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

Feng Chen, Azadeh Farzan, José Meseguer, Grigore Roşu
Department of Computer Science,
University of Illinois at Urbana-Champaign.
{fengchen,afarzan,meseguer,grosu}@cs.uiuc.edu

Abstract. JavaFAN (Java Formal ANalisis) is a multithreaded program analysis framework based on rewriting logic specifications of the semantics of Java. It can currently 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 using a breadth-first strategy, 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.

1 Introduction

There is a general belief in the algebraic specification community that all traditional programming language features can be described with equational specifications [?]. What is less known or tends to be ignored is that concurrency, which is a feature of almost any current programming language, cannot be naturally handled by equational specification unless one makes deterministic restrictions on how the different processes or threads are interleaved. While some of these restrictions may be acceptable, as most programming languages also provide thread or process scheduling algorithms, most of them are unacceptable in practice because concurrent execution typically depends upon the external environment, which is unpredictable. Rewriting logic [?] extends equational logic with rewriting rules and has been mainly introduced as a unified model of concurrency; indeed, 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 analysis techniques specific to the latter to infer concurrency properties about the former. There is already a substantial body of case studies, of which we only mention [?], 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. Further evidence on this claim includes modeling of a wide range of programming language features that has been developed and tested as part of a recent course taught at the University of Illinois [?].
The focus in this paper is on providing a range of formal analysis techniques for Java bytecode multi-threaded programs. Our approach is based on a formal rewriting logic specification of the Java Virtual Machine (JVM) bytecode concurrent semantics, which is executable in the Maude language [?]. This specification, consisting of only 2,000 lines of Maude code, defines both the sequential and the concurrent semantics of Java threads and can be used to perform the following types of formal analysis:

1. Symbolic simulation. JVM code execution can be simulated by executing rewrite rules in the JVM specification; it is also possible to execute in this way code in which some inputs or code fragments are symbolic.

2. Breadth-first formal analysis. Unlike simulation, which explores just one concurrent execution, breadth-first formal analysis can explore all concurrent executions of a possibly infinite-state program. Violations of safety properties can thus be searched for, yielding a complete semi-decision procedure for finding such violations limited only by the available memory and time.

3. LTL Model checking analysis. For finite-state Java programs, this provides a decision procedure to verify linear temporal logic (LTL) properties, again limited by the available memory and time.

These three forms of analysis are a consequence of the underlying features of Maude, applied on the same JVM specification, so they all come for free. Symbolic simulation uses the rewriting engine of Maude, which can perform several million rewrite steps per second, modulo associativity, commutativity and/or identity. Since each rewrite step translates into a JVM execution step, our specification yields an acceptably fast JVM interpreter and symbolic simulator. Search analysis uses the search command, which explores the state-space generated by possibly concurrent applications of rewrite rules in a breadth-first order. Model checking uses Maude’s explicit-state LTL model checker, that has competitive performance [?]. However, to facilitate user interaction we have built a prototype tool called JavaFAN (the Java Formal ANalyzer) that wraps the underlying Maude interpreter and accepts JVM code from a user as input. The user can begin with a Java program, compile it, and then input the resulting JVM code to JavaFAN. Symbolic simulation can be performed in JavaFAN without the user even being aware of the underlying Maude. Since breadth-first search and model checking analyses require specifying the relevant formal properties—state predicates to search for, or LTL formulae involving such predicates—at present such properties have to be specified in the underlying Maude system.

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 [?]; the same HTTP address also contains the Maude JVM semantics, example codes, and a downloadable JavaFAN prototype.

Related Work. There is a vast literature on formal analysis of Java programs (either at the Java or JVM levels) that we cannot exhaustively review here. We can classify the different approaches as focusing on either sequential Java programs or concurrent Java programs. Our work clearly 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 that based on the ACL2 logic and theorem prover [?], and that based on a formal JVM semantics and reasoning based on Abstract State Machines (ASM) [?]. Our approach seems complementary to both of these approaches, 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. Our work also complements the ASM work, in the sense that the formal analyses they perform are *dynamic safety checks*, whereas we perform in a sense stronger analyses through search and model checking.

