Formal JVM Code Analysis in JavaFAN

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

Abstract. JavaFAN uses a Maude rewriting logic specification of the JVM semantics as the basis of a software analysis tool with competitive performance. It supports formal analysis of concurrent JVM programs by means of symbolic simulation, breadth-first search, and LTL model checking. We discuss JavaFAN's executable formal specification of the JVM, illustrate its formal analysis capabilities using several case studies, and compare its performance with similar Java analysis tools.

1 Introduction

There is a general belief in the algebraic specification community that all traditional programming language features can be described with equational specifications [2, 9, 29]. 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 specifications, 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 [17] 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 those definitions to build formal analysis tools for such languages. There is already a substantial body of case studies, of which we only mention [25, 24, 28], backing up 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 [22]. In this paper we give detailed evidence for a second key claim, namely that rewriting logic specifications can be used *in practice* to build simulators and formal analysis tools for mainstream programming languages such as Java with competitive performance. Here, we focus on Java's bytecode, but our methodology is general and can be applied also to the Java source code level and to many other languages.

The JavaFAN (Java Formal Analyzer) tool specifies the semantics of the most commonly used JVM bytecode instructions (150 out of the 250 total) as a Maude module specifying a rewrite theory $T_{\mathrm{JVM}} = (\Sigma_{\mathrm{JVM}}, E_{\mathrm{JVM}}, R_{\mathrm{JVM}})$, where $(\Sigma_{\mathrm{JVM}}, E_{\mathrm{JVM}})$ is an equational theory giving an algebraic semantics with semantic equations E_{JVM} to the *deterministic* JVM instructions, whereas R_{JVM} is a set of *rewrite rules*, with concurrent transition semantics, specifying the behavior of all *concurrent* JVM instructions. The three kinds of formal analysis currently supported in JavaFAN are: (1) *symbolic simulation*, where the theory T_{JVM} is executed in Maude as a JVM intepreter supporting fair execution and allowing some input values to be symbolic; (2) *breadth-first search*, where the entire, possibly infinite, state space of a program is explored starting from its initial state using Maude's `search` command to find safety property violations; and (3) *model checking*, where if a program's set of reachable states is finite, linear time temporal logic (LTL) properties are verified using Maude's LTL model checker.

A remarkable fact is that, as we explain in Section 4, even though T_{JVM} gives indeed a *mathematical semantics* to the JVM, it becomes the basis of a formal analysis tool whose performance is *competitive* and in some cases surpasses that of other Java analysis tools. The reasons for this are twofold. On the one hand, Maude [3] is a high-performance logical engine, achieving millions of rewrites per second on real applications, efficiently supporting search, and performing model checking with performance similar to that of SPIN [13]. On the other, the algebraic specification of system states, as well as the equations E_{JVM} and rules R_{JVM}, have been *optimized* for performance through several techniques explained in Section 3.5, including keeping only the dynamic parts of the state explicitly in the state representation, and making most equations and rules *unconditional*. In this regard, rewriting logic's distinction between the equations E_{JVM} and the rules R_{JVM} has a crucial performance impact in drastically reducing the sate space size. The point is that rewriting with the rules R_{JVM} takes place *modulo* the equations E_{JVM}, and therefore only the rules R_{JVM} affect state space size. Our experience in specifying the JVM in rewriting logic is that we gain the best benefits from algebraic (equations) and SOS [20] (Rules) paradigms in a combined way, while being able to distinguish between deterministic and concurrent features in a way not possible in either SOS or algebraic semantics.

Related Work. The different approaches to formal analysis for Java can be classified 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 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 [15], and another based on a formal JVM semantics and reasoning based on Abstract State Machines (ASM) [23]. 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) [1, 12], 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 the same or even larger size. Other related work includes [21], 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 related approaches include the work on defensive JVM [5], and the collective effort around the JML specification language and verification tools for sequential Java, e.g. [16, 26].

