Formal Analysis of Java Programs in JavaFAN

Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign
{afarzan,fengchen,meseguer,grosu}@cs.uiuc.edu

Abstract. JavaFAN is a Java program analysis framework, that can symbolically execute multithreaded programs, detect safety violations searching through an unbounded state space, and verify finite state programs by explicit state model checking. Both Java language and JVM bytecode analyses are possible. JavaFAN's implementation consists of only 3,000 lines of Maude code, specifying formally the semantics of Java and JVM in rewriting logic and then using the capabilities of Maude for efficient execution, search and LTL model checking of rewriting theories.

1 Introduction

JavaFAN (Java Formal ANalyzer) is a tool to simulate and formally analyze multithreaded Java programs at source code and/or bytecode levels. A novel feature of JavaFAN's design is that it is directly based on *formal definitions* of the Java and the JVM semantics in the form of *rewrite theories* that are efficiently executed and analyzed in the Maude language [2]. The following types of analysis are supported: (1) *symbolic simulation*, with Java and JVM specifications used as *interpreters* executing programs with actual or symbolic inputs; (2) *breadth-first search* (BFS) within a concurrent program's state space to find violations of safety properties; (3) *model checking* of linear temporal logic (LTL) properties for programs whose state space is finite. These forms of analysis are efficiently supported by Maude's underlying rewriting, breath-first search, and LTL model checking features [2]. To keep the framework user-friendly, JavaFAN wraps the Maude specifications and accepts Java or JVM code from the user as input.

JavaFAN's specification-based design has a number of important advantages: (1) formal specifications provide a rigorous semantic definition for a language that can be mathematically scrutinized; (2) such formal specifications can be developed with relatively little effort, even for large languages like Java and the JVM; (3) Maude's underlying formal analysis infrastructure is entirely *generic*, so that formal analysis tools become available *for free* for each language so specified; and (4) in spite of their generality, the formal analyses can be performed with *competitive performance*. Section 3 discusses several examples, providing analysis times and comparisons of JavaFAN with other related analysis tools.

The reason for JavaFAN's efficiency is twofold. On the one hand, Maude has a rewrite engine achieving millions of rewrites per second, an efficient BFS

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 501–505, 2004.
© Springer-Verlag Berlin Heidelberg 2004

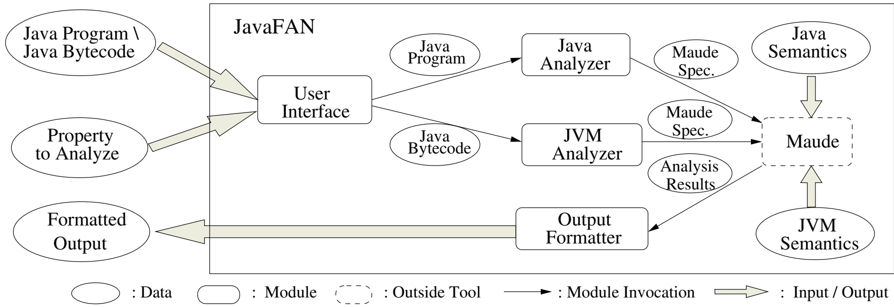

Fig. 1. Architecture of JavaFAN.

algorithm, and an explicit state LTL model checker with performance comparable to SPIN [3]. On the other hand, our approach in specifying the semantics of a concurrent language as a rewrite theory in Maude tries to maximize performance. A *rewrite theory* is a triple (Σ, E, R), with Σ a signature declaring types and function symbols, E a set of equations, and R a set of rewrite rules. Intuitively, the equations E specify the semantics of a language's *deterministic* computations, whereas the rules R specify its *concurrent transitions*. The point is that the *state space* is only defined by the rules R: the smaller R is, the more efficient the analysis. In JavaFAN, R is indeed small compared to E: for Java $|R| = 15$ and $|E| = 600$, and for JVM $|R| = 40$ and $|E| = 300$. A *continuation-based* semantics also increases performance, because most equations and rules then become *unconditional* and thus more efficient to execute. Finally, by distinguishing between the *static* and *dynamic* parts of a program, only the dynamic component is kept in the state representation, with huge resource savings for large programs.

2 Overview of JavaFAN

Figure 1 presents the architecture of JavaFAN. The *user interface* module hides the Maude back-end behind a user-friendly environment. It also plays the role of a dispatcher, sending the Java source code and/or the bytecode to Java and/or JVM analyzers, respectively. The analyzers wrap the input programs into properly defined Maude modules and invoke Maude, which analyzes the code based on formal specifications of the Java language and the JVM. The output formatter collects the output of Maude, transforms it into a user-readable format, and sends it to the user. 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. Native methods and many of the Java built-in libraries are not currently supported. 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.