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) [?], 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. However, note that our approach requires relatively little effort to implement (the current version is a one-semester PhD student work), gives a maximum of flexibility with respect to adding new JVM instructions as the language evolves (there are essentially two lines of Maude code needed per JVM instruction, one to define its syntax and the other for its semantics), and can be easily adapted to other languages (as [?] shows, our current JVM definition follows a generic technique to give rewriting logic semantics to concurrent programming languages). In fact, we consider that despite their particular focus on JVM code, the results presented in this paper should be looked at through the prism of their genericity and language independence, rather than “just another way to implement a JVM”. The point is that Maude’s formal analysis techniques for rewriting logic executable specifications are *generic*, and can be shared by different languages once they are properly formalized in rewriting logic. Pragmatically, this means that one can focus on optimizing and improving the capabilities for simulation, search, model checking, and other formal analyses for rewriting logic specifications, and then get all the resulting benefits for each individual language for free.

Other related work includes [?] which propose an algorithm that takes the bytecode for a method and generates a temporal logic formula that holds if and only if 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 [?], which focuses on dynamic safety checks, and the collective effort around the JML specification language and verification tools for

---

\(^1\) Most of our examples were provided by the JPF team; we especially thank Willem Visser for his prompt help. We are still in the bureaucratic process of obtaining a copy of JPF from NASA, so our claims are based on discussions with JPF members. A thoroughful comparison will be done as soon as we obtain JPF.
sequential Java [?], where formal executable specifications of Java semantics in PVS are used, e.g. see [?], to verify Java programs.

An alternative approach to define analysis tools for Java is to devise language translators, generating from Java code input code for other simpler languages, and then to analyze the generated code. Bandera [?] extracts abstract models from Java programs, specified in several formalisms, including the input language PROMELA of SPIN [?], which can then be further analyzed with specialized existing tools, such as SPIN. JCAT [?] also translates Java into PROMELA. Since SPIN does not support dynamic data structures, fixed-size heaps and stacks need to be allocated, which can lead to natural worries regarding the correctness of the translation. [?] 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, it turns out that, at least in the case of [?], unnecessary overheads are also generated. For example, exactly the same Remote Agent Java code on which JavaFan takes way less than 1 second to analyze at both the source-code and the byte-code levels, takes more than 2 seconds even on the most optimized version of the tool in [?].

2 Background on Rewriting Logic and Maude

In this section we explain the general concurrent object-oriented formal specification method we use, to specify the state of the JVM as a “pool” or “soup” of objects and messages, and to specify the object interactions by rewrite rules. As a whole, the specification is a rewrite theory, that is, a triple $(\Sigma, E, R)$, with $(\Sigma, E)$ an equational specification with signature of operators $\Sigma$ 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 $(\Sigma, E, R)$, and can be applied modulo the equations $E$.

The concurrent state of an object-oriented system, called a configuration, has typically the structure of a multiset made up of objects and messages. Such a multiset can be visualized as a “pool” or “soup” in which different objects and messages are freely floating and can interact with each other. We can view configurations as built up by a binary multiset union operator which we can represent with empty syntax (i.e., juxtaposition) as $\_\_$. (Following the conventions of mix-fix notation, underscore symbols $\_$ are used to indicate argument positions.) The operator $\_\_$ is declared to satisfy the structural laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union. 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.
Objects belong to classes, which can be specified in Maude with special syntax. For example, a Buffer class defines objects that store a list of numbers in their q attribute. Each buffer is read by an object whose name is stored in its reader attribute. The Maude syntax for such a class is

```
class Buffer | q: List[Nat], reader: Oid .
```

where Oid is the sort of object identifiers, and List[Nat] is the sort of lists of natural numbers. Objects, classes, and messages can all be axiomatized by an equational theory \((\Sigma, E)\) describing the concurrent states of a given object system. The rewrite rules \(R\) then describe the concurrent transitions of the system. They can either involve several objects, or a single object communicating through message passing. Their general form is,

```
\begin{align*}
  r & : M_1 \ldots M_n \langle O_1 : F_1 | \text{atts}_1 \rangle \ldots \langle O_m : F_m | \text{atts}_m \rangle \\
  & \rightarrow \langle O_{i_1} : F'_{i_1} | \text{atts}'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} | \text{atts}'_{i_k} \rangle \\
  & \langle Q_1 : D_1 | \text{atts}''_1 \rangle \ldots \langle Q_p : D_p | \text{atts}''_p \rangle \\
  & M'_1 \ldots M'_q \text{ if } C
\end{align*}
```

