Automated Software Engineering, 10, 203–232, 2003

Model Checking Programs

WILLEM VISSER wvisser@email.arc.nasa.gov
RIACS/NASA Ames Research Center, Moffet Field, CA 94035, USA

KLAUS HAVELUND
GUILLAUME BRAT
Kestrel Technologies/NASA Ames Research Center, Moffet Field, CA 94035, USA

SEUNGJOON PARK
FLAVIO LERDA
RIACS/NASA Ames Research Center, Moffet Field, CA 94035, USA

Abstract. The majority of work carried out in the formal methods community throughout the last three decades has (for good reasons) been devoted to special languages designed to make it easier to experiment with mechanized formal methods such as theorem provers, proof checkers and model checkers. In this paper we will attempt to give convincing arguments for why we believe it is time for the formal methods community to shift some of its attention towards the analysis of programs written in modern programming languages. In keeping with this philosophy we have developed a verification and testing environment for Java, called Java PathFinder (JPF), which integrates model checking, program analysis and testing. Part of this work has consisted of building a new Java Virtual Machine that interprets Java bytecode. JPF uses state compression to handle big states, and partial order and symmetry reduction, slicing, abstraction, and runtime analysis techniques to reduce the state space. JPF has been applied to a real-time avionics operating system developed at Honeywell, illustrating an intricate error, and to a model of a spacecraft controller, illustrating the combination of abstraction, runtime analysis, and slicing with model checking.

Keywords: model checking, Java, symmetry, abstraction, runtime analysis, static analysis

1. Introduction

The majority of work carried out in the formal methods community throughout the last three decades, since Hoare's axiomatic method for proving programs correct (Hoare, 1969), has been devoted to special languages that differ from main stream programming languages. Typical examples are formal specification languages (The RAISE Language Group, 1992; Bjørner and Jones, 1982; Spivey, 1992), purely logic based languages used in theorem provers (Gordon, 1988; Owre et al., 1996; Cornes et al., 1995), and guarded command languages used in model checkers (Melton et al., 1996; McMillan, 1993; Larsen et al., 1998). In a few cases, modeling languages have been designed to resemble programming languages (Holzmann, 1997b), although the focus has been on protocol designs. Some of these linguistic choices have made, and still make, it feasible to experiment more conveniently with new algorithms and frameworks for analyzing system models. For example, a logic based language is well suited for rewriting, and a rule based guarded command notation is convenient for a model checker. We believe that continued research in special languages is important since this research investigates semantically clean language concepts and will impact future language designs and analysis algorithms.

We, however, want to argue that a next important step for the formal methods subgroup of the software engineering community could be to focus some of its attention on real programs written in modern programming languages. We believe that studying programming languages somehow will result in some new challenges that will drive the research in new directions as described in the first part of the paper. Our main interest is in multi-threaded, interactive programs, where unpredictable interleavings can cause errors, but the argument extends to sequential programs.

In the second part of the paper, we describe our own effort to follow this vision by presenting the development of a verification, analysis and testing environment for Java, called Java PathFinder (JPF). This environment combines model checking techniques with techniques for dealing with large or infinite state spaces. These techniques include static analysis for supporting partial order reduction of the set of transitions to be explored by the model checker, predicate abstraction for abstracting the state space, and runtime analysis such as race condition detection and lock order analysis to pinpoint potentially problematic code fragments. Part of this work has consisted of building a new Java Virtual Machine (JVM^{JPF}) that interprets Java bytecode. JVM^{JPF} is called from the model checking engine to interpret bytecode generated by a Java compiler.

We believe it is an attractive idea to develop a verification environment for Java for three reasons. First, Java is a modern language featuring important concepts such as object-orientation and multi-threading within one language. Languages such as C and C++, for example, do not support multi-threading as part of their core. Second, Java is simple, for example compared to C++. Third, Java is compiled into bytecode, and hence, the analysis can be done at the bytecode level. This implies that such a tool can be applied to any language that can be translated into bytecode.[1] Bytecode furthermore seems to be a convenient breakdown of Java into easily manageable bytecode instructions; and this seems to have eased the construction of our analysis tool. JPF is the second generation of a Java model checker developed at NASA Ames. The first generation of JPF (JPF1) (Havelund, 1999a; Havelund and Pressburger, 1999) was a translator from Java to the Promela language of the Spin model checker.

The paper is organized as follows. Section 2 outlines our arguments for applying formal methods to programs. Section 3 describes JPF and Section 4 its integration within the BANDERA toolset. Section 5 describes related work on model checking C programs. Section 6 presents two applications of JPF: a real-time avionics operating system developed at Honeywell, illustrating an intricate error; and a model of a space craft controller, illustrating the combination of abstraction, runtime analysis, and slicing with model checking to locate a deadlock. Both of these errors were problems in the real code of these systems. Finally, Section 7 contains conclusions and a description of future work.

2. Why analyze code?

It is often argued that verification technologies should be applied to designs rather than to programs since catching errors early at the design level will reduce maintenance costs later on. We do agree that catching errors early is crucial. State of the art formal methods also most naturally lend themselves to designs, simply due to the fact that designs have less complexity, which make formal analysis more feasible and practical. Hence, design verification is a very important research topic, with the most recent popular subject being analysis of statecharts (Harel, 1987), such as for example found in UML (Booch et al., 1999). However, we want to argue that the formal methods community should focus some of its attention on programs for a number of reasons that we will describe below.

First of all, programs often contain fatal errors despite the existence of careful designs. Many deadlocks and critical section violations, for example, are introduced at a level of detail which designs typically do not deal with, if formal designs are made at all. This was for example demonstrated in the analysis of NASA's Remote Agent spacecraft control system written in the LISP programming language, and analyzed using the Spin model checker (Havelund et al., 1998). Here several classical multi-threading errors were found that were not really design errors, but rather programming mistakes such as forgetting to enclose code in critical sections. One of the missing critical section errors found using Spin was later introduced in a sibling module, and caused a real deadlock during flight in space, 60,000 miles from earth (Havelund et al., 2000); see Section 6.1. Another way of describing the relationship between design and code is to distinguish between two kinds of errors. On the one hand there are errors caused by flaws in underlying complex algorithms. Examples of complex algorithms for parallel systems are communication protocols (Havelund and Shankar, 1996; Helmink et al., 1994) and garbage collection algorithms (Havelund, 1999b; Russinoff, 1994). The other kind of error is more simple minded concurrency programming errors, such as forgetting to put code in a critical section or causing deadlocks. Errors of this kind will typically not be caught in a design, and they are a real hazard, in particular in safety critical systems. Complex algorithms should probably be analyzed at the design level, although there is no reason such designs cannot be expressed in a modern programming language. However, as will be shown on a real example in Section 6.2, deep design errors can also appear in the code.

Second, one can argue that since modern programming languages are the result of decades of research, they are the result of good language design principles. Hence, they may be good design/modeling languages. This is to some extent already an applied idea within UML where statechart transitions (between control states) can be annotated with code fragments in your favorite programming language. In fact, the distinction between design and program gets blurred since final code may get generated from the UML designs. An additional observation is that some program development methods suggest a prototyping approach where the system is incrementally constructed using a real programming language, rather than being derived from a pre-constructed design. This was for example the case with the Remote Agent (Muscettola et al., 1998) mentioned above. Furthermore, any research result on programming languages can benefit design verification since designs typically are less complex.

A third, and very different kind of, argument for studying verification of real programs is that such research will force the community to deal with very hard problems, and this will hopefully drive research into new areas. We believe, for example, that it could be advantageous for formal methods to be combined with other research fields that traditionally have been more focused on programs, such as static program analysis and testing. Such techniques are typically less complete, but they often scale better. We believe that the objective of formal methods is not only to prove programs correct, but also to debug programs and locate errors. With such a more limited ambition, one may be able to apply techniques which are less complete and based on heuristics, such as certain testing techniques.

Fourth, studying formal methods for programming languages may, furthermore, have some derived advantages for the formal methods community since there is a tendency to standardize programming languages. This may make it feasible to compare and integrate different tools working on the same language—or on “clean subsets” of these languages. As mentioned above, it would be very useful to study the relationship between formal methods and other areas such as program analysis and testing techniques. Working at the level of programs will make it possible to better interact with these communities. We have already had one such experience in our informal collaboration with Kansas State University, where our tool generated a slicing criterion based on runtime analysis, and their tool could slice the Java program based on this criterion, where after we could apply our model checker to the resulting program. A final derived advantage will be the many orders of magnitude increased access to real examples and users who may want to experiment with the techniques produced. This may have a very important impact on driving the research towards scalable solutions.

In general, it is our hope that formal methods will play a role for everyday software developers. By focusing on real programming languages we hope that our community will be able to interact more intensively on solving common problems. Furthermore, the technology transfer problem so often mentioned may vanish, and instead be replaced by a technology demand.

3. Model checking Java programs

It is well known that concurrent programs are non-trivial to construct, and with Java essentially giving the capability for anyone to write concurrent programs, we believe, a model checker for Java might have a bright future. In fact, one area where we believe it can have an immediate impact is in environments where Java is taught. In the rest of this section we will address some of the most important issues in the model checking of programming languages. Specifically, we will highlight the major reasons whymodel checking programs is considered hard, and then illustrate how we tackle these problems within JPF.

3.1. Complexity of language constructs

Input languages for model checkers are often kept relatively simple to allow efficient processing during model checking. Of course there are exceptions to this, for example Promela, the input notation of Spin (Holzmann, 1997b), more resembles a programming language than a modeling language. General programming languages, however, contain many new features almost never seen in model checking input languages, for example, classes, dynamic memory allocation, exceptions, floating point numbers, method calls, etc. How will these be treated? Three solutions are currently being pursued by different groups trying to model check Java: one can translate the new features to existing ones, one can create a model checker that can handle these new features, or, one can use a combination of translation and a new/extended model checker.