Another approach to define analysis tools for Java is based on *language translators*, generating simpler language code from Java programs and then analyzing them later. Bandera [6] extracts abstract models from Java programs, specified in different formalisms, such as PROMELA, which can be further analyzed with specialized tools such as SPIN. JCAT [7] also translates Java into PROMELA. [19] 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 [19]; for example, exactly the same Remote Agent Java code that can be analyzed in 0.3 second in JavaFAN [8] takes more than 2 seconds even on the most optimized version of the tool in [19].

In section 2 we present a brief background on Maude's methodology. A detailed description of our model is given in 3. In Section 4, we present the various kinds of formal analysis done for the Java programs together with the performance results for several case studies. Finally, Section 5 presents the conclusion and future work.

2 Rewriting Logic, Maude and its Object Methodology

Here we briefly explain our methodology to specify the state of a concurrent system, in this case focusing on the JVM, as a "pool" or "soup" of objects whose interaction is modeled by rewrite rules. As a whole, the specification of the JVM is a *rewrite theory*, that is, a triple $(\Sigma_{\mathrm{JVM}}, E_{\mathrm{JVM}}, R_{\mathrm{JVM}})$, with Σ_{JVM} a signature of operators, E_{JVM} a set of equational axioms, and R_{JVM} a collection of labeled Σ_{JVM}-rewrite rules. The equations describe the *static* structure of the JVM's state space as an algebraic data type, as well as the operational semantics of its deterministic features. The *concurrent transitions* that can occur in different threads are described by the rules R_{JVM}. Arbitrary interleavings of rewrite rules are possible, leading to different concurrent computations of a multithreaded JVM program. The rewriting rules R_{JVM} are applied *modulo* the

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

equations E_{JVM}. Important equations are those of *associativity*, *commutativity* and *identity* (ACU) of binary operations — such as the multiset union operation that builds up the "soup" of objects — allowing us to effectively define the state infrastructure of the JVM. Even though we focus on the algebraic definition of the JVM in this paper, the same methodology has been used to define the Java language as well as several other programming languages [8, 22].

Maude [3] supports, executes, and formally analyzes rewriting logic theories, via a series of efficient algorithms for term rewriting, state-space breadth-first search, and linear temporal logic (LTL) model checking. Once the JVM is formally specified as a rewrite theory in Maude, the above provide us with JVM program analysis tools at no additional cost, capable of performing fair interpretation and simulation, potentially infinite state-space exploration for detecting safety violations, as well as LTL model-checking of JVM multithreaded programs. E_{JVM} contains associativity, commutativity and identity equational axioms to represent the concurrent state of the JVM computation as a *multiset* of entities such as the threads, Java classes, Java objects, etc. Following a well-established methodology in rewriting logic, for which Maude provides generic support [3], we call these entities *objects*. To avoid terminology confusion with Java objects, we may sometimes call them *Maude objects*. Unless differently specified, from now on by "object" we mean a "Maude object". Maude supports a fully generic object-oriented specification environment, where one can define classes and then objects as instances of classes. Aiming at a maximum of efficiency for our JVM analysis tools, we decided *not* to use Maude's generic OO meta-level framework and, instead, to define a minimal object infrastructure at the core level. As a consequence, we have dropped the generic definition of classes, "hardwiring" our object types according to the JVM language.

Most of our equations and rules are applied modulo ACU, which in Maude is a highly optimized and efficient process. For example, Figures 3 and 4 present typical object-oriented rewrite rules. An object in a given state is formally 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 Rewriting Logic Semantics of the JVM

We use Maude to specify the operational semantics of a sufficiently large subset of JVM bytecode. This includes 150 out of 250 bytecode instructions, defined in about 2000 lines of Maude code, including around 300 equations and 40 rewrite rules. We support multithreading, dynamic thread and object creation, virtual functions, recursive functions, inheritance, and polymorphism. Exception handling, garbage collection, native methods and many of the Java built-in libraries are not supported in the current version. The formal semantics of each instruction is defined based on the informal description of JVM in [27]. Section 3.2 explains the operational semantics of the deterministic part of the JVM given by the 300 equations in E_{JVM}, and Section 3.3 discusses the semantics of the concurrent part of JVM specified by the 40 rewrite rules in R_{JVM}.