where \(r\) is the label, the \(M_s\) are message expressions, \(i_1, \ldots, i_k\) are different numbers among the original 1, \ldots, \(m\), and \(C\) is the rule’s condition. That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created. Concurrent rewriting with the rules \(R\) then provides a “truly concurrent” model of computation for object systems [7]. Under reasonable assumptions about the rewrite theory \((\Sigma, E, R)\), rules can be efficiently executed. Maude’s object-oriented modules [?] support the syntax conventions mentioned above. In addition, such rewrite theories \((\Sigma, E, R)\) can be formally analyzed in Maude by means of its search command, or, for finite-state systems, by its underlying LTL model checker [?].

3 Semantics

We use Maude to specify the operational semantics of a sufficiently large subset of the Java language and of the JVM. We support 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.

Java and the JVM are modeled differently. For Java, a quite efficient continuation-based style is adopted. 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 RWL (i.e. equations and rewrite rules) to specify the changes to the program state. Our major design goal is to reduce the size and number of system’s state. The former has been achieved through separating the static and dynamic parts of the program, and maintain only the dynamic part in the system’s state. For the latter, we reduced the number of rewrite rules in the specification. See Section 3.4 for a detailed discussion on these major optimizations.
3.1 Continuation-based Semantics of Java

The semantics of the Java language is defined in a modular way. Different features of the language are modeled in separate modules to ease extensions and maintenance. The specification contains 75 modules, around 1,200 lines of code including around 600+ equations and 15 rewriting rules [?].

```
subsort StateAttribute < State .
op c : Context -> StateAttribute . *** Thread Context
op s : 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
```

Fig. 1. Java Program State.

**States of the Program.** We adopt the well-known idea of representing program states for object-oriented languages [?], in which the environment is used to map the names of variables to their locations in the store, and the store is used to map locations to values. Objects are stored as environments along with their class types. In order to support multi-threaded programs, we have introduced the notion of thread context which contains the environment (see Figure 1). The thread context consists of three elements (Figure 2): (1) a continuation, (2) the thread environment, and (3) the corresponding object. The continuation keeps track of the control context of the thread, which explicitly specifies the next step. For a more detailed discussion on continuations, see the next Section.

```
subsort ContextItem < Context .
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
```

Fig. 2. Java Thread Context

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.

**Continuations.** The straightforward semantics of addition can be written using one conditional equation as follows.

```
ceq eval(E1 + E2) = v1 + v2 if v1 := eval(E1) \land v2 := eval(E2) .
```

The expressions E1 and E2 are evaluated first, and then the integer results are added to compute the result. This approach is easy to understand, but the interpreter has to maintain the control context implicitly, which results in unnecessary big system states in some cases. Besides, the application of conditional equations is more expensive compared to unconditional ones in Maude, as the result of storing and recovering information. Instead, continuations explicitly store the control context of the program. The addition operation can be specified using continuations as follows.
K is the remaining part of the continuation. When the expressions on the top of the continuation \((E_1, E_2)\) are evaluated, the results will be passed to the remaining continuation. Moreover, no conditional equations are required in the continuation-based specification.

Continuations provide a natural way to implement threads. A continuation is used to store the execution flow of a thread. One continuation at a time is picked up nondeterministically to be executed by the interpreter. Continuations also facilitate the definition of flow-control instructions, e.g. `break`, `continue`, `return`, and exceptions.

**Semantics of the Java Language.** In this section we present one examples of how the Java language statements are modeled. The formal semantics of the Java language is defined based on the informal descriptions from [7].

\[
\begin{align*}
\text{eq } & k((X += E) \rightarrow K) = k((\text{loc}(X) \rightarrow += (E) \rightarrow K)) . \\
\text{rl } & c(k(L \rightarrow += (E) \rightarrow K), \text{context}), m([L,V] M) \Rightarrow \\
& c(k([E | V] \rightarrow += (\text{set&fetch}(L) \rightarrow K)), \text{context}), m([L,V] M) .
\end{align*}
\]

**Fig. 3.** Semantics of `+=` Expression

Figure 3 shows the semantics of `+=` operator \((X += E)\) in Java. In the equation, the memory location of the left operand is computed using operation \(\text{loc}(X)\) and together with the `+=` operator and the right operand are put on top of the continuation. Having the location \(L\), the operation `+=`, and the expression \(E\) on top of the computation, the rewrite rule creates three new tasks for the continuation: (1) Computing the expression \(E\) and add it to the value \(V\) (value associated to location \(L\)) which is done through \([E | V] \rightarrow +=\); (2) write the result from part 1 to location \(L\) and also pass to the remaining continuation \(K\) which is done through `set&fetch(L)`. Since we are accessing memory location \(L\), this has to be done through a rewrite rule (see Section 3.4).