3.1.1. Translation. The first version of JPF (Havelund and Pressburger, 1999), as well as the JCAT system (Demartini et al., 1999a), was based on a translation from Java to Promela. Although both these systems were successful in model checking some interesting Java programs (Havelund and Skakkebaek, 1999; Demartini et al., 1999a), such source-to-source translations suffer from two serious drawbacks:

Language coverage—Each language feature of the source language must have a corresponding feature in the destination language. This is not true of Java and Promela, since Promela for example, does not support floating point numbers.

Source required—In order to translate one source to another, the original source is required, which is often not the case for Java, since only the bytecodes are available—for example in the case of the libraries and code loaded over the WWW.

For Java, the requirement that the source exists can be overcome by translating directly from bytecodes. This is the approach used by the BANDERA tool (Corbett et al., 2000a), where bytecodes, after some manipulation, are translated to either Promela or the SMV model checker's input notation. The Stanford Java model checker also uses this approach, by translating bytecodes to the SAL intermediate language for model checking (Park et al., 2000). Their SAL model checker is however specifically developed for the purpose of checking programs with dynamic data-structures and hence could be argued to fall into the custom-made model checker category below.

3.1.2. Custom-made model checker. In order to overcome the language coverage problem it is obvious that either the current model checkers need to be extended, or a new custom-made model checker must be developed. Some work is being done on extending the Spin model checker to handle dynamic memory allocation (Demartini et al., 1999b; Visser et al., 1999), but again in terms of Java this only covers a part of the language and much more is required before full Java language coverage will be achieved this way.With JPF we took the other route, we developed our own custom-made model checker that can execute all the bytecode instructions, and hence allow the whole of Java to be model checked. The model checker consists of our own Java Virtual Machine (JVM^{JPF}) that executes the bytecodes and a search component that guides the execution. Note that the model checker is therefore an explicit state model checker, similar to Spin, rather than a symbolic one based on Binary Decision Diagrams such as SMV (McMillan, 1993). Also, we decided that a depth-first traversal with backtracking would be most appropriate for checking temporal liveness properties (breadth-first liveness checking is inefficient due to the problems in detecting cycles). A nice side-effect of developing our own model checker was the ease with which we are able to extend the model checker with interesting new search algorithms—this would, in general, not have been easy to achieve with existing model checkers (especially not with Spin). A major design decision for JPF was to make it as modular and understandable to others as possible, but we sacrificed speed in the process—Spin is at least an order of magnitude faster than JPF.We believe this is a price worth paying in the long run.

JPF is written in Java and uses the JavaClass package[2] to manipulate classfiles. Although we again sacrifice speed to some extent by not using C/C++, there is no doubt in our minds that doing JPF in Java has saved us months on development time. The initial system, which could only handle integer based bytecodes (i.e. the same language subset as the Java model checkers translating to Spin), was developed in 3 man-months. The system as described in this paper, required approximately 15 man-months.

3.1.2.1. Language and properties supported. The JVM^{JPF} supports *all* Java bytecodes, hence any program written in *pure* Java can be analyzed. Unfortunately, not all Java programs consist of *pure* Java code—one often finds that certain methods are defined as being *native* to the operating system. When a Java program calls methods that have no corresponding bytecodes, then JPF cannot determine what the state of these code fragments will be and hence cannot handle programs that, for example, access the file system (user-defined class-loaders, file I/O operations, etc.), or communicate over a network, contains GUI code, etc. Fortunately, many native methods do not have side-effects and hence simple wrapper-methods can be written that translate the inputs and outputs to the native method, which then allow the original method to be called and all state changes to happen after returning from the call. However, if the native method itself causes an error, JPF will not be able to detect it, unless its output also causes an error in the Java code. Furthermore, JPF can only handle *closed* systems, i.e. a system and the environment it will execute in. This however is also the case in testing, where a test-harness is required to *close* a system, and is not considered a drawback of the approach.

The current model checker can check for deadlocks, invariants and userdefined assertions in the code, as well as Linear Time Temporal Logic (LTL) properties. In fact JPF supports all properties expressible in the BANDERA tool, the interested reader is referred to Corbett et al. (2000b) for more detail.

3.2. Complex states

In order to ensure termination during explicit state model checking one must know when a state is revisited. It is common for a hashtable to be used to store states, which means an efficient hash function is required as well as fast state comparison.

The Verisoft system (Godefroid, 1997) was developed to model check software, but the design premise was that the *state* of a software system is too complex to be encoded efficiently, hence Verisoft does not store any of the states it visits (Verisoft limits the depth of the search to get around the termination problem mentioned above). Since the Verisoft system executes the actual code (C/C++), and has little control over the execution, except for some user-defined "hooks" into communication statements, it is almost impossible to encode the system state efficiently. This insight also convinced us that we cannot tie our model checking algorithm in with an existing JVM, which is in general highly optimized for speed, but will not allow the memory to be encoded easily. In Stoller (2000), a state-less model checking algorithm similar to that of Verisoft is described for Java. This system instruments the bytecodes for a program with "hooks" to allow model checking.

Our design philosophy was to keep the states of the JVM in a complex data-structure, but one that would allow us to encode the states in an efficient fashion in order to determine if we have visited states before. Specifically, each state consists of three components: information for each thread in the Java program, the static variables (in classes) and the dynamic variables (in objects) in the system. The information for each thread consists of a stack of frames, one for each method called, whereas the static and dynamic information consists of information about the locks for the classes/objects and the fields in the classes/objects. Each of the components mentioned above is a Java data-structure. In early stages of JPF development we did store these structures directly in a hashtable, but with terrible results in terms of memory and speed: 512 MB would be exhausted after only storing ± 50000 states, and ± 20 states could be evaluated each second (on a SPARC ULTRA60).

The solution we adopted to make the storing of states more efficient, was a generalization of the *Collapse* method from Spin (Holzmann, 1997a): each component of the JVM state is stored separately in a table, and the index at which the component is stored is then used to represent the component. More specifically, each component (for example the fields in a class/object) is inserted in a table for that component; if the specific component is already in the table its index is returned, and if it is unique it is stored at the next open slot and that index is returned. This has the effect of encoding a large structure into no more than an integer[3] (see figure 1). Collapsing states in this fashion allows fast state comparisons, since only the indexes need to be compared and not the structures themselves. The philosophy behind the collapsing scheme is that although many states can be visited by a program the underlying components of many of these states will be the same. A somewhat trivial example of this is when a statement updates a local variable within a method: the only part of the system that changes is the frame representing the method, all the other parts of the

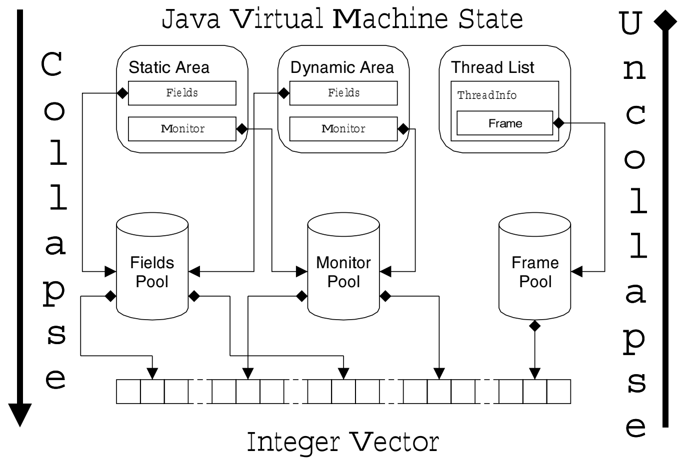

Figure 1. Collapsing and recreating the JVM state.

system state are unaffected and will *collapse* to the same indexes. This actually alludes to the other optimization we added: only update the part of the system that changes, i.e., keep the indexes calculated for the previous state the same, only calculate the one that changed (to date we have only done this optimization in some parts of the system). After making these changes the system could store millions of states in 512 MB and could evaluate between 500 and 1500 states per second depending on the size of the state (on a SPARC ULTRA60).

It was however clear from profiling the system execution that there was still one major source of inefficiency—the collapsing of states was only used for the states stored in the hashtable, but in order to allow backtracking the un-collapsed states are stored in a stack. More specifically, whenever a new state is generated a copy of this state is made and put on the stack, during backtracking this state is removed again and execution continues. The Java "clone" operation is used to make copies of states, but this operation is notoriously slow since our states are represented by such a complex data-structure. Memory consumption was also high due to the complexity of each state, and we could seldom analyze a system withmore than 10000 states in a depth-first path. A very simple, and above all novel solution, however presented itself: use the reverse of the collapse operation to recreate a state from its collapsed description (see figure 1). We could now use the collapsed state description in both the hashtable and the stack, and during backtracking the collapsed state is uncompressed by reversing the lookup in the tables (i.e. use the index to retrieve the original object from the table). This saves time since recreating the state from its collapsed form is faster than copying the state, and also saves memory since we now only create one collapsed copy of the state, which is stored in the hashtable, and we keep a reference to this state in a stack entry. Lastly, as before, since only part of the state changes during each transition we can also just uncollapse the parts that changed during backtracking. These last changes improved memory usage 4 fold and the model checker can now evaluate between 6000 and 10000 states per second depending on the size of the state (on a SPARC ULTRA60). For a more detailed description of the state compression used within JPF the interested reader is referred to Lerda and Visser (2001).

JPF in its current state already illustrates that software systems with complex states can be efficiently analyzed (see Section 6), but with some further extensions and better hardware platforms to run it on, we believe, systems of up to 10 k lines of code could be analyzed.

