JavaFAN: A Rewriting Logic Approach to Formal Analysis of Multithreaded Java Programs*

Azadeh Farzan
University of Illinois at Urbana Champaign
201 N. Goodwin
Urbana, IL
afarzan@cs.uiuc.edu

Feng Chen
University of Illinois at Urbana Champaign
201 N. Goodwin
Urbana, IL
fengchen@cs.uiuc.edu

Jose Meseguer
University of Illinois at Urbana Champaign
201 N. Goodwin
Urbana, IL
meseguer@cs.uiuc.edu

Grigore Rosu
University of Illinois at Urb
Champaign
201 N. Goodwin
Urbana, IL
grosu@cs.uiuc.edu

ABSTRACT

JavaFAN (Java Formal ANalysis) is a multithreaded program analysis framework based on rewriting logic specifications of Java. It can perform several types of analysis, including symbolic execution of Java programs, detection of safety violations searching through the potentially unbounded state space of a multithreaded program, and explicit state model checking of programs whose state space is finite. Both Java source-code and byte-code analyses are possible. The former is user-friendly, with counter-examples directly related to familiar Java source-code, and the latter affords a more precise analysis of the running code, not depending on the correctness of the compiler, and can be used even when the Java source-code of the program is not available.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

Rewriting logic [18] extends equational logic with rewriting rules and has been mainly introduced as a *unified model of concurrency*. Many formal theories of concurrency have been naturally mapped into rewriting logic during the last decade. A next natural challenge is to define mainstream concurrent programming languages in rewriting logic and then use rewriting-based analysis techniques to reason about concurrent program properties. There is already a substantial body of case studies, of which we only mention [26, 25, 28], backing one of the key claims of this paper, namely that *rewriting logic can be fruitfully used as a unifying framework for defining programming languages*.

The focus here is on showing that a rewriting-based approach can be computationally used in practice to formally analyze concurrent programs in a real programming language like Java, and in this way of providing formal analysis techniques for Java multi-threaded programs. Our techniques are based on rewriting logic specifications of the Java language and of the Java Virtual Machine (JVM) bytecode semantics, both executable in the Maude language [3]. These specifications define both the sequential and the concurrent semantics of Java threads and can be used to perform the following types of formal analysis, at both the source-code and the byte-code levels: (1) *symbolic simulation*, with specifications used as *interpreters* that execute programs with actual or symbolic inputs; (2) *breadth-first formal analysis* of a concurrent program's state space to find violations of safety properties; and (3) *model checking* of linear temporal logic (LTL) properties for programs whose state space is finite. To make the analysis framework user friendly, we have implemented a tool called JavaFAN (the Java Formal ANalyzer) that wraps the underlying Maude specifications and accepts Java or JVM code from a user as input. One can analyze a Java program either directly, using the Java semantics, or first compile it and then input the resulting JVM code to JavaFAN. One can also analyze directly byte-code when the source-code is not available. Symbolic simulation can be performed without the user even being aware of the underlying Maude.

We have analyzed in JavaFAN a number of Java programs. Some of the case studies conducted are reported in this paper. Due to space limitations, other case studies are reported on the WWW [8]; the same HTTP address also contains the Maude JVM semantics, example codes, and a downloadable JavaFAN.

Related Work. There is a vast literature on formal analysis of Java programs that we cannot exhaustively review

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

FSE NewPort Beach, CA

here. We can classify the different approaches as focusing on either *sequential* or *concurrent* programs. Our work falls in the second category. More specifically, it belongs to a family of approaches that use a *formal executable specification of the concurrent semantics* of Java or the JVM as a basis for formal reasoning. Two other approaches in precisely this category are one based on the ACL2 logic and theorem prover [16], and another based on a formal JVM semantics and reasoning based on Abstract State Machines (ASM) [24]. Our approach seems complementary to both of these, in the sense that it provides new formal analysis capabilities, namely search and LTL model checking. The ACL2 work is in a sense more powerful, since it uses an inductive theorem prover, but this greater power requires greater expertise and effort.

Outside the range of approaches based on executable formal specification, but somewhat close in the form of analysis, is NASA's Java Path Finder (JPF) [12, 2], which is an explicit state model-checker for Java bytecode based on a modified version of a C implementation of a JVM. Preliminary rough comparisons of JavaFAN and JPF[1] are encouraging, in the sense that we can analyze the same types of JVM programs of relatively the same size.