### 3.2 Object-based Semantics of the JVM

Our JVM model covers 150 out of 250 bytecode instructions in about 2000 lines of Maude code, including around 300 equations and 40 rewrite rules. To begin with, we describe the representation of states in our model. The state of the JVM is represented as a pool (multiset) of objects and messages in Maude [7]. Rewrites (with rewrite rules and equations) in this pool (modulo associativity, commutativity, and identity) model the changes in the state of the JVM.

The JVM, as an abstract machine, has four basic components: (1) the Java class space, (2) the thread space, (3) the heap, and (4) the state transition machine, updating the internal state at each step.

In our model, no specific entity plays the role of the state transition system, and the strict separation of the classes, threads, and objects no longer exists.
Instead, the pool contains them all, and interactions between them corresponds to state changes. Objects in the pool 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. These categories of objects and their interactions are described in Section 3.2.

The Pool

Java Objects are modeled by an object in the pool containing four attributes.

\[
\text{< O:JavaObject | Addr:HeapAddress, FieldValues:FieldValues, CName:ClassName, Lock:Lock >}
\]

(1) the heap address at which the object is stored, (2) list of all instance fields and their values \(^2\), (3) Object’s class name, and (4) lock associated with the object.

Java Threads are modeled by an object with three attributes.

\[
\text{< T:JavaThread | callStack: CallStack, Status: CallStackStat, ORef: HeapAddress >}
\]

(1) a stack of frames similar to a run-time call stack, (2) a flag indicating the scheduling status of the thread, and (3) the address of the object to which the thread is associated. The call stack is a stack of frames, where each frame models the activation record of a method call. Therefore, at any time, the top frame corresponds to the activation record of the method currently being executed. A frame is a tuple defined as follows.

\[
s \in \text{op} \cdot \text{IntInstLabeledPgmLocalVarsOperandStackSyncFlagClassName} \rightarrow \text{Frame}.
\]

This tuple contains the program counter, the instruction being executed, the code of the method being executed, a list of values of the local variables, the operand stack of the local computations, a flag to indicate whether the invocation has locked the corresponding class, or the corresponding object, or nothing at all, and finally the name of the class from which the method has been invoked.

Java Classes are divided into static and dynamic parts (see the beginning of Section 3), represented by JavaClassC and JavaClassV objects respectively. The dynamic part contains three attributes: (1) the constant pool of the Java class file, (2) the class lock, and (3) the values of the static fields which has the same structure as the FieldValues in Java objects.

\[
\text{< C:JavaClassC | SupClass:ClassName, StaticFields:FlatFNL, Fields:FlatFNL, Methods:MethodList >}
\]

\[
\text{< C':JavaClassV | ConstPool:ConstantPool, Lock:Lock, StaticFieldValues:FieldPairList >}
\]

\(^2\) Note that a single field may have more than one value, depending on its appearance in more than one class in the hierarchy of superclasses of the Java class from which the object is instantiated; therefore, in this list we store all the fields and their values separately for each class in the hierarchy of the superclasses of the object’s class to support shadowing.
The static part contains four attributes: (1) the name of immediate superclass, (2) a list of static fields for all the classes in the hierarchy of superclasses of this class\(^3\), (3) a list of instance fields with the same structure as the list of static fields, and (4) a list of methods.

**Auxiliary Objects.** Several objects floating in the pool do not belong to any of the above categories. They have been added for definitional/implementation purposes. Examples include: (1) an object collecting the outputs of the threads. This object contains a list of values. When a thread wants to print a value, it adds this value to the end of this list; (2) a heap manager, that maintains the last address being used on the heap (we do not model garbage collection at the moment, but a modification of the heap manager can add garbage collection to our current JVM definition); (3) there are several Java built-in classes that had to be defined apriori. Support for input/output, creation of new threads, and `wait/notify` facilities are among the most important ones.