3.3. Curbing the state space explosion

Maybe the most challenging part of model checking is reducing the size of the state space to be explored to something that your tool can handle. Since designs often contain less detail than implementations, model checking is often thought of as a technique that is best applied to designs, rather than implementations. We believe that applying model checking by itself to programs will not scale to programs of much more than 10000 lines. The avenue we are pursuing is to augment model checking with information gathered from other techniques in order to handle large programs. Specifically, we are investigating the use of symmetry reductions, abstract interpretation, static analysis and runtime analysis to allow more efficient model checking of Java programs. Figure 2 illustrates the architecture

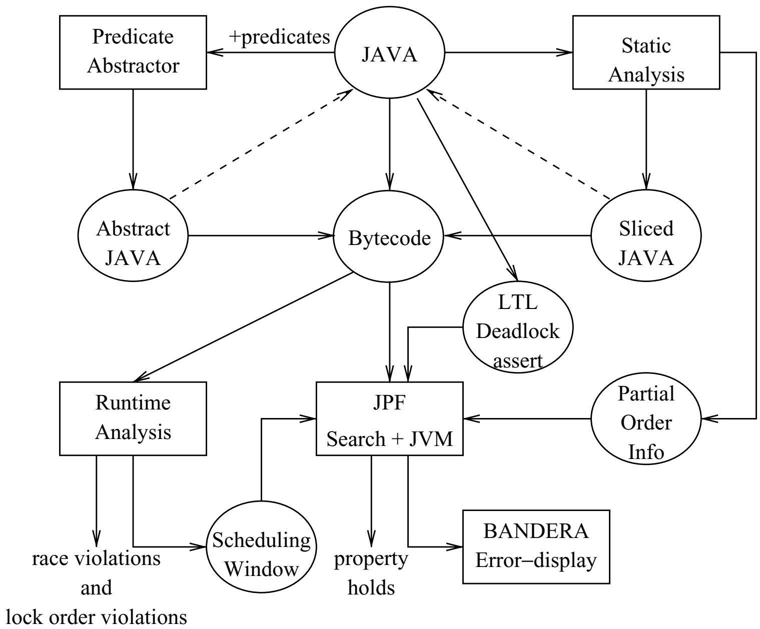

Figure 2. The JPF tool architecture. Dotted lines indicate iterative analysis.

of JPF and its companion tools (abstraction tool, static analyzer and runtime analyzer) that will be described in detail below.

3.3.1. Symmetry reductions. The main idea behind symmetry reductions (Clarke et al., 1993, 1998; Emerson and Sistla, 1993; Ip and Dill, 1993) is that symmetries induce an equivalence relation on states of the system, and while performing analysis of the state space (for example during model checking) one can discard a state if an equivalent state has already been explored. Typically a canonicalization function is used to map each state into a unique representative of the equivalence class. Various schemes have been proposed for efficiently implementing such functions (Ip and Dill, 1993) and the complexity of this problem is discussed in Clarke et al. (1998). Software programs can in general induce a great many symmetries, but here we will focus on a number of symmetry related problems found when analyzing Java programs: class loading and two forms of symmetry in the heap (dynamic area). The problem we are trying to avoid is the analysis of states that are equivalent to previously analyzed states.

Java programs have dynamic behavior and one cannot predict which classes will be loaded, objects will be instantiated, or even in which order these will occur. This lack of order would seem to suggest an appropriate representation for the static area (where static variables for each class are stored) and the dynamic area (where objects are allocated) should be as sets. Comparing sets is however too time consuming, but an obvious ordering can be used, namely, the order in which classes are loaded or objects created. This however means that states will be considered to be different, if their only difference is the order of class loading (similarly if the same objects are placed in different locations in the dynamic area). Of course, since we analyze Java programs depth-first, different interleavings of transitions will cause the above-mentioned problem. What is required is to ensure that the static area and dynamic area have a canonical representation regardless of which interleaving of transitions is being executed.

A canonicalization function for the static area is simple to define, since we can order the locations where the static variables of a class will be in the static area by ordering the class names. For each class loader in Java the class names must be unique, and since we do not consider the case of more than one class loader being used a simple mapping of class names to positions in the static area is enough. For example, if class A is loaded before class B in one interleaving then the static variables for class A will be stored at position 0 in the static area, and this mapping $A \rightarrow 0$ will be remembered, when class B is loaded the mapping $B \rightarrow 1$ will be remembered. After backtracking let us assume class B is now loaded before A, then the mapping for B will be recalled and B's static variables will be loaded at position 1 even though position 0 is available (class A's static variables will be loaded there).

Unfortunately, a similar approach with object allocation in the dynamic area is not sufficient since there can be many objects instantiated from the same class. One can however identify each object allocation in a Java program by uniquely identifying each "NEW" bytecode.[4] This is not yet sufficient to define a mapping, since the same "NEW" can be executed more than once, for example when an allocation is in a loop. An occurrence number, that is incremented each time the new is executed and decremented whenever the instruction is backtracked over, can then be used to identify each allocation. Although the combination of the new-identifier and an occurrence number will distinguish many cases where there is symmetry, it does not resolve all cases. For example if the same allocation code can be executed from two different threads the symmetry reduction will be missed and equivalent states will be considered different. A thread reference can be added to distinguish this case. Clearly there is a trade-off between the precision of the canonicalization function and the time taken to calculate it—we chose to rely only on the new-identifier and the occurrence number in our current system.

Readers familiar with partial order reduction rules (Holzmann and Peled, 1994; Godefroid, 1996) might notice that the symmetry reductions described above are closely related. As will be seen from the following example partial order reductions, where unnecessary interleavings of independent transitions in different threads are not executed, subsume some, but not all, of the reductions achieved by the canonical view of the heap (similarly for class loading).

```
class S1 { int x; }         class S2 { int y; }
class FirstTask             class SecondTask
   extends Thread {          extends Thread {
  public void run() {         public void run() {
    S1 s1; int x = 1;           S2 s2; int x = 1;
    s1 = new S1();              s2 = new S2();
    x = 3;                      x = 3;
} }                         } }
```

```
class Main{
  public static void main(String[] args){
   FirstTask task1 = new FirstTask();
   SecondTask task2 = new SecondTask();
   task1.start(); task2.start();
} }
```

The program above has two independent threads that both allocate object entries in the heap, and since “s1” and “s2” can be swapped around in the heap depending on the interleaving chosen, symmetry reductions are applicable. When performing a deadlock analysis on this example without any symmetry reductions or partial-order reductions, JPF reports evaluating 258 states, with just symmetry reduction it reports 105 states, with just partialorder reduction (see Section 3.3.3 for more details) it generates 68 states and with both symmetry and partial-order reductions it generates only 38 states.

Lastly, there is one more form of symmetry reduction in the dynamic area that is required for Java programs, namely garbage collection. Garbage refers to objects that have been allocated, but can now no longer be reached from any data-structure in the program. The problem with garbage is that, unless it is removed (collected), the size of the state will grow indefinitely, and hence all states will be considered different. For example, without garbage collection the following program is essentially infinite-state, since each time round the loop, the string to be printed is allocated a new buffer to allow the printing method to print it.

```
class Main {
  public static void main (String args[]) {
    while(true) {
      System.out.println("0");
} } }
```

We use a form of mark-and-sweep to do garbage collection. Although not often thought of as such, garbage collection is clearly a canonicalization function that would allow symmetry reduction—states with and without garbage can be equivalent. Analyzing the above program with JPF with garbage collection results in the “0” being printed only twice before all states are generated. A good overview of garbage collection for model checking can be found in Iosif and Sisto (2000).

3.3.2. Abstraction. Recently, the use of abstraction algorithms based on the theory of abstract interpretation (Cousot and Cousot, 1992), has received much attention in the model checking community (Graf and Saidi, 1997; Das et al., 1999; Saidi, 1999; Saïdi and Shankar, 1999; Colón and Uribe, 1998). The basic idea underlying all of these is that the user specifies an *abstraction function* for certain parts of the data-domain of a system. The model checking system then, by using decision procedures, either automatically generates, on-the-fly during model checking, a state-graph over the abstract data (Graf and Saidi, 1997; Saidi, 1999; Das et al., 1999) or automatically generates an abstract system, that manipulates the abstract data, which can then be model checked (Saïdi and Shankar, 1999; Colón and Uribe, 1998). The trade-off between the two techniques is that the generation of the state-graph can be more precise, but at the price of calling the decision procedures throughout the model checking process, whereas the generation of the abstract system requires the decision procedures to be called proportionally to the size of the program. It has been our experience that abstractions are often defined over small parts of the program, within one class or over a small group of classes, hence we favor the generation of abstract programs, rather than the on-the-fly generation of abstract state-graphs. Also, it is unclear whether the abstract state-graph approach will scale to systems with more than a few thousand states, due to the time overhead incurred by calling the decision procedures.

Specifically we have developed an abstraction tool for Java that takes as input a Java program annotated with user-defined predicates and, by using the Stanford Validity Checker (SVC) (Barrett et al., 1996), generates another Java program that operates on the abstract predicates. For example, if a program contains the statement `x++` and we are interested in abstracting over the predicate `x==0`, written as `Abstract.addBoolean("B",x==0)`, then the increment statement will be abstracted to the code: "`if (B) then B = false else B = Verify.randomBool()`" where nondeterministic choice is indicated by the `randomBool()` method that gets trapped by the model checker. The BANDERA tool uses similar techniques to abstract the data-domains of, for example, an integer variable to work over the abstract domains *positive*, *negative* and *zero* (the so-called sign abstraction), by using the PVS model checker. The novelty of our approach lies in the fact that we can abstract predicates over more than one class: for example, we can specify a predicate `Abstract.addBoolean("xGTy", A.x > B.y)` if class A has a field x and class B has a field y. The abstracted code allows for many instantiations of objects of class A and B to be handled correctly—the interested reader is referred to Visser et al. (2000) for more details on the techniques used. Although our Java abstraction tool is still under development we have had very encouraging results. For example we can, in a matter of seconds, abstract the omnipresent infinite-state Bakery algorithm written in Java to one that is finite-state and can be checked exhaustively. In Section 6.1 we also show how the abstraction tool is used on a real example.