Other related work includes [23], which proposes an algorithm that takes the bytecode for a method and generates a temporal logic formula that holds iff the bytecode is safe; an off-the-shelf model checker can then be used to determine the validity of the formula. Among the formal techniques for *sequential* Java programs, some approaches similar in spirit to ours include the ACL2-based work on the defensive JVM [4], which focuses on dynamic safety checks, and the collective effort around the JML specification language and verification tools for sequential Java [7], where formal executable specifications of Java semantics in PVS are used, e.g. see [27], to verify Java programs.

Another approach to define analysis tools for Java is based on *language translators*, generating simpler language code from Java programs and then analyzing the later. Bandera [5] extracts abstract models from Java programs, specified in different formalisms, such as PROMELA, which can be further analyzed with specialized tools sach as SPIN [13]. JCAT [6] also translates Java into PROMELA. [21] presents an analysis tool which translates Java bytecode into C++ code representing an executable version of a model checker. While the translation-based approaches can benefit from abstraction techniques being integrated into the generated code, they inevitably lead to natural worries regarding the correctness of the translations. Unnecessary overhead seems to be also generated, at least in the case of [21]; for example, exactly the same Remote Agent Java code that can be analyzed in 0.1 second in JavaFAN [8] takes more than 2 seconds even on the most optimized version of the tool in [21].

2. SPECIFYING CONCURRENT LANGUAGES IN REWRITING LOGIC

In this section we explain the general rewriting logic formal specification methodology that we use to define concurrent programming languages. As a whole, the specification of a language is a *rewrite theory*, that is, a triple (Σ, E, R), with (Σ, E) an equational specification with signature of operators Σ and a set of equational axioms E; and with R a collection of labelled rewrite rules. The equational specification describes the *static* structure of the concurrent system's state space as an algebraic data type. The *dynamics* of the system are described by the rules in R that specify local concurrent *transitions* that can occur in the system axiomatized by (Σ, E, R), and can be applied *modulo* the equations E. Particularly important equations are those of associativity, commutativity and identity of binary operations, which allow us to elegantly and effectively define (and implicitly implement) the crucial infrastructure of programming languages, including environments and stores as well as the lookup operations on these.

Maude [3] supports, executes and formally analizes rewriting logic theories, via a suite of efficient algorithms for term rewriting, state-space search, and linear temporal logic (LTL) model checking. These features are essential for JavaFAN. Thanks to Maude's `frewrite` command, the formal specifications for Java and the JVM become *interpreters*, in which we can simulate the *fair execution* of a concurrent Java program at the source and JVM levels. Maude's `search` command and its LTL model checker allow exhaustive explorations of a concurrent program's state space. An important point about our methodology is that, since according to the semantics of rewriting logic [18], rewriting with R happens *modulo* E, only the rules in R (not the equations E) can produce a state space explosion. Therefore, all deterministic aspects of the Java computation are specified with equations, with rules used only for concurrent features. The number of such rules is relatively small (only 15 in a 150 JVM instruction specification, and a similar number for Java) making state spaces as small as possible. The `search` command provides breadth-first search and can be used as a semidecision procedure to find errors in possibly infinite state spaces. LTL model checking instead is a decision procedure but requires such state spaces to be finite. As already pointed out, E, besides having confluent equations specifying deterministic Java computations, contains associativity, commutativity and identity (ACI) axioms to represent the concurrent state of a Java or JVM computation as a *multiset* of entities such as the memory, continuations, and in the JVM specification a collection of object and message data structures representing JVM entities (objects, threads, and so on) as objects in the ACI soup. Therefore, most of our equations and rules are applied *modulo* ACI, which in Maude is a highly optimized and efficient process for commonly occurring lefhandside patterns. For example, Figures **??** and **??** present two typical object-oriented rewrite rules. An *object* in a given state is represented as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*.