**Interactions in the Pool.** Interactions between the above categories of objects simulate the program execution. They can interact directly or by passing messages to each other. In most rewrites, there is always a Java thread involved, and based on the bytecode instruction being executed, the thread interacts with Java objects, Java classes, or auxiliary objects to complete the execution. See the following section for examples of these interactions.

**Bytecode Semantics**

In this section we present some examples of how the bytecode instructions are modeled in our system. We hope these examples can give the reader a flavor of our specification. The entire Maude representation and a collection of examples can be found in [?]. We have defined the formal semantics of each instruction based on the informal description from [?].

The `iadd` instruction is executed inside the thread and no interaction with the outside is needed. It is modeled by the equation shown in Figure 4. The two values \( I \) and \( J \) on top of the operand stack are popped, and the sum \( I + J \) is pushed. The program counter is moved forward by the size of the `iadd` instruction to reach the beginning offset of the next instruction. The current instruction (which was `iadd` before) is also changed to be the next instruction in the current method code. Everything else remains the same.

\[
eq < T : \text{JavaThread} \mid \text{callStack}: \{\text{PC}, \text{iadd}, \text{Pgm}, \text{LocalVars}, (I \# J \# \text{OperandStack}), \text{SyncFlag}, \text{ClassName}\} \text{CallStack}, \text{Status}: \text{scheduled}, \text{ORef} : \text{OR} >
\]

\[
= < T : \text{JavaThread} \mid \text{callStack}: \{\text{PC} + \text{size}(\text{Pgm}[\text{PC}]), \text{Pgm}[\text{PC} + \text{size}(\text{Pgm}[\text{PC}])], \text{Pgm}, \text{LocalVars}, (I + J \# \text{OperandStack}), \text{SyncFlag}, \text{ClassName}\} \text{CallStack}, \text{Status}: \text{scheduled}, \text{ORef}: \text{OR} > .
\]

**Fig. 4.** The `iadd` instruction

\(^3\) compiled in a preprocessing phase form the class information for efficiency purposes
**getfield** instruction, which fetches the value of an object field, is modeled by the rule in Figure 5. One thread and two Java classes are involved in its execution. The I operand is an index to the constant pool of the Java class `ClassName` referring to the field information ([I, [ClName, fieldname]]), namely, field's name, and the name of the class of the field. The Java object O whose reference REF(K) is on top of the operand stack (note that the value of attribute Addr of O is K as well) is the desired object. On the right hand side of the rule, the value of the indicated field of object O is placed on top of the operand stack of the thread (FV[ClName, FieldName] # OperandStack).

3.3 Synchronization

We support three means of synchronization in our model: (1) synchronized sections, (2) synchronized methods, and (3) `wait`/`notifyAll` methods from Java built-in class `Object`.

For the permission to enter a synchronized section, the lock of the corresponding object has to be acquired first, and then be released at the exit point. Figure 6 presents how a thread acquires the lock of an object at the Java language level. `wl(obj)` and `endSyn(obj)` mark the entrance and exit of the synchronized section bl. The status of lock of obj will change from `unlocked` to `locked` as the result.

```
rl1 [GETFIELD] : < T : JavaThread | callStack : ([PC, getfield(I)], Pgm, LocalVars, REF(K) # OperandStack, SyncFlag, ClassName] CallStack), Status: scheduled, ORRef: OR > < ClassName : JavaClassV | ConstPool : ([I, [ClName, FieldName]] ConstantPool), REST > < O : JavaObject | Addr: K, FieldValues: FV, REST' > => < T : JavaThread | callStack : ([PC + size(Pgm[PC]), Pgm[PC + size(Pgm[PC])]], Pgm, LocalVars, FV[ClName, FieldName] # OperandStack, SyncFlag, ClassName] CallStack), Status: scheduled, ORRef: OR > < ClassName : JavaClassV | ConstPool : ([I, [ClName, FieldName]] ConstantPool), REST > < O : JavaObject | Addr: K, FieldValues: FV, REST' > .
```

**Fig. 5. getfield Instruction**

```
rl c(k(obj -> synchronized(bl) -> K), context), l([obj, Ls] Ll) = r1 c(k(wl(obj) -> synchronized(bl) -> K), context), l([obj, unlocked] Ll) => c(k(bl -> endSyn(obj) -> K), context), l([obj, locked] Ll) .
```