3.1 Algebraic Representation of the JVM State

Here, we describe the representation of *states* in our model. Our major design goal has been to reduce the size and the number of system states to improve the performance of the formal analysis. The reduction in size has been achieved through separating the *static* and *dynamic* aspects of the program, maintaining only the dynamic part in the system's state. To reduce the number of states, we keep the number of rewrite rules in the specification minimal. A detailed discussion on these optimizations is given in Section 3.5.

The JVM has four basic components: (1) the 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 state of the JVM is represented as a multiset of objects and messages[2] in Maude [3]. Rewrites (with rewrite rules and equations) model the changes in the state of the JVM.

Elements of the multiset. Objects in the multiset fall into four categories:

1. Maude objects which represent Java objects,
2. Maude objects which represent Java threads,
3. Maude objects which represent Java classes, and
4. auxiliary Maude objects used mostly for definitional purposes.

Below, we discuss each in detail.

Java Objects are modeled by objects containing the following attributes.

```
< O:JavaObject | Addr:HeapAddress, FieldValues:FieldValues, CName:ClassName, Lock:Lock >
```

The `Addr` attribute refers to the heap address at which the object is stored. Physical heap addresses are employed only because they are used in the bytecode to refer to objects. The `FieldValues` attribute contains all instance fields and their values. 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. The sort `FieldValues` is a list of pairs, with each pair consisting of a class name and a list. The latter list by itself consists of pairs of field names and field values. Therefore, based on the current class of the object, we can extract the right value for a desired field. The `CName` attribute holds the name of the object's class. The `Lock` attribute holds the lock associated with the object.

Java Threads are modeled by objects with the following attributes.

```
< T : JavaThread | callStack: CallStack, Status: CallStackStat, ORef: HeapAddress >
```

[2] Messages are used as a method to define the semantics in our model. One can use a somewhat different approach which does not include any messages.

The `callStack` attribute models the runtime stack of threads in Java. It is a stack of `frame`s, 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.

```
op [_,_,_,_,_,_,_] : Int Inst LabeledPgm LocalVars OperandStack SyncFlag ClassName -> Frame .
```

The first component is an integer representing the program counter. The second component is the next instruction of the thread to be executed. The third component is a complete list of the instructions of the method, along with their corresponding offsets. The fourth component is the list of the current values of the local variables of the method. The fifth component contains the current operand stack, which carries instruction arguments and results. The sixth component is a flag indicating whether the call of the current method has locked the corresponding class (`SLOCKED`) or the corresponding object (`LOCKED`) or nothing at all (`UNLOCKED`). The last component represents the class from which the method has been invoked.

The `Status` attribute is a flag indicating the scheduling status of the thread: `scheduled` when the thread is ready to execute the next instruction, or `waiting` otherwise. Examples of threads with `waiting` status include a thread waiting for the completion of a communication it has started in order to get the code of the method being invoked. The `Oref` attribute contains the address of the object to which the thread is associated.

Java Classes. Each class is divided into *static* and *dynamic* parts (see Section 3.5), represented by `JavaClassS` and `JavaClassD` objects respectively. These objects contain the following attributes.

```
< C:JavaClassS| SupClass:ClassName,StaticFields:FlatFNL,Fields:FlatFNL,Methods:MethodList >
< C' : JavaClassD | ConstPool:ConstantPool, Lock:Lock, StaticFieldValues:FieldPairList >
```