3. JAVA AND JVM REWRITING SEMANTICS

We use Maude to specify the operational semantics of a sufficiently large subset of Java and the JVM, including multithreading, inheritance, polymorphism, object references, and dynamic object allocation. We do not support native methods and many of the Java built-in libraries at the moment. Besides mathematical correctness, efficiency of the re-

[1]Authors thank Willem Visser for examples and valuable information about JPF.

```
subsort StateAttribute < State .
op _,_ : State State -> State [assoc comm id: empty] .
op c : Context -> StateAttribute .  *** Thread Context
op m : Store -> StateAttribute .    *** Store the mapping from locations to values
op l : LockList -> StateAttribute . *** The synchronization locks
op w : LockList -> StateAttribute . *** The waiting locks
op s : ObjEnv -> StateAttribute .   *** The static fields of classes
op out : Output -> StateAttribute . *** Collected outputs
```

Figure 1: Signature of the Java Program State.

```
subsort ContextItem < Context .
op _,_ : Context Context -> Context [assoc comm id: noItem] .
op k : Continuation -> ContextItem . *** The continuation of the thread
op e : Env -> ContextItem .          *** The local environment for the thread
op o : Object -> ContextItem .       *** The object running the thread
```

Figure 2: Signature of the Java Thread Context

sulting tools has also been a key goal in our formal language definitions. Consequently, Java and the JVM are modeled differently. For Java, a quite efficient continuation-based style is adopted, while for the JVM we use an object oriented style that makes the specification simpler and easier to understand. The essential idea, however, is the same: the use of rewriting logic to specify the changes to the program state. Our language definitions underwent a series of optimizations, which have significantly improved the overall efficiency of JavaFAN. A major design principle is to keep the size and the number of system's states minimal, so that Maude's internal algorithms perform optimally. The former has been achieved through separating the *static* and *dynamic* aspects of the program, maintaining only the dynamic part in the system's state. For the latter, we reduced the number of rewrite rules in the specification. Subsection 3.3 discusses these optimizations in more detail.

3.1 Continuation-based Semantics of Java

The semantics of Java is defined modularly –different features of the language are defined in separate modules– to ease extensions and maintenance. The specification contains 75 modules, about 600 equations and 15 rewriting rules. We next show a few snapshots of Java's definition, see [8] for the entire definition.

3.1.1 States

A state is an AC soup of state attributes, such as threads, memory, synchronization information, etc. Figure 1 shows the main constructors for the Java state. We adopt the well-known representation of program states for object-oriented languages [15], where an *environment* maps variable names to locations, and a *store* maps locations to values. Objects are stored as environments enriched with their class types. To also support multi-threaded programs, we introduce the notion of *thread context*, which consists of three components (Figure 2): (1) a continuation, (2) the thread environment, and (3) the corresponding object. The `continuation` maintains the control context of the thread, which explicitly specifies the next steps to be performed by the thread.
In addition to the thread context, there are other shared (by threads) state attributes, including the static fields of the classes, the store of values, the synchronization information, and also the collected outputs of the execution.

```
eq k((X += E) -> K) = k(loc(X) -> += (E) -> K) .
rl c(k(L -> += (E) -> K), context), m([L,V] M) =>
                c(k([E | V] -> + -> (set&fetch(L) -> K)), context), m([L,V] M) .
```

Figure 3: Semantics of += Expression

3.1.2 Continuations

Conditional rewrite rule definitions tend to be inefficiently executed by rewriting systems because of the potentially unbounded *control context* that needs to be maintained. Continuations are a typical technique to transform the unconstrollable control context into controllable *data context*, by stacking the sequence of actions that still need to be executed by a program or thread. The use of continuations has resulted in a definition of Java using only unconditional equations and rewriting rules, which can execute Java program orders of magnitude faster than a conditional rewriting semantics. E.g., the addition operation can be specified using continuations as follows.

```
Evaluate the operand expressions first:     eq k((E1 + E2) -> K) = k((E1, E2) -
. Add the results and pass the final result:  eq k((v1, v2) -> + -> K) = k((v1
K).
```

`K` represents the remaining part of the continuation. Once the expressions on the top of the continuation (`E1, E2`) are evaluated, their results will be passed to the remaining continuation. Continuations significantly facilitate the definition of flow-control instructions, such as `break`, `continue`, `return`, and exceptions.

Each thread context contains one continuation item, which stores the remaining execution flow of the thread. Thus, the two generic equations above can apply within any thread context in the state soup.