Abstractions for model checking often over-approximate the behavior of the system, in other words, the abstracted system has as a subset the behaviors of the original system. Since the properties that are typically checked are universally quantified over all paths, over-approximations preserve correctness—if a property holds in the abstracted system it is also true of the original system. Unfortunately, when it comes to model checking programs, or any other type of system for that matter, it is often the case that we are interested in finding errors, not showing correctness. And here lies a problem: over-approximations do not preserve errors, i.e. errors in the abstract system might be due to new behaviors that were added and are not present in the original system. Eliminating these spurious errors is an active research area (Saidi, 1999, 2000; Ball and Rajamani, 2000b; Clarke et al., 2000). We adopted a pragmatic approach to this problem that seems to work very well in practice (Dwyer et al., 2001; Pasareanu et al., 2001). This work was inspired and implemented in JPF by Corina Pasareanu from the BANDERA group at Kansas State University and a full account of the approach can be found in Pasareanu et al. (2001).

The basic idea is as follows: from a theorem in Saidi (2000) it follows that any path in the abstracted program that is free of nondeterministic choices is also a path of the original program, hence if an error occurs on such a "choose-free" path then it is not spurious. JPF has a special mode in which it searches for errors only on paths that are choose-free—since nondeterminism in JPF is trapped by recognizing special method calls, it is easy to truncate a search whenever such a call occurs. Of course, if no error is found in this special mode, then the result is inconclusive since an error might exist, but the abstraction is not adequate to find the error in the choose-free mode. The next step is now to look for errors that may contain nondeterministic choices, if such an error exists, we can run this path in a simulation mode on the original program (there is a 1-to-1 mapping of code from the abstract to the original code) and if it diverges, i.e. the abstract path says statement $s1$ should be executed but the concrete program says $s2$ should be executed, then we can use the last decision point taken before divergence to refine the abstraction. If the path does not diverge we can also be sure that the error is not spurious. Note that we do not need to symbolically execute the abstract path on the concrete program, since the Java programs we check are by definition closed systems, i.e. they take no unknown input, and also each program has a single initial state.

3.3.3. Static analysis. Static analysis of programs consists of analyzing programs without executing them. In general, the analysis is performed without making assumptions about the inputs of the program. The analysis results are therefore valid for any set of inputs. A wide variety of techniques fall under the static analysis umbrella; e.g., data flow analysis, set and constraint resolution, abstract interpretation, and theorem proving can all be applied to static analysis problems (with various degrees of success). They all derive some properties about a program. These properties are then used in slicing, code optimization, code parallelization, abstract debugging, code verification, code understanding, or code re-engineering for examples.

Our interest in static analysis lies in its potential for reducing the size of the state space generated by a program. Therefore, we have focused our efforts on three static analysis problems that can result in state space reduction: static slicing, partial evaluation, and partial order computation. Static slicing takes a program and a slicing criterion and generates a smaller program that is functionally equivalent to the original program with regard to the criterion. Partial evaluation (at least our version) propagates constant values and simplifies expressions in the process. Partial order computation focuses on identifying statements that can be safely interleaved with any statement on a different thread. The combined use of these analyses results in smaller state spaces, and therefore, helps reduce the state explosion problem. However, they do it in different manners. On the one hand, static slicing and partial evaluation generate a (functionally equivalent) smaller program that results in a smaller state space as shown in figure 3. Black states indicate states that directly affect the slicing criterion (e.g., because they modify a variable involved in a property we want to check). After slicing, only the states affecting the slicing criterion remain in the state space. On the other hand, partial order computation does not change the size of the program, but its results can be used to further reduce the state space by eliminating unnecessary interleavings. In the rest of this section we discuss static slicing and its application in model checking. We then briefly describe our partial order computation approach.

One approach to reducing the size of programs, and therefore the size of the state space to be model checked, is to eliminate statements that are not relevant to the property one

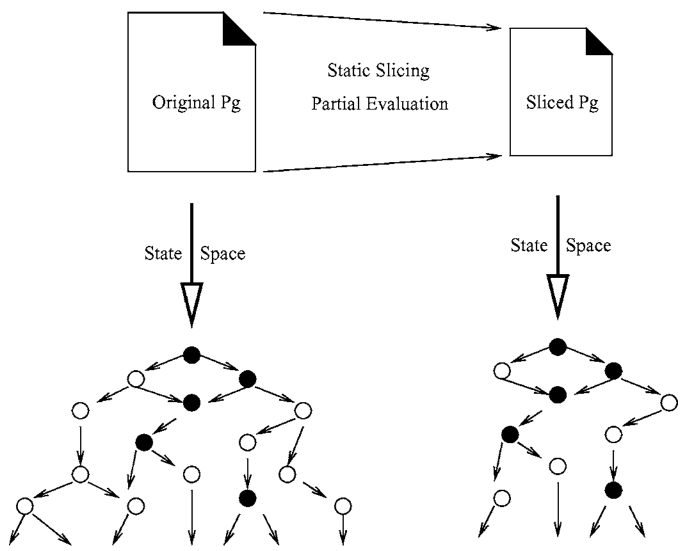

Figure 3. Reduction of programs using static slicing.

wants to verify. In static analysis, this process is known as program slicing (Weiser, 1984). It has been studied quite extensively and the interested reader can find a detailed survey on slicing in (Tip, 1995). In general, a program slice is defined by the parts of a program that may affect (or be affected by) a slicing criterion. Typically a slicing criterion consists of a set of program points of interest. The sliced program is smaller than the original program and is functionally equivalent with respect to the slicing criterion. In this paper, we focus on works that use slicing as a program reduction tool for model checking as shown in Clarke et al. (1999), Hatcliff et al. (1999), and Millett and Teitelbaum (1998).

When slicing for model checking, criteria are often related to the properties that one wants to check, e.g., for a given property $\mathcal{P}$, the slicing criterion is the set of program points affecting the values of the variables present in $\mathcal{P}$. Therefore, every statement affecting the slicing criterion should be present in the slice (or sliced program); otherwise, the resulting program is not functionally equivalent to the original program. If such a statement was missing from the slice, it could result in a situation where the model checker states that a property holds on the sliced program even though it does not hold on the original program. This type of slicing is called closure slicing: a closure slice of a program P with respect to program point p and variable x consists of all statements that may affect the value of x at p.

Closure slicing is not quite sufficient in our case. In order to generate a state space, JPF executes the program. Therefore, we need the sliced program to be executable: an executable slice of P with respect to p and x is a reduced program whose behavior with respect to x cannot be distinguished from the behavior of P with respect to x at point p. Closure slices are usually obtained by computing the closure of a dependence graph obtained by some type of interprocedural data and control dependence analysis. Fortunately, it has been shown that a closure slice can be extended to an executable slice (Binkley, 1993). Therefore, the main problem is reduced to the computation of closure slices.

A similar approach has been applied to slicing and model checking VHDL programs (Clarke et al., 1999). Since VHDL programs consist of concurrent processes the authors had to adapt traditional slicing techniques to handle concurrency. Roughly speaking, their approach consists of mapping VHDL constructs to traditional sequential program constructs in such a way that valid VHDL traces are also valid traces of the sequential program. Once this transformation is performed, they apply traditional, yet quite precise, interprocedural slicing techniques defined for sequential languages such as C or Ada. Other works have taken a more direct route without any transformation. Thus, in Millett and Teitelbaum (1998), a slicing technique is described for Promela programs which can be used with the Spin model checker. Their technique is directly inspired by the work of Cheng on slicing concurrent programs (Cheng, 1997) with extensions to handle dynamic process creation. In essence, their approach consists of performing dependence analysis on system dependence graphs (SDG) which represents not only sequential dependencies but also concurrent dependencies. Therefore, an SDG is similar to the program dependence graph for a sequential program except that it has additional edges to represent dependencies due to concurrency. For example, it has "non-deterministic" edges between the guard of a guarded command and its guarded statements and data dependence edges between statements using shared variables. This extended SDG is a conservative approximation of data and control dependencies in the presence of interleaving. When all possible interleavings are considered the size of the SDG may be quite large. However it can be pruned when atomic statements are used in the Promela programs. This is an example were partial order computation is used before static slicing. Still, the analysis is quite imprecise because of many approximations. Yet the authors claim that it yields significant reductions in practice.

JPF uses the slicing tool of the BANDERA toolset which implements the work of Hatcliff et al. (1999) on static slicing of concurrent Java programs. Their technique consists of computing a set of program dependencies affecting the slicing criteria. These dependencies include the traditional dependencies (data, control and divergence) for sequential programs as well as their counterparts (interference, synchronization and ready dependencies) for concurrent programs. Informally, interference dependencies represent cases where the definition of shared variables can reach across threads. Synchronization dependence focuses on the use of synchronize statements; it basically states that if a variable is defined at a node inside some critical region, then the locking associated with that region must be preserved (i.e., the inner-most enclosing synchronize statement must be present in the slice). Ready dependence states that a statement n is dependent on a statement m if m's failure to complete (e.g., because a wait or notify never occurs) can block the thread containing n. In BANDERA, slicing is not performed on the Java source code, but on its (3-address code) representation called Jimple (Jimple is an intermediate representation for Java used in the Soot compiler developed atMcGill University (Valle-Rai et al., 1999)). In BANDERA, Jimple code is then translated into Promela or SMV code and then model checked. In order to use slicing and abstraction iteratively, and, since abstraction works on the source code level, we have to convert the sliced Jimple program back to Java source code using annotations that describe the original Java program. This approach has benefited JPF in several ways. First, using BANDERA, we can extract slicing criteria (i.e., program points) automatically from the properties verified by JPF. Second, BANDERA also provides support for partial symbolic evaluation, which yields smaller state spaces. Third, we can re-use the dependence analysis performed by BANDERA to compute partial order information.