Table 1. Thread Game Times.

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

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. A state is a multiset of state attributes, such as threads, memory, synchronization information, etc. To support multi-threaded programs, we introduce the notion of *thread context*, which consists of three components: (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 it.

Object-Based Semantics of the JVM. The state of the JVM is represented as a multiset of objects and messages in Maude. 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 each rewrite, there is usually one thread involved, together with classes and/or objects that may be needed to execute the next bytecode instruction.

3 Experiments

Using the underlying search and model checking features of Maude, JavaFAN can be used to formally analyze Java programs. *Breadth-first search analysis* (supported through Maude's `search` command) 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. This empowers JavaFAN to analyze programs with possibly infinite state spaces. For finite state programs, it is also possible to perform explicit-state model checking (using Maude's model checker) of properties specified in linear temporal logic (LTL).

JavaFAN has effectively been applied on a number of examples. Performance results are given in seconds on a 2.4 GHz Linux PC. Detailed discussions on the examples can be found in [4]. Results are given at both Java and JVM levels.

Remote Agent (RA) [5] has two running threads: a *planner* that generates plans from mission goals, and an *executive* that executes the plans. The code

Table 2. Dining Philosophers Times.

Tests	**JVM**	**Java**
DP(5)	4.5	9.9
DP(6)	33.3	81.7
DP(7)	4.4m	15.1m
DP(8)	13.7m	98m
DP(9)	803.2m	—
DF(5)	3.2m	19.2
DF(6)	23.9m	2.4m
DF(7)	686.4m	27m

contains a missing critical section, that leads to a data-race between two concurrent threads, which further caused a deadlock. JavaFAN finds the deadlock in 0.3 of a second in the bytecode level and 0.09 of a second in the source-code level, while the tool in [8] finds it in more than 2 seconds in its most optimized version.

Thread Game [7] is a simple multithreaded program which shows the possible data races between two threads accessing a common variable. Each thread reads the value of the static variable `c` twice and writes the sum of the two values back to `c`. The 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 achieved [7]. JavaFAN (using the `search` command) addresses this question for any specific value of N. Table 1 presents the result for some numbers.

Dining Philosophers. We have model checked a deadlock-prone (DP) and a deadlock-free (DF) version of this problem. The property that we have model checked for this example is whether all the philosophers can eventually dine. The LTL formula is $\Diamond$Check(N), where `N` is the number of philosophers. The proposition `Check(N)` is specified in Maude by the user. For the deadlock-prone version, the model checker generates a counterexample – a sequence of states that leads to the deadlock – and states the property to be true in the deadlock-free version. Currently, JavaFAN can detect the deadlock for up to 8 philosophers at the bytecode level and up to 9 philosophers at the Java Language level in a reasonable amount of time (Table 2). It can also prove the program deadlock-free for up to 7 philosophers at both levels. This compares favorably with JPF [1, 6] which for the same program cannot deal with 4 philosophers.

2-Stage Pipeline is a pipeline computation, where each pipeline stage executes as a separate thread. Stages interact through *connector* objects. The property we have model checked for this program states 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 $\Box(\text{c1stop} \rightarrow \Diamond(\neg\text{stage1return}))$. 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 [8] which without

using the partial order reduction performs the task in more than 100 minutes (both on a 2.4GHz PC).

4 Conclusions and Future Work

We have discussed JavaFAN's design, and experiments suggesting that it can analyze Java and JVM programs with competitive performance. Perhaps the most important experience gained is that a formal specification based methodology to develop formal analysis tools for concurrent programs is a rigorous, cost-effective, and practical approach when realized on a high-performance logical engine. The methodology itself is generally applicable to other languages. Future work will involve further experimentation, development of similar tools for other languages, partial order reduction optimization, and widening the range of formal analyses supported, including program abstraction and theorem proving.

References

1. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *ASE'00*, pages 3 – 12, 2000.
2. 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`.
3. Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc.* 10th *Intl. SPIN Workshop*, volume 2648, pages 230–234. Springer LNCS, 2003.
4. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. JavaFAN. `fsl.cs.uiuc.edu/es/javafan`.
5. 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, August 2001. Previous version appeared in Proceedings of the 4th SPIN workshop, 1998.
6. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366 – 381, April 2000.
7. J. S. Moore. `http://www.cs.utexas.edu/users/xli/prob/p4/p4.html`.
8. D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java model checking. In *ASE'01*, pages 253 – 256, 2000.