The `SupClass` attribute contains the name of the immediate superclass of the class represented. The attribute `StaticFields` is a list of pairs, each pair consisting of a class name along with the list of static field names of that class. The classes in the first components of the pairs in this list are exactly the class represented by this object along with all its ancestors. These lists are compiled in a preprocessing phase. The `Fields` attribute has exactly the same structure as `StaticFields`, but for instance fields. The `ConstPool` attribute models the constant pool in the Java class file. In our model the constant pool is an indexed list containing information about methods, classes, and fields. Bytecode instructions refer to these indices, that the threads use to extract (from the constant pool) the required information to execute the instructions. By looking at the ith entry of the constant pool, we get a `FieldInfo`, which contains a field name and the name of the class the field belongs to, or a `MethodInfo`, which contains the method name, the name of the class the method belongs to, and the number of arguments of the method, or a `ClassInfo`, which only contains a class name. Examples of instructions which refer to the constant pool include,

– `new` #5, which creates a new object of the class whose name can be found in the 5th element of the constant pool, or

– `invokevirtual #3`, which invokes a method whose information (name, class, and number of arguments) can be found at the 3rd entry of the constant pool.

The `Methods` attribute contains a list of tuples, each representing a method. The structure of the tuple is as follows:

```
op {_,_,_,_,_} : MethodName MethodFormals MethodSync LabeledPgm Int -> Method .
```

The tuple components respectively represent the method name, a list of types of formal arguments of the method, a flag indicating whether or not the method is synchronized, the code of the method, and the number of local variables of the method. The `StaticFieldValues` attribute is exactly the same as `FieldValues` already discussed for `JavaObject`, except that this list refers to the values of static fields (which are stored inside the class) as opposed to the values of instance fields (which are stored inside the object). The `Lock` attribute holds the lock associated with the class.

Auxiliary Objects: Several objects in the multiset 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 prints a value, it adds this value to the end of this list. Input is assumed to be hardwired in the Java program at the moment.
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. A *thread name manager*, that is used to generate new thread names.
4. There are several *Java built-in classes* that had to be apriori defined. The support for input/output, creating new threads, and `wait/notify` facilities are among the most important ones. All of these built-in classes have been created separately and are added as part of the initial multiset.

3.2 Equational Semantics of Deterministic JVM Instructions

If a bytecode instruction can be executed locally in the thread, meaning that no interaction with the outside environment is needed, that instruction's semantics can be specified using only equations. The equations specifying the semantics of all these deterministic bytecode instructions form the E_{JVM} part of the JVM's rewrite theory. In this section we present some examples of how deterministic bytecode instructions are modeled in our system. The complete Maude representation and a collection of examples can be found in [8].

iadd instruction is executed locally in the thread, and therefore, is modeled by the equation shown in Figure 1. 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. Nothing else is changed in the thread. Many of the bytecode instructions are typed. In this example, by defining `I` and `J` to be integer variables, we support dynamic type checking as well. Several dynamic checks of this kind are supported.

```
eq <  T : JavaThread | callStack: [PC, iadd, Pgm, LocalVars, (I # J # OperandStack),
                       SyncFlag, ClassName] CallStack, Status: scheduled, ORef : OR >
=  < T : JavaThread | callStack: [PC + size(Pgm[PC]), Pgm[PC + size(Pgm[PC])], Pgm, LocalVars,
    ((I + J) # OperandStack), SyncFlag, ClassName] CallStack, Status: scheduled, ORef: OR > .
```

Fig. 1. The `iadd` instruction.

Invokevirtual is used to invoke a method from an object. It is among the most complicated bytecode instructions and its specification includes several equations and rewrite rules. The equation in Figure 2 is the first part of `invokevirtual` semantics. One thread, one Java object, and one Java class are involved. When

```
ceq < T : JavaThread | callStack: [PC, invokevirtual(I), Pgm, LocalVars, OperandStack,
                       SyncFlag, ClassName] CallStack, Status: scheduled, ORef: OR >
    < ClassName : JavaClassV | StaticFieldValues: SFV, , Lock: L
                               ConstPool:[I, {J, MethodName, CName]] ConstantPool >
    < O : JavaObject | Addr: K , FieldValues: FV, CName: ClName, Lock: L >
=   < T : JavaThread | callStack: [PC, invokevirtual(I), Pgm, LocalVars, OperandStack,
                       SyncFlag, ClassName] CallStack, Status: waiting, ORef: OR >
   < ClassName : JavaClassV | StaticFieldValues: SFV, Lock: L,
                              ConstPool: [I, {J, MethodName, ClName]] ConstantPool >
   < O : JavaObject | Addr: K, FieldValues: FV, CName: ClName, Lock: L >
   (GetMethod MethodName ofClass ClName ArgSize J forThread T)
if K==int((popLocalVars(J+1,OperandStack))[0]) .
```