Within JPF, static analysis is also used to determine which Java statements in a thread are independent of statements in other threads that can execute concurrently. This information is then used to guide the partial-order reductions (Holzmann and Peled, 1994) built into JPF. Partial-order reduction techniques ensure that only one interleaving of independent statements is executed within the model checker. It is well established from experience with the Spin model checker that partial-order reductions achieve an enormous state-space reduction in almost all cases. We have had similar experience with JPF, where switching on partial-order reductions caused model checking runs that ran for hours to finish within minutes. We believe model checking of (Java) programs will not be tractable in general if partial-order reductions are not supported by the model checker and in order to calculate the independence relations required to implement the reductions, static analysis is required.

Even though static analysis has already given us great benefits in terms of state space reduction, we plan on investigating how we can improve the precision of its results (and therefore, achieve greater reductions). We are especially interested in researching how model checking and static analysis can feed off each other's results to achieve greater precision. Some, like Cousot (Cousot and Cousot, 1997), have already stated their beliefs that both techniques can be used in parallel; intermediate results can be used by processes to increase their precision. Cousot's study focused on a particular static analysis technique called abstract interpretation and symbolic model checking; it may be possible to extend it to explicit-state model checking. Yet, we are not convinced that a parallel approach is practical given the difference of speed between the two techniques. Our initial premise is that an iterative approach (where static analysis and model checking are used successively) may be more practical.

3.3.4. Runtime analysis. Runtime analysis is conceptually based on the idea of executing a program once, and observing the generated execution trace to extract various kinds of information. This information can then be used to predict whether other different execution traces may violate some properties of interest (in addition of course to demonstrating whether the generated trace violates such properties). The important observation here is that the generated execution trace itself does not have to violate these properties in order for their potential violation in other traces to be detected. Runtime analysis algorithms typically will not guarantee that errors are found since they after all work on a single arbitrary trace. They also may yield false positives in the sense that analysis results indicate warnings rather than hard error messages. What is attractive about such algorithms is, however, that they scale very well, and that they often catch the problems they are designed to catch. That is, the randomness in the choice of run does not seem to imply a similar randomness in the analysis results. In practice runtime analysis algorithms will not store the entire execution trace, but will maintain some selected information about the past, and either do analysis of this information on-the-fly, or after program termination.

An example is the data race detection algorithm Eraser (Savage et al., 1997) developed at Compaq, and implemented for C++ in the Visual Threads tool (Harrow, 2000). Another example is a locking order analysis called Lock-Tree which we have developed. Both these algorithms have been implemented in JPF to work on Java programs. Below we describe these two algorithms, and how they can be run stand-alone in JPF to identify data race and deadlock potentials in Java programs. Then we describe how these algorithms are used to focus the model checker on part of the state space that contains these potential data race and deadlock problems. Note that runtime analysis is different from runtime monitoring, as supported in systems such as Temporal Rover (Drusinsky, 2000) and MaC (Lee et al., 1999), where certain user-specified properties are monitored during execution. We are, however, currently also exploring the integration of this kind of technology with runtime analysis.

3.3.4.1. Data race detection. The Eraser algorithm detects data race potentials. A concrete data race occurs when two concurrent threads simultaneously access a shared variable and when at least one access is a write; hence the threads use no explicit mechanism to prevent the accesses from being simultaneous. The program is guaranteed data race free if for every variable there is a nonempty set of locks that all threads own when they access the variable. The Eraser algorithm can detect that a data race on a variable is possible (potential) even though no concrete data races have occurred, by observing and remembering which locks are active whenever it is accessed. The algorithm works by maintaining for each variable x a set $set(x)$ of those locks active when threads access the variable. Furthermore, for each thread t a set $set(t)$ is maintained of those locks taken by the thread at any time. Whenever a thread t accesses the variable x, the set $set(x)$ is refined to the intersection between $set(x)$ and $set(t)(set(x) := set(x) \cap set(t))$, although the first access just assigns $set(t)$ to $set(x)$. Our algorithm differs from Savage et al. (1997) since there the initial value of $set(x)$ is the set of all locks in the program. In a Java program objects (and thereby locks) are generated dynamically, hence the set of all locks cannot be pre-calculated. A race condition may be possible if $set(x)$ ever becomes empty.

The simple algorithm described above yields too many warnings as explained in Savage et al. (1997). First of all, shared variables are often *initialized* without the initializing thread holding any locks. The above algorithm will yield a warning in this case, although this situation is safe. Another situation where the above algorithm yields unnecessary warnings is if a thread creates an object, where after several other threads read the object's variables (but no-one is writing after the initialization). To avoid warnings in these two cases, (Savage et al., 1997) suggests an extension to the algorithm by associating a state machine to each variable in addition to the lock set. Figure 4 illustrates this state machine. The variable starts in the VIRGIN state. Upon the first write access to the variable, the EXCLUSIVE state is entered. The lock set of the variable is not refined at this point. This allows for initialization without locks. Upon a read access by another thread, the SHARED state is entered, now with the lock refinement switched on, but without yielding warnings in case the lock set goes empty. This allows for multiple readers (and not writers) after the initialization phase. Finally, if a new thread writes to the variable, the SHARED-MODIFIED state is entered, and now lock refinements are followed by warnings if the lock set becomes empty.

The generic Eraser algorithm has been implemented to work on Java by modifying the JVM^{JPF} to perform this analysis when the `eraser` option is switched on. Each thread is associated with a lock set (a Java object representing a set), and each variable (field) in

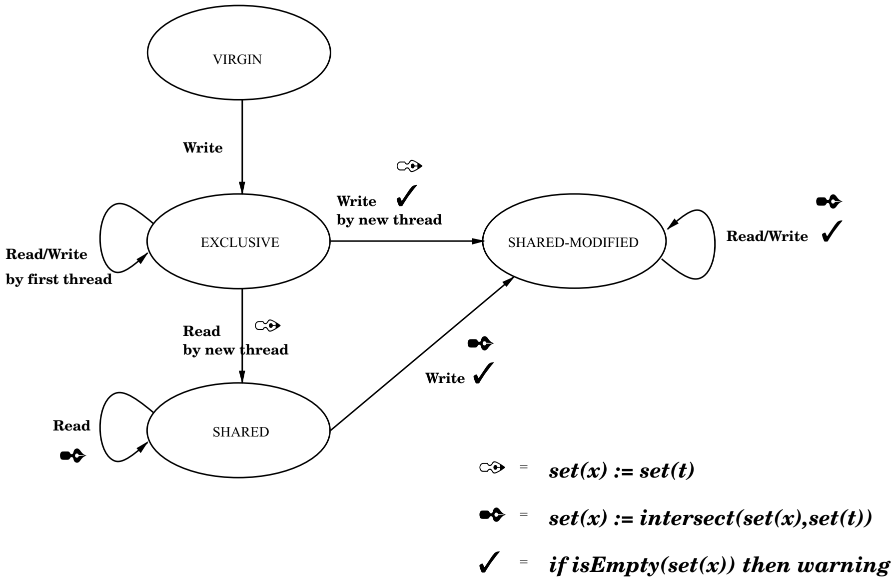

Figure 4. The Eraser algorithm associates a state machine with each variable x. The state machine describes the Eraser analysis performed upon access by any thread t. The pen heads signify that lock set refinement is turned on. The $\surd$ sign signifies that warnings are issued if the lock set becomes empty.

each object is associated with an automata of the type shown in figure 4 (a Java object representing the automata and lock set).

The JVM^{JPF} accesses the bytecodes via the JavaClass package (Java-Class, 2000), which for each bytecode delivers a Java object of a class specific for that bytecode. The JVM^{JPF} extends this class with an `execute` method, which is called by the verification engine, and which represents the semantics of the bytecode. The runtime analysis is obtained by instrumenting the `execute` methods of selected bytecodes, such as the GETFIELD and PUTFIELD bytecodes that read and write object fields, the static field access bytecodes GETSTATIC and PUTSTATIC, and all array accessing bytecodes such as for example IALOAD and IASTORE. The bytecodes MONITORENTER and MONITOREXIT, generated from explicit `synchronized` statements, are instrumented with updates of the lock sets of the accessing threads to record which locks are owned by the threads at any time; just as are the bytecodes INVOKEVIRTUAL and INVOKESTATIC for calling synchronized methods. The INVOKEVIRTUAL bytecode is also instrumented to deal with the built-in `wait` method, which causes the calling thread to release the lock on the object the method is called on. Instrumentations are furthermore made of bytecodes like RETURN for returning from synchronized methods, and ATRHOW that may cause exceptions to be thrown within synchronized contexts.

3.3.4.2. Deadlock detection. A classical deadlock situation can occur where two threads share two locks and attempt to take the locks in different order. An algorithm that detects

```
Thread 1:                          Thread 2:
synchronized(L1){                  synchronized(L1){
  synchronized(L3){                  synchronizd(L2){
    synchronized(L2){};                synchronized(L3){}
    synchronized(L4){}               }
  }                                };
};
synchronized(L4){                  synchronized(L4){
  synchronized(L2){                  synchronized(L3){
    synchronized(L3){}                 synchronized(L2){}
  }                                  }
}                                  }
```

Figure 5. Synchronization behavior of two threads.

such lock cycles must in addition take into account that a third lock may protect against a deadlock like the one above, if this lock is taken as the first thing by both threads, before any of the other two locks are taken. In this situation no warnings should be emitted. Such a protecting third lock is called a *gate lock*.