3.1.3 Thread communication

Threads can communicate via shared variables. The order in which they access shared variables or synchronization objects leads to different multithreaded computations, including potentially eroneous ones. In order for the Maude search and model checking procedures to appropriately explore all the multithreaded computations of a Java program, one has to ensure that the semantics of all the statements involving potentially shared data, such as read/write of variables, are defined as rewriting rules rather than equations.
Figure 3, e.g., shows the semantics of the `+=` operator (`X += E`). The equation says how the statement should be evaluated: the location of the left operand needs to be first obtained, then the marker `+=` followed by the expression is placed in the continuation, to state what one should do with the location once calculated. Having a location `L`, the operation `+=`, and an expression `E` on top of the continuation, the rewrite rule creates three new tasks for the continuation: (1) Compute the expression `E` and add it to the value `V` (value associated to location `L`) which is done through `[E | V] -> +`; (2) write the result from part (1) to location `L` and also pass it to the remaining continuation `K` via the operation `set&fetch(L)`, which is not defined here due to space limitations. What is important here is that the above has to be a rewriting rule instead of an equation because it involves a memory read; another thread may write the same location, so one wants to potentially allow both permutations of location accesses.

3.2 Object-based Semantics of the JVM

The state of the JVM is represented as a multiset of objects and messages in Maude [3].

Objects in the multiset fall into four major categories: (1) objects which represent Java objects, (2) objects which represent Java threads, (3) objects which represent Java classes, and (4) auxiliary objects used mostly for definitional purposes. Rewrites (with rewrite rules and equations) in this multiset (modulo associativity, commutativity, and identity) model the changes in the state of the JVM. In a rewrite, there is usually one thread involved, together with classes and/or objects that may be needed to execute the next bytecode instruction. Reader can find a detailed discussion on the JVM model in [9].

3.3 Optimizations

Below, we discuss two major optimizations we applied at the both levels to decrease the size and number of system states.

3.3.1 Size of the State

In order to keep the state of the system small, we only maintain the dynamic part of the Java classes inside the system state. Every attribute of Java threads and Java objects can potentially change during the execution, but Java classes contain attributes that remain constant all along, namely, the methods, inheritance information, and field names. This, potentially huge amount of information, does not have to be carried along in the state of the JVM. The attributes of each class are grouped into *dynamic* and *static* attributes. The former group appears as a part of system's state, and the latter group is kept outside the pool, in a constant accessed through auxiliary operations.

3.3.2 Rules vs. Equations

Using equations for all deterministic computations, and rules only for concurrent ones leads to great savings in state space size. The key idea is that the are only two cases in which a thread interacts with (possibly changes) the outside environment are shared memory access and acquiring locks. Examples of the former include the semantics of `+=` at Java language level (see Section 3.1) and of the instruction `getfield` (see Section **??**) at the bytecode level. As an example for the latter case (see Section **??**), we refer the reader to semantics of acquiring a lock at both levels.

4. FORMAL ANALYSIS

Using the underlying fair rewriting, search and model checking features of Maude, JavaFAN can be used to formally analyze Java programs in bytecode format. The Maude's specification of the JVM can be used as an interpreter to simulate fair JVM computations by rewriting. *Breadth-first search analysis* is a semi-decision procedure that can be used to explore all the concurrent computations of a program looking for safety violations characterized by a pattern and a condition. Infinite state programs can be analyzed this way. For finite state programs it is also possible to perform explicit-state model checking of properties specified in linear temporal logic (LTL).

4.1 Simulation

Our Maude specification provides executable semantics for Java and the JVM, which can be used to execute Java programs in source code and bytecode formats. This simulator can also be used to execute programs with symbolic inputs. Maude's `frewrite` command provides fair rewriting with respect to objects, and since all Java threads are defined as objects in the specification, no thread ever starves, although no specific scheduling algorithm is imposed[2].

> *This part has to be rewritten very carefully.*
>
> *To facilitate user interaction, the JVM semantics specification is integrated within a prototype tool, called JavaFAN, that accepts standard bytecode as its input. The user can use* `javac` *(or any Java compiler) to generate the bytecode. She can then execute the bytecode in JavaFAN, being totally unaware of Maude. We use* `javap` *as the disassembler on the class files along with another disassembler* `jreversepro` *[14] to extract the constant pool information that* `javap` *does not provide.*

4.2 Breadth-first Search

Using the simulator (Section 4.1), one can explore only one possible trace (modeled as sequence of rewrites) of the Java program being executed. Maude's `search` command allows exhaustively exploring all possible traces of a Java program. The breadth-first nature of the `search` command gives us a semi-decision procedure to find errors even in infinite state spaces, being limited only by the available memory. Below, we discuss a number of case studies.