Fig. 2. The `invokevirtual` instruction.

the thread reaches the `invokevirtual` instruction, by looking at the reference on top of the operand stack (`REF(K)`), it figures out from what object it has to call the method. The method information (see Section 3.1) will be extracted from the constant pool. The class `ClassName` needs to be involved, since the constant pool is stored inside this class. The class (`ClName`) is the current[3] class of the object `O`, therefore the code of the desired method should be extracted from the constant part of it. The thread will send a message asking for the code of the method, sending all the information to uniquely specify it. The last entity before the condition is a message. This message is consumed later and the desired method is sent back to the thread through another message. The thread receives the message, and that is when the invocation is complete. If the method being invoked is a *synchronized* method, the thread has to acquire a lock before the invocation is complete. This then has to be done using a rewrite rule (see Section 3.5).

3.3 Rewriting Semantics of Concurrent JVM Instructions

The semantics of those bytecode instructions that involve interaction with the outside environment is defined using rewrite rules, thus allowing us to explore all the possible concurrent executions of a program. In this section we present the semantics of several concurrent bytecode instructions.

monitorenter (Figure 3) is used to acquire a lock. This makes a change in the shared space between threads, and so has to be specified by a rewrite rule. One Java object and one Java thread are involved. The thread executing `monitorenter` acquires the lock of the object whose reference is on top of the operand stack (`REF(K)`). The heap address of the object (`K`) is matched with

[3] Note that this can change dynamically.

```
rl [MONITORENTER1] :
   <  T : JavaThread | callStack: [PC, monitorenter, Pgm, LocalVars, (REF(K) # OperandStack),
     SyncFlag, ClassName] CallStack, Status: scheduled, ORef: OR >
   < O : JavaObject | Addr: K, Lock: Lock(OIL, NoThread, 0), REST >
=> < T : JavaThread | callStack: [PC + size(Pgm[PC]), Pgm[PC + size(Pgm[PC])], Pgm, LocalVars,
     OperandStack, SyncFlag, ClassName] CallStack, Status: scheduled, ORef : OR >
   < O:JavaObject | Addr: K, Lock: Lock(OIL, T, 1), REST > .
```

Fig. 3. The `monitorenter` instruction.

this reference, and the lock of the object is changed to indicate that the object is now locked once by the thread `T` (note that a thread can lock or unlock an object several times). See section 3.4 for a more detailed discussion on locking and unlocking procedures.

getfield is a more complex instruction modeled by the rewrite rule in Figure 4. One thread and two Java classes are involved in this rule. The `I` operand is an index to the constant pool referring to the field information (`[I, {ClName, fieldname}]`), namely, field's name and its corresponding class name. The Java class `ClassName` is needed to extract the constant pool. The Java object `O` is identified by matching its heap address `K` with the reference `REF(K)` on top of the operand stack. On the right hand side of the rule, the thread proceeds to the

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

Fig. 4. `getfield` Instruction.

next instruction, and the value of the indicated field of object `O` is placed on top of the operand stack of the thread (`FV[ClName, FieldName] # OperandStack`).

3.4 Synchronization

We support three means of thread synchronization: (1) `synchronized` sections, (2) `synchronized` methods, and (3) `wait/notifyAll` methods. In this section we explain how these means of synchronization are modeled.