**Fig. 6. wait for lock semantics**

Java synchronized sections are translated into sections surrounded by `monitorenter` and `monitorexit` bytecode instructions, which acquire and release the lock of the object on top of the operand stack respectively. Figure 7 presents the semantics for `monitorenter` at the bytecode level. The lock of each Java object has three components: (1) a list of identifiers of threads that have waited on this lock, (2) the identifier of the thread that currently owns the lock, and (3) an integer indicating how many times the owner has acquired the lock.
In Figure 7, the heap address of the object \((K)\) is matched with the address on top of the operand stack of the thread \((\text{REF}(K))\). The lock is marked to belong to this thread \((\text{Lock}(\text{OIL, T, 1}))\), and the thread can proceed to the critical section. The counterpart, \text{monitorexit} decreases the counter to 0 and removes its association to the thread.

The \textit{synchronized} methods are modeled in a very similar way; the lock of the object from which the method is being invoked is acquired and released at method invocation and return points, respectively.

The thread calling \text{wait} and \text{notifyAll} methods should already own the lock of the corresponding object. When \text{wait} is called, the thread releases the object lock and goes to sleep, until notified by some other thread. Several threads can wait on the same object lock. When \text{notifyAll} is called, the threads waiting on the lock are awakened. They all compete to acquire the lock to continue their execution. In our system, the winner is chosen nondeterministically, while a variety of algorithms are used in the existing implementations of the JVM to choose the winner deterministically.

At the Java level, we use a special continuation operation \text{wait*} to show that the thread is in waiting mode. If the thread receives a notify notice at this level, its status is marked to be \textit{ready}. A rewrite rule lets all the threads with \textit{ready} status compete to acquire the lock and continue their execution.

At the bytecode level, the thread executing \text{wait} adds its identifier to the list of thread identifiers waiting on the corresponding lock, marks the lock as released, and saves the number of times it has acquired the lock locally. When some other thread calls \text{notifyAll} on the same lock, a (broadcast) message is created containing the list of all threads which have waited on the lock up to that point. This message will then be consumed by all these threads, causing them to wake up and compete for the lock.

### 3.4 Optimizations

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

**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. Therefore, the attributes of each class are grouped into dynamic and static
attributes. The former group appears as an object in the pool (system’s state),
and the latter group is kept outside the pool, in a constant accessed through
auxiliary operations.

Rules vs. Equations. We have used both rewrite rules and equations in our
specification. The rewrite rules provide support for modeling concurrent systems,
while equations are used for the sequential part of the execution as well as for the
supporting infrastructure. In our system, the transition system for the formal
analysis (see Section 4) is built based on the rewrite rules, while equations are
applied automatically to reach canonical forms. Therefore, reducing the num-
ber of rewrite rules by changing them to equations saved us a large number of
states in the transition system. However, this reduction should only be done if
it respects the concurrency. The idea is that only two kinds of tasks have to be
modeled by rules; the only two cases in which the thread interacts with (possibly
changes) the environment: (1) shared memory access, and (2) acquiring locks.
Examples of the former include the semantics of += at Java language level (see
Section 3.1) and the semantics of the instruction getfield (see Section 3.2) at
the bytecode level. As an example for the latter case (see Section 3.3), we refer
the reader to semantics of acquire a lock at the Java language level, and the
semantics of the monitorenter instruction which performs the same task at the
bytecode level.

It is hard to distinguish between local memory access and shared memory
access at the Java language level. Therefore, in our specification every mem-
ory access is modeled by a rewrite rule at that level. At the bytecode level,
nevertheless, different instructions are used to access local and shared memory.
This knowledge gives us the advantage of using rewrite rules only for the shared
memory accesses and using equations for the local ones.

4 Formal Analysis

Using the underlying search and model checking features of Maude, JavaFAN
can be used to formally analyze Java programs at both the source code and the
bytecode levels. Breadth-first search analysis is a semi-decision procedure that
can be used to explore all the concurrent computations of a program looking for
bugs 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
that can be used as simulators and also for formal analysis purposes. To facilitate
user interaction, Java and the JVM semantic specifications are integrated within
a prototype tool, called JavaFAN, that accepts either Java programs or standard
bytecode as its inputs.