4.2.1 Remote Agent.

The Remote Agent (RA) is an AI-based spacecraft controller that has been developed at Nasa Ames Research Center and has been part of the software component of NASA's Deep Space 1 shuttle, the first New Millennium Mission testing several cutting-edge technologies such as the ionic engine and the on-board optical navigation. However, on Tuesday, May 18th, 1999, Deep Space 1's software deadlocked 96 million kilometers away from the earth and consequently had to be manually interrupted and restarted from ground. The blocking was due to a missing critical section in the RA that had led to a data-race between two concurrent threads, which further caused a deadlock [10, 11]. This real life example shows that even quite experienced programmers can miss data-race errors in their programs. Moreover, these errors are so subtle that they often cannot be exposed by intensive testing procedures, such as NASA's, where more than 80% of a project's resources go into testing. This justifies formal analysis techniques like the ones presented in this paper which could have caught that error.

The RA consists of three components: a Planner that generates plans from mission goals; an Executive that executes the plans; and a Recovery system that monitors RA's status. The Executive contains features of a multithreaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multithreading errors. Events and tasks are two major components (see Figure **??** in the Appendix). In order to catch the events that occur while tasks are executing, each event has an associated event counter that is

[2]By not committing to any specific thread scheduling, we have the advantage of detecting the violations that may happen in some scheduling schemes, but not in others.

increased whenever the event is signaled. A task then only calls `wait_for_event` in case this counter has not changed, hence, there have been no events since it was last restarted from a call of `wait_for_event`.

The error in this code results from the unprotected access to the variable `count` of the class `Event`. When the value of `event1.count` is read to check the condition, it can change before the related action is taken, and this can lead to a possible deadlock. This example has been extensively studied in [10, 11]. Using the search capability of our system, we also found the deadlock in the same faulty copy in 0.1 second in source code level and in 0.3 second in bytecode level. This is while the tool in [21] finds it in more than 2 seconds in its most optimized version[3].

The Thread Game.

The Thread Game [20] is a simple multithreaded program which shows the possible data races between two threads accessing a common variable (see Figure **??** in the Appendix). Each thread reads the value of the static variable `c` twice and writes the sum of the two values back to `c`. Note that these two readings may or may not coincide. An interesting question is what values can `c` possibly hold during the infinite execution of the program. Theoretically, it can be proved that all natural numbers can be reached [20].

We can use Maude's search command to address this question for each specific value of N. The search command can find one or all existing solutions (sequences) that lead to get the value N. Tablel 1 presents some numbers and the amount of time (in seconds) to find a solution for that number in both source code and bytecode levels.

N	50	100	200	400	500	1000
JVM	7.2	17.1	41.3	104	4.5m	10.1m
Java	2.7	6.6	17	54.7	2m	5.1m

Table 1: Thread Game Times.

4.3 Model Checking

Maude's model checker is explicit state and supports Linear Temporal Logic. This general purpose rewriting logic model checker can be directly used on the Maude specification of JVM's concurrent semantics. This way, we obtain a model checking procedure for Java programs for free. The user has to specify in Maude the atomic propositions to be used in order to specify relevant LTL properties. We illustrate this kind of model checking analysis by the following examples.

4.3.1 Dining Philosophers.

See Figure **??** in the Appendix for the version of the dining philosophers problem that we have used in our experiments. The property that we have model checked is whether all the philosopher can eventually dine. Each philosopher prints her ID when she dines. Therefore, to check whether the first philosopher has dined, we only have to check if 1 is written in the output list (see Section **??** for the output process). The LTL formula can be built based on propositions defined as follows. `op Check : Int -> Prop`, where `Check(N)` will be true at some state if the output list contains all the numbers from 1 to N. In this case, we check the following LTL formula using the `modelCheck`, where `InitialState` is the initial state of the program defined automatically. The formula that we model checked is $\Diamond$Check(n) for n philosophers. The model checker generates counterexamples, in this case a sequence of states that lead to a possible deadlock. The sequence shows a situation in which each philosopher has acquired one fork and is waiting for the other fork. Currently, we can detect the deadlock for up to 9 philosophers (Table 2). We also model checked a slightly modified version of the same program which avoids deadlock. In this case, we can prove the program deadlock-free when there are up to 7 philosophers. This compares favorably with JPF [2, 12] which for the same program cannot deal with 4 philosophers.

[3]All the performance results given in this section are in seconds on a 2.4GHz PC.

Table 2: Dining Philosophers Times

Tests	**Times(s)**
DP(4)	0.64
DP(5)	4.5
DP(6)	33.3
DP(7)	4.4m
DP(8)	13.7m
DP(9)	803.2m
DF(4)	21.5
DF(5)	3.2m
DF(6)	23.9m
DF(7)	686.4m

4.3.2 2-stage Pipeline.