The `synchronized` sections in Java are translated into sections surrounded by `monitorenter` (see Figure 3) and `monitorexit` bytecode instructions. During the execution of both, an object reference is expected to be on top of the operand stack whose corresponding lock is acquired and released respectively. Each Java object is modeled by a Maude object that includes a `Lock` attribute. This attribute has a tuple structure of the following form:

```
op Lock : OidList Oid Int -> Lock .
```

The first component is a list of identifiers of all threads that have waited on this object. This corresponds to `wait` and `notifyAll` methods (see below). The second component shows what thread currently owns the lock of this object (`NoThread` if none). The third component is a counter that shows how many times the owner of the lock has acquired the lock, since each lock can be acquired several times by the same owner.

When a thread encounters the `monitorenter` instruction, it checks whether the lock of the corresponding object is free. If so, the lock is changed to belong to this thread, and the thread can proceed to the critical section. It is also possible that the lock is not free, but has been acquired by the same thread before. In this case, only the counter is increased by one. When the thread finishes the execution of the critical section and reaches the `monitorexit` instruction, it simply decreases the counter by one. If the counter becomes zero, the lock is marked as free.

The `synchronized` methods are modeled in a very similar way. The difference is that, when the method is synchronized, `monitorenter` and `monitorexit` are replaced by method invocation and return, respectively. These methods are modeled through different rewrite rules, since different bytecode instructions are used for them.

Adding support (with little effort) for the `wait` and `notifyAll` methods of the Java built-in class `Object` is an interesting problem that we have solved. Similar to synchronization primitives, `wait` and `notifyAll` are called expecting an object reference on top of the operand stack. The thread (calling these methods) should already own the lock of the object on top of the operand stack. When `wait` is called, the thread releases the lock of the corresponding object, which it must own, and goes to sleep. It will not continue unless notified by another thread. The lock of the object is marked as free, the identifier of the current thread is added to the list of threads waiting on this object (the first component of the lock), and the integer indicating the number of times the thread had locked the corresponding object is stored locally in the sleeping thread, so that it can be recalled when the thread wakes up.

When `notifyAll` is called, a (broadcast) message is created containing the list of all threads which have waited on the corresponding object up to that point. This message will then be consumed by all the threads in this list. Each thread that consumes the message will *try to* wake up. In order to continue their execution, all these threads have to compete to acquire the lock on the specific object, to follow the rest of their executions inside the synchronized section. After the lock becomes available, one thread nondeterministically [4] acquires it.

3.5 Optimizations

Below, we discuss two major optimizations we have applied to decrease the size and number of system states, as well as the size of the state space.

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

[4] In our model, but in general various implementations of the JVM use a variety of algorithms to choose the thread. By not committing to any specific deterministic choice approach, our formal analysis can discover subtle violations that may appear in some JVM implementations, but may not show up in others.

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 in the multiset, and the latter group is kept outside the multiset, in a Maude constant accessed through auxiliary operations.

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 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 the instruction `getfield` (see Section 3.3) where a rule has been used. As an example for the latter case, we refer the reader to semantics of the `monitorenter` instruction (see Section 3.3). Since only the 40 rules in R_{JVM} contribute to the size of the state space, which is basically a graph with states as nodes and rewrite transitions as edges, we obtain a much smaller state space than if all the deterministic bytecode instructions had been specified as rules, in which case 340 rules would be used.

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 the JVM, which can be used to execute Java programs in bytecode format. 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. This assumption of fairness (with respect to threads) coincides with real models of JVM with a built-in scheduler, since scheduling algorithms also take the fairness into account. This fairness assumption does not mean that a deadlock is avoided; a *deadlock* in our model is a state in which no more rewrites are possible. The fair rewriting helps us avoid the situations in which a thread stuck in a loop is being executed forever, while other threads that can also be executed are starving.

To facilitate user interaction, the JVM semantics specification is integrated within the JavaFAN tool, 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.