For Java programs we have a wrapper to reformat the programs to be rec-
ognizable by Maude, since the grammar of Java causes some parsing difficulties
in Maude.

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. We also use another disassembler
jreverserpro [?] to extract the constant pool information that javap does not
provide.

4.2 Search

Using the simulator (Section 4.1), one can explore only one possible trace (mod-
eled as sequence of rewrites) of the Java program being executed. Maude’s
search command allows exhaustively exploring the different traces of a Java
program to find reachable states according to a given pattern and a condition.
As already mentioned, the breadth-first nature of the search command allows
us to find errors even in infinite state spaces, being limited only by the available
memory. Below, we discuss a number of case studies.

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 [?]. 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 to testing. This justifies formal analysis
techniques like the ones presented in this paper which could have caught that
terror.

The RA consists of three components (see Figure 8 in the appendix): a Plan-
der 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 vul-
nerable to multithreading errors. Events and tasks are two major components
illustrated. In order to catch the events that occur while tasks are executing,
each event has an associated event counter that is 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 [1,2]. Using the search capability of our system, we also found the deadlock in the same faulty copy. It takes the JavaFAN 0.3 of a second to find it at the bytecode level and 0.09 of a second to find it at the Java language level, on a standard PC platform running Linux.

The Thread Game. The Thread Game [?] is a simple multithreaded program which shows the possible data races between two threads accessing a common variable (see Figure 9 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 [?].

We use Maude’s search command to address this question for a specific value of N. We consider some positive integer N and check whether c can hold value N. The search command can find one or all existing solutions (sequences) that lead to get the value N. We have tried numbers up to 1000 where for all of them a solution is found in a reasonable amount time (Table 1).

<table>
<thead>
<tr>
<th>Tests</th>
<th>JVM</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>TG(100)</td>
<td>16.1s</td>
<td>6.6s</td>
</tr>
<tr>
<td>TG(200)</td>
<td>39.9s</td>
<td>17s</td>
</tr>
<tr>
<td>TG(500)</td>
<td>4.1m</td>
<td>2.0m</td>
</tr>
<tr>
<td>TG(1000)</td>
<td>10.1m</td>
<td>5.1m</td>
</tr>
</tbody>
</table>

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 used on the Maude specifications of Java and JVM’s concurrent semantics. This way, we obtain a model checking procedure for Java at both levels 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 example.

Dining Philosophers. The code for the version of Dining Philosophers that we have used, can be found in Figure 10 in the appendix. The property that we have model checked for this example is whether the first 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. The LTL formula can be built based on propositions that the user can define in Maude as follows.

\[
\text{op Check : Oid Int -> Prop .}
\]

In the above description, \( \text{Check(ObjectId, N)} \) will be true at some state if \( \text{ObjectId} \)'s output list (in this case the object collecting the output) contains all the numbers from 1 to \( N \). In the special case mentioned above, we are going to check the following LTL formula using the \text{modelCheck} command

\[
\text{red modelCheck(InitialState, <> Check('output, 1)) .}
\]

where \( \text{InitialState} \) is the initial state of the program, which is defined separately and automatically.

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 8 philosophers at the bytecode level and up to 9 philosophers at the Java Language level in a reasonable amount of time (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 for up to 6 philosophers at the bytecode level and up to 7 philosophers at Java language level. This compares favorably with JPF which for the same program can not deal with 4 philosophers [?].

Table 2. Dining Philosophers Times

<table>
<thead>
<tr>
<th>Tests</th>
<th>JVM</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(2)</td>
<td>0.14s</td>
<td></td>
</tr>
<tr>
<td>DP(3)</td>
<td>0.30s</td>
<td></td>
</tr>
<tr>
<td>DP(4)</td>
<td>0.64s</td>
<td>1.2s</td>
</tr>
<tr>
<td>DP(5)</td>
<td>4.5s</td>
<td>9.9s</td>
</tr>
<tr>
<td>DP(6)</td>
<td>33.3s</td>
<td>87.1s</td>
</tr>
<tr>
<td>DP(7)</td>
<td>4.47m</td>
<td>15.1m</td>
</tr>
<tr>
<td>DP(8)</td>
<td>13.74m</td>
<td>98m</td>
</tr>
<tr>
<td>DP(9)</td>
<td>803.17s</td>
<td></td>
</tr>
<tr>
<td>CDP(2)</td>
<td>0.4s</td>
<td></td>
</tr>
<tr>
<td>CDP(3)</td>
<td>3.2s</td>
<td></td>
</tr>
<tr>
<td>CDP(4)</td>
<td>21.5s</td>
<td>2.48s</td>
</tr>
<tr>
<td>CDP(5)</td>
<td>3.22m</td>
<td>19.2s</td>
</tr>
<tr>
<td>CDP(6)</td>
<td>23.93m</td>
<td>2.4m</td>
</tr>
<tr>
<td>CDP(7)</td>
<td>27m</td>
<td></td>
</tr>
</tbody>
</table>
5 Conclusion and Future Work