The algorithm for detecting this situation is based on the idea of recording the locking pattern for each thread during runtime as a *lock tree*, and then, when the program is terminated, comparing the trees for each pair of threads. The lock tree that is recorded for a thread represents the nested pattern in which locks are taken by the thread. As an artificial example, consider the code fragments of two threads in figure 5. Each thread takes four locks L1, L2, L3 and L4 in a certain pattern. For example, the first thread takes L1; then L3; then L2; then it releases L2; then takes L4; then releases L4; then releases L3; then releases L1; then takes L4; etc.

This pattern can be observed, and recorded in a finite tree of locks for each thread, as shown in figure 6, by just running the program. As can be seen from the trees, a deadlock is potential because thread 1 in its left branch locks L3 (node identified with 2) and then L4 (4), while thread 2 in its right branch takes these locks in the opposite order (11, 12). There are furthermore two additional ordering problems between L2 and L3, one in the two left branches (2, 3 and 9, 10), and one in the two right branches (6, 7 and 12, 13). However, neither of these pose a deadlock problem since they are protected by the *gate locks* L1 (1, 8) respectively L4 (5, 11). Hence, one warning should be issued.

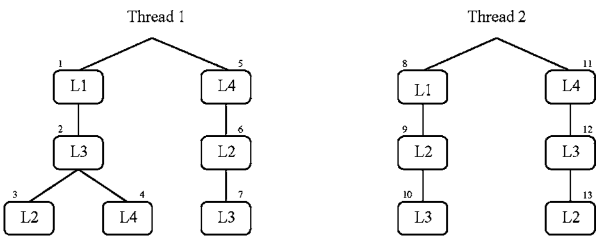

Figure 6. Lock trees corresponding to threads in figure 5.

When being built, each tree has at any time a *current node*, where the path from the root (identifying the thread) to that node represents the *lock nesting* at this point in the execution. The lock operation creates a new child of the current node if the new lock has not previously been taken (is not in the path above). The unlock operation just backs up the tree if the lock really is released, and not owned by the thread in some other way. When the program terminates, the analysis of the lock trees is initiated. Each pair of trees $(t_1; t_2)$ are compared, and for every node n in t_1 it is checked that no node below n is above any occurrence of n in t_2. In order to avoid issuing warnings when a *gate lock* prevents a deadlock, occurrences of n in t_2 are marked after being examined, and nodes below marked nodes are not considered until the marks are removed when the analysis backtracks from the corresponding node in t_1. The following bytecodes will activate calls of the lock and unlock operations in these tree objects for the relevant threads: MONITORENTER and MONITOREXIT for entering and exiting monitors, INVOKEVIRTUAL and INVOKESTATIC for calling synchronized methods or the built-in `wait` method of the Java threading library, bytecodes like RETURN for returning from synchronized methods, and ATRHOW that may cause exceptions to be thrown within synchronized contexts.

3.3.4.3. Using runtime analysis to guide model checking. The runtime analysis algorithms described in the previous two sections can provide useful information to a programmer as stand alone tools. In this section we will describe how runtime analysis furthermore can be used to guide a model checker. The basic idea is to run the program in simulation mode first, using the JVM^{JPF} simulator, with all the runtime analysis options turned on, thereby obtaining a set of warnings about data races and lock order conflicts. The threads causing the warnings are stored in a *race window*.When the simulation is terminated, forced or according to the program logic, the resulting race window (in fact an extension of it, see below) will then be fed into the model checker, which will now search the state space, but now only focusing its attention on the threads in the window. That is, the model checker only schedules threads that are in the window.

However, before the model checker is applied, the race window is extended to include threads that create or otherwise influence the threads in the original window. The purpose is to obtain a small self-contained sub-system containing the race window, which can be meaningfully model checked. The extended window can be thought of as a dynamic slice of the program. The extension is calculated on the basis of a *dependency graph*, created by a dependency analysis also performed during the pre-simulation. More specifically, the dependency graph is a mapping from threads t to triples $(\alpha; \rho; \omega)$, where α is the ancestor thread that spawned t, ρ is the set of objects that t reads from, and ω is the set of objects that t writes to. The window extension operation performs a fix-point calculation by creating the set of all threads *reachable* from the original window by repeatedly including threads that have spawned threads in the window, and by including threads that write to objects that are read by threads in the window. The following bytecodes are instrumented to operate on the dependency graph: INVOKEVIRTUAL for invoking the start method on a thread; and PUTFIELD, GETFIELD, PUTSTATIC, GETSTATIC for accessing variables.

4. Integration with BANDERA

In this paper we argue the virtues of analyzing source code, but in order for such analysis to be useful to the software development community, one also requires the tools to be user-friendly. Unlike in the formal methods community where a textual interface will suffice due to the expert knowledge of the users, here we are interested in our tools to be used by real programmers and therefore ease of use is paramount. In this sense JPF was initially lacking, since when an error was discovered it would result in a textual output of each source line that was to be executed to get to the error.

We therefore decided to integrate JPF with the BANDERA tool (Corbett et al., 2000a), since we could use their error-displaying capabilities, which allow the user to step through the code line by line, forwards and backwards, while also having the capability to observe any object in memory. The integration was straight-forward due to the modular design principles adhered to by both projects and the fact that both systems were written in Java-the entire integration required two weeks by two developers, one from each project, working together. The integration also had the added bonus for us that it allowed the use of the BANDERA front-end tools, namely a slicer (see Section 3.3.3) and an abstractor and allowed us to express user-defined assertions, as well as pre- and postconditions to methods as Javadoc comments. BANDERA on the other hand gained a powerful Java model checker to augment Spin and SMV, since both have restrictions as to which Java programs can be checked.

5. Related work

Although we have mentioned some works that are related to ours in the Java context there are also two significant projects where the target language for model checking is C: the SLAM project at Microsoft and the FeaVer model checker at Lucent. These model checkers have in common that they both rely heavily on abstraction techniques to create a finite-state model from C code that can then be analyzed.

5.1. SLAM

The aim of this work is to do reachability analysis for large sequential C programs, with specific application to device drivers (Ball and Rajamani, 2000b; Ball et al., 2001b). The project, in a similar fashion to ours, focuses on the combination of many different techniques to accomplish this goal: static analysis, abstraction, symbolic execution and model checking. Specifically a model checker for boolean programs is used (Ball and Rajamani, 2000a), i.e. all the variables in the program are of the boolean type. The basic idea is to abstract the original C program by extracting the control-flow graph, then to check reachability of a program statement. Assuming the control-flow graph is not disconnected, the statement is reachable. Next the path of instructions to the statement is symbolically executed on the original program, and when a divergence is encountered (the path that is being executed differs from what can be executed next—by definition this must happen at a choice point in the program) a boolean variable is created to capture this choice point, in other words a new boolean program is created that makes this path infeasible. Of course, if no divergence is encountered then reachability has been shown. Next, the process is repeated with the new boolean program where the infeasible path is removed. The problem is of course that showing that a path is (in)feasible can be undecidable, and if this happens the checker returns a "don't know" result to the reachability question.

The most striking difference with our approach is that this work is only for sequential programs, but recently they have started to also consider multithreaded C programs (Ball et al., 2001a). Furthermore, they use a similar predicate abstraction to ours, but they start by abstracting the program to its most over-approximated state, and then re-introducing predicates to build the program up to one where reachability can either be shown or not. Whereas we start with the complete program and only use abstraction selectively to remove information in parts of the program.

5.2. FeaVer

FeaVer is a software model checking system based on the Spin model checker and was used to verify properties of Lucent's PathStar access server. The system mechanically extracts (Holzmann, 2000) a verification model from unedited C code, and verifies it against a library of logic properties (Holzmann and Smith, 1999, 2000). The abstraction process here is semi-automated in the sense that a user-defined lookup table is used to automatically translate C code to Promela code, i.e. each C source line is mapped via the table to a line of Promela code. Abstraction occurs since very complex lines of C code can be replaced by simple abstract code in Promela, for example, a function call can be replaced by a "skip" command if the call bears no significance to the verification problem. This idea might seem straight-forward, but it worked very well for the PathStar model checking since the code was relatively stable, and hence although the manual creation of the table took some effort, it was then very stable and could be reused with small modifications whenever the code changed.

The significance of this work is two-fold: firstly, it is, to the best of our knowledge, the first case where model checking was used to analyze a large software application in a commercial setting and secondly, the model checking found an order of magnitude more errors in the code than the traditional testing team (75 versus 5) (Holzmann and Smith, 2000). The Spin model checker is currently being extended to analyze C code in a more direct fashion rather than using the lookup table.

6. Applications of JPF tools

In this section we describe the application of JPF and its related tools to two real-world examples. The first is a model of a spacecraft controller (Section 6.1) which we use to illustrate how JPF can find errors that were introduced in the coding phase (i.e. after design). This example also illustrates how the different techniques used in JPF can be combined. The second example is a real-time operating system (Section 6.2) with a subtle error in the time-partitioning of threads, that is in fact an example of an error that was introduced during design, but was not discovered during the design due to a lack of detail.

6.1. The remote agent spacecraft controller

The Remote Agent (RA) is an AI-based spacecraft controller that has been developed at NASA Ames Research Center. It consists of three components: a Planner that generates plans from mission goals; an Executive that executes the plans; and finally a Recovery system that monitors the RA's status, and suggests recovery actions in case of failures. The Executive contains features of a multi-threaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multi-threading errors. In fact, during real flight in May 1999, the RA deadlocked in space, causing the ground crew to put the spacecraft on standby. The ground crew located the error using data from the spacecraft, but asked as a challenge to our group if we could locate the error using model checking. This resulted in an effort described in (Havelund et al., 2000), which we shall shortly describe in the following. Basically we identified the error using a combination of code review, abstraction, and model checking using JPF1, the first generation of Java PathFinder. During code review we got a suspicion about the error since it resembled one discovered using the Spin model checker before flight (Havelund et al., 1998). The modeling therefore focused on the code under suspicion for containing the error. What we will describe in the following is the abstraction process using the abstraction tool, which also works for the new generation of JPF.