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. 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 [8] for the code). 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 [10, 11]. Using the search capability of our system, we also found the deadlock in the same faulty copy in 0.3 seconds. This is while the tool in [19] finds it in more than 2 seconds in its most optimized version[5].

The Thread Game. The Thread Game [18] is a simple multithreaded program which shows the possible data races between two threads accessing a common variable (see [8] for the code). Each thread reads the value of the static variable

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

`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 [18].

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. We have tried numbers up to 1000 where for all of them a solution is found in a reasonable amount time (Table 1).

N	50	100	200	400	500	1000
Time(s)	7.2	17.1	41.3	104	4.5m	10.1m

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.

Dining Philosophers. See [8] for the version of the dining philosophers problem that we have used in our experiments (DP). 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 3.1 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 (DF). In this case, we can prove the program deadlock-free when there are up to 7 philosophers. This compares favorably with JPF [1, 12] which for the same program cannot deal with 4 philosophers.

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

2-stage Pipeline implements a pipeline computation (see [8] for the code), 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 $\Box(\text{c1stop} \rightarrow \Diamond(\neg\text{stage1return}))$. 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 [19] which without using the partial order reduction performs the task in more than 100 minutes.

5 Lessons Learned and Future Work

We have presented JavaFAN, explained its design, its rewriting logic semantic basis, and its Maude implementation. We have also illustrated JavaFAN's formal analysis capabilities and its performance on several case studies. The main lessons learned are that, using a rewriting logic semantics and a high-performance logical engine as a basis to build software analysis tools for conventional concurrent programs has the following advantages: (1) it is *cost-effective* in terms of amount of work needed to develop such tools; (2) it provides a *generic* technology that can bee applied to many different languages and furthermore the analysis tools for each language come essentially *for free* from the underlying logical engine; and (3) it has *competitive performance* compared to similar software analysis tools tailored to a specific language.

As always, there is much work ahead. On the one hand, in collaboration with Feng Chen support for the Java source code level has also been added to to JavaFAN and we plan to gain more experience and further optimize the tool at both levels. On the other, we plan to extend the range of formal analyses supported by JavaFAN, including, among the others, support for *program abstraction*, to model check finite-state abstractions of infinite-state programs, and for *theorem proving*, using Maude's inductive theorem prover (ITP) [4] as a basis. Furthermore, since the general techniques used in JavaFAN are in fact language-independent, we hope that other researchers find these techniques useful and apply them to develop similar tools for other concurrent languages.

Acknowledgment. Research supported by NASA/NSF grant CCR-0234524 and by ONR grant N00014-02-0715.

References

1. G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *Automated Software Engineering 2000*, pages 3 – 12, 2000.
2. M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1):54–99, January 1987.
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. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium*, April 1998.
5. R. M. Cohen. The defensive Java Virtual Machine specification. Technical report, Electronic Data Systems Corp, 1997.

6. 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 *International Conference on Software Engineering*, pages 439 – 448, 2000.
7. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577 – 603, 1999.
8. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. JavaFAN. `fsl.cs.uiuc.edu/javafan`.
9. J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.
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, August 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, April 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. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.
16. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object Oriented Programming, Systems, and Applications*, pages 105–106, 2000.
17. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
18. J. S. Moore. `http://www.cs.utexas.edu/users/xli/prob/p4/p4.html`.
19. D. Y. W. Park, U. Stern, J. U. Sakkebaek, and D. L. Dill. Java model checking. In *Automated Software Engineering*, pages 253 – 256, 2000.
20. G. D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981.
21. J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop "Formal Underpinnings of Java" OOPSLA*, October 1998.
22. G. Roşu. Programming Language Design - CS322 Course Notes.
23. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
24. M. Stehr and C. Talcott. Plan in Maude: Specifying an active network programming language. In *Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, 2002.
25. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In *Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, 2002.
26. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 299 – 312, 2001.
27. B. Venners. *Inside The Java 2 Virtual Machine*. McGraw-Hill, 1999.
28. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.
29. M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.