatting elements of different entries to be logged may be interleaved.

Again, as in all non-trivial examples, deadlock warnings are almost impossible to investigate in detail without a call graph browsing tool. Nevertheless, an effort was made. After 12 minutes, it was found that the first deadlock warning was a false alarm due to the lack of context sensitivity in Jlint's call graph analysis. After this, most warnings could be dismissed as duplicates of the first one. In the two remaining cases, Jlint's warnings did not give the full loop context, so they could not be used.

Results: Most sequential warnings could be evaluated very quickly. The problems found were code convention violations, which would not necessarily cause run-time

Code base	Rover			DS1		Total
Problem category	`wait/` `notify`	Data race (field)	Data race (method)	Data race (field)	Data race (method)	
Read-only fields	–	9	19	–	3	31
Synchronization for all callers	12	–	7	1	2	22
Return copy of data	–	–	3	–	1	4
Thread-local copy during operation	–	1	1	–	–	2
Thread-safe container	–	–	1	–	2	3
One thread instance	–	1	–	–	–	1
Total	12	11	31	1	8	63

Table 4. Design patterns for avoiding data races in seemingly unsafe methods.

errors. However, they are easy to fix and should be addressed. Reviewing the data race warnings was relatively simple, although it would have been much easier with a call graph visualization tool. Most false positives could have been prevented by a more complete call graph analysis or recognizing a few simple design patterns.

5 Design patterns in multi-threaded software

Sections 3 and 4 have shown that sequential properties are easy to evaluate with the aid of a static analysis tool. This is not the case with multi-threading problems. There are two ways to improve the situation: Make the evaluation of warnings easier using visualization tools, or improve the quality of the analysis itself, reducing the false positives. We focused on the latter aspect. When analyzing the warnings, it soon became apparent that only a few common code idioms were behind the problems. The remainder of this paper investigates what patterns are used to avoid multi-threading problems.

Table 4 shows an overview of the different design patterns used in the code of the two space exploration projects to avoid conflicts with unprotected fields or methods. The counts correspond to the applications of these patterns, all of which result in one or more spurious warnings when analyzed with Jlint. When using these patterns, there appears to be a data race, if a method is considered in isolation or without considering thread ownership. There is no data race when considering the entire program.

The most common idiom used to prevent data races was the use of read-only values. Read-only fields are usually declared `final` and not changed after initialization. Because this declaration discipline is not always followed strictly, recognizing it statically is not always trivial, but nevertheless feasible by checking all uses of a given field in the entire code. Ensuring global thread-safety in such cases is of course only possible in the absence of dynamic class loading. Other design patterns include:

- Ensuring mutual exclusion in an unsafe method by having all callers of that method acquire a common lock. Such callers work as a thread-safe wrapper around unsynchronized parts of the code.

– The usage of (deep) copies of data returned by a method ensures that the “working copy” used subsequently by the caller remains thread-local [18]. This eliminates the need for synchronization in the caller.
– Copying method parameters restricts data ownership to the called method and the current thread [18]. The callee then does not have to be synchronized, but it is not allowed to use any shared data other than the copied parameters supplied by the caller. Doing otherwise would again require synchronization.
– Legacy container data structures such as `Vector` are inherently thread-safe because they internally use synchronization [21].
– Finally, if there exists only one thread instance of a particular class, no data races can occur if that thread is the only type that calls a certain method.

Two cases of false positives were not included in this summary: unused methods (dead code) and conditional locking based on a global flag used for debugging `wait`/`notify` locking (which was permissible in the original C++ Rover code but not in the Java version).

This study indicates that four design patterns prevail in cases where code is apparently not thread-safe: Synchronization of all callers, use of read-only values, thread-local copies of data, and the use of thread-safe container classes. Although simple patterns prevail, their usage is not always trivial: Some of the data race warnings for the Rover code pointed out cases where it was attempted to use the read-only pattern, but the use was not carried out consistently throughout the project. Such a small mistake violates the property that guarantees thread-safety. This discussion so far concerned only data race warnings. No prevailing pattern has been found in the case of deadlocks, where the programmer has to ensure no cyclic lock dependency arises between threads.

6 Conclusions

Space exploration software is complex. The high costs incurred by potential software failures make the application of fault-finding tools very fruitful. Jlint was very successful as such a tool in both case studies, complementing the strengths of other tools. In each project, the study found four or five significant problems within only 15 minutes of evaluating Jlint’s warnings. The multi-threading warnings were more difficult and time-consuming to evaluate but still effective at pointing out critical parts in the code.

An analysis of the false positives showed that in apparently thread-unsafe code, four common design patterns ensure thread-safety in all cases. Static analysis tools should therefore be extended with specific algorithms geared towards these patterns to reduce false positives. Furthermore, these patterns were not always applied correctly and are still a significant source of programming errors. This calls for tools that verify the correct application of these patterns, thereby pointing out even more subtle errors than previously possible.

References

1. C. Artho. Finding Faults in Multi-Threaded Programs. Master’s thesis, ETH Zürich, 2001.

2. C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *Proceedings of the 13th ASWEC*, pages 68–75. IEEE CS Press, 2001.
3. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In *VVEIS'03*, April 2003.
4. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy, 2001.
5. E. G. Benowitz and A. F. Niessner. Java for Flight Software. In *Space Mission Challenges for Information Technology*, July 2003.
6. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
7. G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *SEI Software Model Checking Workshop*, 2003. Extended abstract.
8. J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. 22nd International Conference on Software Engineering*, Ireland, 2000. ACM Press.
9. D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.
10. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, France, 1997.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Virtual Language Specification, Second Edition*. Addison Wesley, 2000.
12. J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.
13. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
14. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *5th NASA Langley Formal Methods Workshop*, June 2000. USA.
15. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
16. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
17. G. Holzmann and M. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE'99, International Conference on Software Engineering*, USA, 1999. IEEE/ACM.
18. D. Lea. *Concurrent Programming in Java, Second Edition*. Addison Wesley, 1999.
19. T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.
20. Sun Microsystems. Connected, limited device configuration. specification version 1.0a, may 2000. `http://java.sun.com/j2me/docs/`.
21. Sun Microsystems. Java 2 documentation. `http://java.sun.com/j2se/1.4/docs/`.
22. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
23. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
24. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, 2000.