The major two components to be modeled were events and tasks, as illustrated in figure 7. The figure shows a Java class `Event` from which event objects can be instantiated. The class

```
class Event {
  int count = 0;
  public synchronized void wait_for_event() {
    try{wait();}catch(InterruptedException e){};
  }
  public synchronized void signal_event(){
    count = count + 1;
    notifyAll();
} }

class Planner extends Thread{
  Event event1,event2;
  int count = 0;
  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      /* Generate plan */
      event2.signal_event();
} } }
```

Figure 7. The RAX error in Java.

has a local counter variable and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called `wait_for_event`. In order to catch 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 new events since it was last restarted from a call of `wait_for_event`. The figure shows the definition of one of the tasks. The task's activity is defined in the `run` method of the class `Planner`, which itself extends the `Thread` class, a built-in Java class that supports thread primitives. The body of the `run` method contains an infinite loop, where in each iteration a conditional call of `wait_for_event` is executed. The condition is that no new events have arrived, hence the event counter is unchanged.

The program shown has theoretically infinitely many reachable states due to the repeated increment of the count variable in the events. We use abstraction to remove these variables by specifying `Abstract.remove(count)` in the classes of `Event` and `Planner`. In place of these variables, we declare abstraction predicates corresponding to those predicates in the program that involve `count` variables. For instance, in the definition of the `Planner` class we put `Abstract.addBoolean("EQ",count == event1.count)`. After having annotated the program with these abstraction declarations, the abstraction tool is applied and a new abstracted program is generated. JPF thereafter reveals the deadlock in this abstracted program. The error trace shows that the Planner first evaluates the test "`(count == event1.count)`", which evaluates to true; then, before the call of `event1.wait for event()` the Executive signals the event, thereby increasing the event counter and notifying all waiting threads, of which there are none. The Planner now unconditionally waits and misses the signal. The solution to this problem is to enclose the conditional wait in a critical section such that no events can occur in between the test and the wait. In fact, the same pattern occurred in several places and in all other places there was such a critical section around. This was simply an omission.

The abstract Java model of what happened on board the spacecraft was created based on a suspicion about the source of the error obtained during code review. This suspicion was created by the fact that this same pattern had been found to cause errors in a different part of the RA during the pre-flight effort using the Spin model checker two years before (Havelund et al., 1998). The source of the error, a missing critical section, could, however, have been found automatically using the Eraser data detection algorithm. The variable `count` in class `Event` is accessed unsynchronized by the Planner's run method in the line: "`if (count == event1.count)`", specifically the expression: `event1.count`. Hence even though the `signal event` called by the Executive will increase the variable synchronized, the above condition in the Planner can be executed even during such a signal. This may cause a data race where the `count` variable is accessed simultaneously by the Planner and the Executive. When running JPF in Eraser mode, it detects this race condition immediately. This could be enough to locate the error, but only if one can see the consequences. The JPF model checker, on the other hand, can be used to analyze the consequences.

To illustrate JPF's integration of runtime analysis and model checking, the example was made slightly more realistic by adding extra threads that made the Java program resemble the real system. The new program had more than 10^{60} states. Then we applied JPF in its special runtime analysis/model checking mode. It immediately identified the race condition using the Eraser algorithm, and then launched the model checker on a thread window consisting of those threads involved in the race condition: the Planner and the Executive, locating the deadlock—all within 25 seconds. As an additional experiment in collaboration with the designers of the BANDERA tool, we fed part of the result of the race detection, namely the variable that is accessed unprotected, into BANDERA's slicing tool, which in turn created a program slice where all code irrelevant to the value of the counter had been removed. JPF then found the deadlock on this sliced program. This illustrates our philosophy of integrating techniques from different disciplines: abstraction was used to turn an infinite program into a finite one, runtime analysis was used to pinpoint problematic code, slicing was used to reduce the program, and finally the model checker was launched to analyze the result (see figure 2).

6.2. The DEOS avionics operating system

The DEOS real-time operating system, developed by Honeywell for use within business aircraft, is written in C++. During a manual analysis of the code the developers noticed a subtle error in the system, that testing had not picked up. Without relating what the error was, a slice of the original code, that contained the error, was handed over to NASA Ames with the goal being to see whether a model checker could find the error. The error was subsequently found after a translation of the code to Promela. A full account of this verification exercise can be found in Penix et al. (2000). Since the slice of DEOS is fairly large, approximately 1000 lines of C++, and the error very subtle, it seemed like a good candidate on which to validate our philosophy of model checking code directly. As a first step the C++ code was translated to Java; this was straight-forward, since the original C++ code contained very little pointer arithmetic etc. This resulted in 14 Java classes containing approximately 1000 lines of code. The DEOS system must be put in parallel with a nondeterministic environment in order to do model checking. Luckily the environment created for the Promela model could be re-used (by translation into Java) to a large extent. This added another 6 classes to the system, for a combined total of 1443 lines of Java code, making it by far the largest example (in terms of lines of code) attempted by JPF at the time (since then we have handled programs at least 3 times larger). We also decided to use a local assertion check to indicate when the error occurred, instead of the LTL property used during the SPIN analysis (prompted by the fact that at the time the tool did not support LTL checking). The major benefit of the assertion check was that it flagged the error when it happened, rather than sometime later as was the case with the LTL check. The assertion check was very complex to calculate, 75 lines of Java code, since it involved many of the internal variables used within DEOS (it was developed by one of the DEOS developers). Note this assertion check was written in Java and was specific to the DEOS program, it had no relationship with the original LTL property used.

As with the Spin version we started off by limiting the search-depth of the model checker, since the original system had infinitely many states. Initial runs were discouraging, since the error was not found after running the system for hours. However when partial-order reductions were switched on the error was found almost instantly. As in the Promela version, large parts of the system are executed in atomic steps. In the Promela version we applied a predicate abstraction by hand to reduce the system to finitely many states, the next step will be to do the same with our Java abstraction tool automatically—the current version of the tool cannot handle the abstraction of predicates over arrays, which is a requirement in this case.

Recently the BANDERA group have also looked at the analysis of the Java version of DEOS with a combination of BANDERA and JPF and had some very encouraging results (Pasareanu et al., 2001; Dwyer et al., 2001). They used dependency analysis driven by the location of the time-partitioning assertion and the data values that it referenced to identify a single field (out of the 92 fields in the program) as influencing the property. A *range* abstraction (where only values 0 and 1 are concrete and all negative numbers and all numbers greater than 1 map to abstract domains) was then used to abstract this field followed by a type inference to determine that two other fields also required abstracting to the *range*. JPF, with the special choose-free mode (see Section 3.3.2), was then invoked to find the error in just 312 steps (down from 471 in the normal mode). Note that running JPF in the choosefree mode was essential since the range abstraction generates many spurious errors. This again shows the power of static analysis and abstraction when model checking programs.

7. Conclusions and future work

In the first part of this paper we argued why the formal methods subgroup of the software engineering community should devote some of their efforts to the analysis of systems described in real programming languages, rather than just to their own special purpose notations. The second part of the paper described how we applied this philosophy to the analysis of Java programs. Specifically, we have shown that model checking can be applied to Java programs, without being hampered by the perceived problems often cited as reasons for why model checking source code will not work. In the process we have shown that augmenting model checking with symmetry reductions, abstract interpretation, static analysis and runtime analysis can lead to the efficient analysis of complex (Java) software. Although the combination of some of these techniques is not new, to the best of our knowledge, our use of symmetry reductions for class loading and heap allocation, the semi-automatic[5] predicate abstraction across different classes, the use of static analysis to support partial-order reductions and the use of runtime analysis to support model checking are all novel contributions.

Although it is hard to quantify the exact size of program that JPF can currently handle—"small" programs might have "large" state-spaces—we are routinely analyzing programs in the 1000 to 5000 line range.

Since we are drawing on different techniques and the synergy between these techniques it should be clear that many areas for future research exist. Besides the obvious extensions and improvements of the different algorithms, there are two areas which we feel are crucial to the success of applying model checking to (Java) source code. Firstly, one needs to develop methods to assist in the construction of "environments" suitable for model checking. Currently the users of a model checker will construct an environment for their models by hand, but we believe some automation will be required if non-experts are to use the (Java) model checker. Secondly, it is naive to believe that model checking will be capable of analyzing programs of 100000 lines or more, hence in these cases one would like to have a "measure" of how much of the system was checked. In software testing this measure is given as a coverage measure and hence we are currently investigating means to calculate typical coverage measures (for example, branch coverage, method coverage, condition/decision coverage, etc.) during model checking with JPF. Lastly, we are also working on the analysis of C/C++ programs, by doing a translation from C/C++ to (extended) Java bytecode, and using a modified version of the JPF model checker.

Acknowledgments

We would like to thank the BANDERA group at Kansas State University, specifically, Matt Dwyer, John Hatcliff, Corina Pasareanu and Robby, for letting us use their tools and for the great support they have given us.

Notes

1. For example, there already exist translators from Eiffel, Ada, OCAML, Scheme and Prolog to bytecode.
2. http://www.inf.fu-berlin.de/~dahm/JavaClass/
3. All the tables are implemented as hashtables, and in some cases the "index" used will be a reference to an object rather than an integer value.
4. "NEW" refers to any bytecode instruction that allocates a new object, hence including allocation of arrays and string constants.
5. Although the user must select the predicates, once selected, the generation of the abstract program is automatic.

References

Ball, T., Chaki, S., and Rajamani, S. 2001a. Parameterized verification of multithreaded software libraries. In *Proceedings of TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*. Genova, Italy.

Ball, T., Podelski, A., and Rajamani, S. 2001b. Boolean and Cartesian abstractions for model checking C programs. In *Proceedings of TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy.