We have presented JavaFAN, have explained its underlying formal executable specification of the concurrent semantics of the JVM, and have illustrated its use in formally analyzing JVM code by simulation, search, and model checking. We have also explained how it complements other approaches. Although the results so far seem encouraging, much work remains ahead. One important issue that needs to be addressed is how to scale up to bigger programs in the face of the combinatorial explosion inherent in both breadth-first search and model checking. 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 [?], as well as the experience in using abstraction to model check programs in work such as, e.g., [?] should be exploited in this regard. Partial-order reduction [?] techniques in the context of rewriting logic model checking is certainly an interesting and promising area of further research. With respect to JVM, a non-trivial issue is also to properly deal with foreign functions, because most of the Java libraries are in fact implemented in C. One way to do it is to follow JPF and consider those atomic; however, this would imply that Maude should be able to call foreign functions during its analysis process, which is a feature to be added soon to Maude. Another way to do it is to define several assembler languages for different processors as Maude modules, and to actually interpret foreign object code as well; though doable, this looks like a rather heavy solution.

Another issue for 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 of the spectrum, it would be interesting to incorporate in JavaFAN domain-specific certification and specification-based monitoring techniques such as those proposed in [?],[?], so that domain knowledge and suitable program annotations can be used to certify domain-specific properties, and so that programs that are not fully verified can be monitored at runtime with respect to relevant formal properties.

References


38. W. Visser. private commmunication.
Appendix
class Event {
    int count = 0;
    public synchronized void wait_for_event() {
        try { wait(); } catch(InterruptedException e){ };
    }
    public synchronized void signal_event() {
        count = (count + 1) % 3;
        notifyAll();
    }
}

class Planner extends Thread {
    Event event1, event2;
    int count = 0;
    public Planner(Event e1, Event e2) {
        this.event1 = e1;
        this.event2 = e2;
    }
    public void run() {
        int count = 0;
        while(true) {
            if (count == event1.count)
                event1.wait_for_event();
            count = event1.count;
            event2.signal_event();
        }
    }
}

class Executive extends Thread {
    Event event1, event2;
    int count = 0;
    public Executive(Event e1, Event e2) {
        this.event1 = e1;
        this.event2 = e2;
    }
    public void run() {
        int count = 0;
        while(true) {
            event1.signal_event();
            if (count == event2.count)
                event2.wait_for_event();
            count = event2.count;
        }
    }
}

Fig. 8. Part of Remote Agent’s code.
class Process extends Thread {
    static int c;
    public Process(int i) {
        c = i;
    }
    public void run() {
        int a, b;
        while (true)
        c = c + c;
    }
}

Fig. 9. The Thread Game

class DiningPhilosophers2 {
    public static void main(String[] args) {
        Fork F1 = new Fork();
        Fork F2 = new Fork();
        Fork F3 = new Fork();
        Fork F4 = new Fork();
        new Philosopher(1, F1, F2).start();
        new Philosopher(2, F2, F3).start();
        new Philosopher(3, F3, F4).start();
        new Philosopher(4, F4, F1).start();
        return;
    }
}
class Philosopher extends Thread {
    int id;
    Fork F1, F2;
    public Philosopher(int i, Fork f1, Fork f2) {
        this.F1 = f1;
        this.F2 = f2;
        this.id = i;
        return;
    }
    void Dine() {
        System.out.print(id);
        return;
    }
    public void run() {
        synchronized (F1) {
            synchronized (F2) {
                Dine();
            }
        }
        return;
    }
}
class Fork { public int num; }

Fig. 10. Dining Philosophers