Figure **??** in the Appendix implements a pipeline computation, where each pipeline stage executes as a separate thread. Stages interact through *connector* objects that provide methods for `adding` and `taking` data. The property we have model checked for this program is related to the proper shutdown of pipelined computation, namely, "the eventual shutdown of a pipeline stage in response to a call to `stop` on the pipeline's input connector". The LTL formula for the property is $\Box$(c1stop $\rightarrow \Diamond(\neg$stage1return)). JavaFAN model checks the property and returns `true` in 17 minutes (no partial order reduction was used). This compares favorably with the model checker in [21] which without using the partial order reduction performs the task in more than 100 minutes.

5. CONCLUSION AND FUTURE WORK

We have presented JavaFAN, have explained its underlying formal executable specification of the concurrent semantics, and have illustrated its use in formally analyzing JVM code by simulation, search, and model checking. Although the results so far are encouraging, much work remains ahead. One important issue is *how to scale up* to larger programs in face of the inherent combinatorial state explosion. Both new algorithms and better control of the *granularity* of the concurrency, when this can be shown to be safe, are needed. Another important technique needed to drastically reduce the state space size is *abstraction*. Recent equational abstraction techniques for Maude specifications [19], as well as the experience in using abstraction to model check programs in work such as, e.g., [1] should be exploited in this

regard. *Partial-order reduction* [22] techniques for rewriting logic model checking is certainly a promissing area of further research. Another non-trivial issue is to properly deal with *foreign functions*, since most Java libraries are in fact implemented in C.

Other future research involves *widening the range of formal analyses*. On one side, theorem proving support would be desirable; the work of the JML and ACL2 researchers will be helpful in this regard, but the kind of logic needed to specify properties must go beyond JML, where only sequential programs are treated. On the other side, it would be interesting to incorporate in JavaFAN *domain-specific certification* and *specification-based monitoring* techniques.

6. ADDITIONAL AUTHORS

7. REFERENCES

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS'01*, LNCS 2031, pages 268 – 283, 2001.

[2] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *ASE'00*, pages 3 – 12, 2000.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, 2003. `http://maude.cs.uiuc.edu/manual`.

[4] R. M. Cohen. The defensive Java Virtual Machine specification. Technical report, Electronic Data Systems Corp, 1997.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R. Zheng, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE'00*, pages 439 – 448, 2000.

[6] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - practice and Experience*, 29(7):577 – 603, 1999.

[7] G. T. L. et al. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.

[8] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. JavaFAN. `fsl.cs.uiuc.edu/es/javafan`.

[9] A. Farzan, J. Meseguer, and G. Roşu. Formal jvm code analysis in javaFAN. In *Proceedings of AMAST'04*, to appear.

[10] 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 *the 5th NASA Langley Formal Methods Workshop*, 2000.

[11] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749 – 765, Aug. 2001. Previous version appeared in Proceedings of the 4th SPIN workshop, 1998.

[12] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366 – 381, Apr. 2000.

[13] G. J. Holzmann. The model checker SPIN. *Software Eng.*, 23(5):279 – 295, 1997.

[14] Jreversepro 1.4.1. `http://jrevpro.sourceforge.net/`.

[15] S. Kamin. Inheritance in smalltalk-80: a denotational definition. In *POPL'88*, pages 80 – 87, 1988.

[16] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[17] Z. Manna and A. Pnueli. *The Temporal Verifciation of Reactive and Concurrent Systems – Specification*. Springer-Verlag Inc., 1992.

[18] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[19] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In *CADE-19*, volume 2741 of *LNCS*, pages 2 – 16, 2003.

[20] J. S. Moore. `http://www.cs.utexas.edu/users/xli/prob/p4/p4.html`.

[21] D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java model checking. In *ASE'01*, pages 253 – 256, 2000.

[22] D. Peled. All from one, one for all: on model checking using representatives. In *CAV'93*, LNCS, pages 409–423, 1993.

[23] J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop "Formal Underpinnings of Java" OOPSLA'98*, Oct. 1998.

[24] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.

[25] M. Stehr and C. Talcott. Plan in Maude: Specifying an active network programming language. In *RTA'02*, volume 71, 2002.

[26] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In *RTA'02*, 2002.

[27] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS'01*, volume 2031 of *LNCS*, pages 299 – 312, 2001.

[28] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.