Ball, T. and Rajamani, S. 2000a. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop*, vol. 1885 of LNCS. Stanford University, California, USA, Springer-Verlag.

Ball, T. and Rajamani, S. 2000b. Checking temporal properties of software with Boolean programs. In *Proceedings of Workshop on Advances in Verification*.

Barrett, C., Dill, D., and Levitt, J. 1996. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design*, vol. 1166 of LNCS, pp. 187–201.

Binkley, D. 1993. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2:31–45.

Bjørner, D. and Jones, C.B. (eds.) 1982. *Formal Specification and Software Development*. Prentice-Hall International.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide*. Addison-Wesley.

Cheng, J. 1997. Dependence analysis of parallel and distributed programs and its applications. In *Proceedings of the 1997 Conference on advances in Parallel and Distributed Computing*.

Clarke, E., Emerson, E., Jha, S., and Sistla, A. 1998. Symmetry reductions in model checking. In *Proceedings of the 10th International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 1427.

Clarke, E., Filkorn, T., and Jha, S. 1993. Exploiting symmetry in temporal logic model checking. In *Proceedings of the Fifth International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 697.

Clarke, E., Fujita, M., Rajan, S., Reps, T., Shankar, S., and Teitelbaum, T. 1999. Program slicing of hardware description languages. Technical Report CMU-CS-99-103, Carnegie Mellon University, School of Computer Science.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In. *Proceedings of the 12th International Conference for Computer-Aided Verification*. Lecture Notes in Computer Science, 1855.

Colón, M. and Uribe, T. 1998. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, vol. 1427 of LNCS.

Corbett, J., Dwyer, M., Hatcliff, J. Pasareanu, C., Robby, Laubach, S., and Zheng, H. 2000a. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*. Limeric, Ireland, ACM Press.

Corbett, J.C., Dwyer, M.B., Hatcliff, J., and Robby 2000b. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the 7th International SPIN Workshop*, vol. 1885 of Lecture Notes in Computer Science, Springer-Verlag.

Cornes, C., Courant, J., Filliatre, J., Huet, G., Manoury, P., Paulin-Mohring, C., Munoz, C., Murthy, C., Parent, C., Saibi, A., and Werner, B. 1995. The Coq proof assistant reference manual, version 5.10. Technical Report, INRIA, Rocquencourt, France. This version is newer than the version used to verify the BRP-protocol in Helmink et al. (1994).

Cousot, P. and Cousot, R. 1992. Abstract interpretation frameworks. *Journal of Logic and Computation*, 4(2):511–547.

Cousot, P. and Cousot, R. 1997. Parallel combination of abstract interpretation and model-based automatic analysis of software. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, AAS'97*. pp. 91–98.

Das, S., Dill, D., and Park, S. 1999. Experience with predicate abstraction. In *CAV '99: 11th International Conference on Computer Aided Verification*, vol. 1633 of LNCS.

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

Demartini, C., Iosif, R., and Sisto, R. 1999b. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 6th SPIN Workshop*, vol. 1680 of LNCS.

Drusinsky, D. 2000. The temporal rover and the ATG rover. In K. Havelund, J. Penix, and W. Visser, editors. *SPIN Model Checking and Software Verification*, vol. 1885 of Lecture Notes in Computer Science, Springer, pp. 323–330.

Dwyer, M., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C., Robby, Visser, W., and Zheng, H. 2001. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, ACM Press.

Emerson, E. and Sistla, A. 1993. Symmetry and model checking. In *CAV '93: 5th International Conference on Computer Aided Verification*, vol. 697 of Lecture Notes in Computer Science.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems*, vol. 1032 of LNCS, Springer-Verlag.

Godefroid, P. 1997. Model checking for programming languages using veriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, pp. 174–186.

Gordon, M.J.C. 1988. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Dordrecht, The Netherlands: Kluwer, pp. 73–128.

Graf, S. and Saidi, H. 1997. Construction of abstract state graphs with PVS. In *CAV '97: 6th International Conference on Computer Aided Verification*, vol. 1254 of LNCS.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Harrow, J. 2000. Runtime checking of multithreaded applications with visual threads. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, vol. 1885 of Lecture Notes in Computer Science, Springer, pp. 331–342.

Hatcliff, J., Corbett, J., Dwyer, M., Sokolowski, S., and Zheng, H. 1999. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings on the 1999 International Symposium on Static Analysis*, pp. 1–18.

Havelund, K. 1999a. Java PathFinder, a translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking—5th and 6th International SPIN Workshops*, vol. 1680 of LNCS, Springer-Verlag. Trento, Italy—Toulouse, France (presented at the 6th Workshop).

Havelund, K. 1999b. Mechanical verification of a Garbage collector. In D. Méry and B. Sanders, editors, *FMPPTA'99: Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, Springer-Verlag. San Juan, Puerto Rico, USA.

Havelund, K. 2000. Using runtime analysis to guide model checking of Java programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, vol. 1885 of Lecture Notes in Computer Science, Springer, pp. 245–264.

Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., and White, J. 2000. Formal analysis of the remote agent before and after flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*.

Havelund, K., Lowry, M., and Penix, J. 1998. Formal analysis of a space craft controller using SPIN. In *Proceedings of the 4th SPIN workshop*, Paris, France. To appear in IEEE Transactions of Software Engineering.

Havelund, K. and Pressburger, T. 1999. Model checking Java programs using Java PathFinder. To appear in a special issue of *International Journal on Software Tools for Technology Transfer* (STTT) containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.

Havelund, K. and Shankar, N. 1996. Experiments in theorem proving and model checking for protocol verification. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, vol. 1051 of LNCS, Springer-Verlag, pp. 662–681.

Havelund, K. and Skakkebaek, J. 1999. Practical application of model checking in software verification. In *Proceedings of the 6th Workshop on the SPIN Verification System*, vol. 1680 of LNCS, Toulouse, France.

Helmink, L., Sellink, M., and Vaandrager, F. 1994. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), Computer Science/Department of Software Technology.

Hoare, C.A.R. 1969. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580.

Holzmann, G. 1997a. State compression in Spin. In *Proceedings of the Third Spin Workshop*. Twente University, The Netherlands.

Holzmann, G. 1997b. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295. Special issue on Formal Methods in Software Practice.

Holzmann, G. 2000. Logic verification of ANSI-C code with Spin. In *Proceedings of the 7th International SPIN Workshop*, vol. 1885 of LNCS, Springer Verlag, pp. 131–147.

Holzmann, G. and Peled, D. 1994. An improvement in formal verification. In *Proc. FORTE94*, Berne, Switzerland.

Holzmann, G. and Smith, M.H. 1999. Software model checking—Extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, Kluwer Academic Publ., pp. 481–497.

Holzmann, G. and Smith, M.H. 2000. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87. Issue on Software Complexity.

Iosif, R. and Sisto, R. 2000. Using garbage collection in model checking. In *Proceedings of the 7th International SPIN Workshop*, vol. 1885 of LNCS, Stanford University, California, USA, Springer-Verlag.

Ip, C. and Dill, D. 1993. Better verification through symmetry. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Application*, North Holland.

JavaClass: 2000, 'JavaClass'. http://www.inf.fu-berlin.de/~dahm/JavaClass/.

Larsen, K.G., Pettersson, P., and Yi, W. 1998. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1/2):134–152.

Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M. 1999. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

Lerda, F. and Visser, W. 2001. Addressing dynamic issues of program model checking. In *Proceedings of the 8th International SPIN Workshop*, vol. 2057 of LNCS 2057, Springer-Verlag.

McMillan, K. 1993. *Symbolic Model Checking*. Boston: Kluwer Academic Publishers.

Melton, R., Dill, D., Ip, C.N., and Stern, U. 1996. Murphi annotated reference manual, release 3.0. Technical Report, Stanford University, Palo Alto, California, USA.

Millett, L.I. and Teitelbaum, T. 1998. Slicing promela and its application to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. 1998. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1/2):5–48.

Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. 1996. PVS: Combining specifi-cation, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*. New Brunswick, NJ, Springer-Verlag, pp. 411–414.

Park, D., Stern, U., Skakkebaek, J., and Dill, D. 2000. Java model checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pp. 253–256.

Pasareanu, C., Dwyer, M., and Visser, W. 2001. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy.

Penix, J., Visser, W., Engstrom, E., Larson, A., and Weininger, N. 2000. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland, ACM Press.

Russinoff, D.M. 1994. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390.

Saidi, H. 1999. Modular and incremental analysis of concurrent software systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pp. 92–101.

Saidi, H. 2000. Model checking guided abstraction and analysis. In *Proceedings of the 7th Static Analysis Symposium*.

Saïdi, H. and Shankar, N. 1999. Abstract andModel check while you prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, vol. 1633 of LNCS, pp. 443–454.

Savage, S., Burrows, M., Nelson, G., and Sobalvarro, P. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

Spivey, M. 1992. *The Z Notation: A Reference Manual*, 2nd edn. Prentice Hall: International Series in Computer Science.

Stoller, S. 2000. Model-checking multi-threaded distributed Java programs. In *Procceedings of the 7th International SPIN Workshop*, vol. 1885 of LNCS, Stanford University, California, USA, Springer-Verlag.

The RAISE Language Group 1992. *The RAISE Specification Language*. Prentice-Hall: The BCS Practitioners Series.

Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189.

Valle-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. 1999. Soot—a Java optimization framework. In *Proceedings of CASCON 1999*.

Visser, W., Havelund, K., and Penix, J. 1999. Adding active objects to SPIN. In *Proceedings of the 5th Workshop on the SPIN Verification System*, Trento, Italy.

Visser, W., Park, S., and Penix, J. 2000. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*.

Weiser, M. 1984. Program slicing. *IEEE Transaction on Software Engineering*.