Model Checking Programs

Willem Visser
RIACS/NASA Ames Research Center
Moffet Field, CA 94035, USA
wvisser@ptolemy.arc.nasa.gov

Klaus Havelund
QSS/NASA Ames Research Center
Moffet Field, CA 94035, USA
havelund@ptolemy.arc.nasa.gov

Guillaume Brat
Kestrel/NASA Ames Research Center
Moffet Field, CA 94035, USA
brat@ptolemy.arc.nasa.gov

SeungJoon Park
RIACS/NASA Ames Research Center
Moffet Field, CA 94035, USA
spark@ptolemy.arc.nasa.gov

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 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, 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 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.

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 [24], has been devoted to special languages that differ from main stream programming languages. Typical examples are formal specification languages [40, 2, 39], purely logic based languages used in theorem provers [13, 33, 7], and guarded command languages used in model checkers [30, 29, 28]. In a few cases, modeling languages have been designed to resemble programming languages [26], although the focus has been on protocol designs. Some of these linguistic choices have made, and still make it feasible to more conveniently experiment 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) [16, 20] 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. Section 4 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 errors were errors in the real code of these systems. Finally, Section 5 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 [15], such as for example found in UML [3]. However, we want to argue that the formal methods community should put some of its attention on programs for a number of reasons that we will describe below.

First of all, programs often contain fatal errors in spite of 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 [19]. 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 [18]; see Section 4.1. 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 [21, 23] and garbage collection algorithms [17, 35]. The other kind of errors are more simple minded concurrency programming errors, such as forgetting to put code in a critical section or causing deadlocks. This kind of errors 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 4.2, 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 preconstructed design. This was for example the case with the Remote Agent [32] 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 may drive the 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 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 lim-

[1] For example, there already exist translators from Eiffel, Ada, OCAML, Scheme and Prolog to bytecode.

ited 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 due to the fact that 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 criteria based on a runtime analysis, and their tool could slice the Java program based on this criteria, 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 to anyone for writing 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 why model 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 [26], 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 [20], as well as the JCAT system [10], were based on a translation from Java to Promela. Although both these systems were successful in model checking some interesting Java programs [22], such source-to-source translations suffer from two serious drawbacks:

Language Coverage — Each language feature of the source language must have a "counterpart" 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 rather doing a translation from bytecodes. This is the approach used by the BANDERA tool [6], where bytecodes, after some manipulation, are translated to either Promela or the SMV model checker's input notation.

3.1.2 Custom-made Model Checker

In order to overcome the language coverage problem it is however 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 [11, 42], 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 [29]. 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 extend 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, that 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 12 man-months. The current model checker can only check for deadlocks, invariants and user-defined assertions in the code; temporal logic model checking will be added in the near future

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 [12] 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, that is in general highly optimized for speed, but will not allow the memory to be encoded easily.

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: 512Mb 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 [25]: 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 stored 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]. 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 changed is the frame representing the method, all the other parts of the system state is unaffected and will *collapse* to the same indexes. This actually alludes to the other simple 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). Currently the system can store millions of states in 512Mb and evaluates between 500 and 1500 states per second depending on the size of the state (on a SPARC ULTRA60).

JPF in its current state already illustrates that software systems with complex states can be efficiently analyzed (see section 4), but with some further extensions and better hardware platforms to run it on, we believe, systems of up to 10k lines of code could be analyzed.

3.3 Curbing the State-explosion

Maybe the most challenging part of model checking is reducing the size of the state-space 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

[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.

by itself to programs will not scale to programs of much more than 10k 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 abstract interpretation, static analysis and runtime analysis to allow more efficient model checking of Java programs.

3.3.1 Abstraction

Recently, the use of abstraction algorithms based on the theory of abstract interpretation [8], have received much attention in the model checking community [14, 9, 36, 37, 5]. 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, and 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 [14, 36, 9] or automatically generates an abstract system, that manipulates the abstract data, which can then be model checked [37, 5]. 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 proportional 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) [1], 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 the `randomBool()` method indicates a nondeterministic choice. The BANDERA tool uses similar techniques to abstract the data-domains of say an integer variable in Java to work over the *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, if class A has a field x and class B has a field y then we can specify a predicate `Abstract.addBoolean("xGTy", A.x > B.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 [43] 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 has finite-state and can be checked exhaustively. In section 4.1 we also show how the abstraction tool is used on a real example.

3.3.2 Static Analysis

Static analysis is a technique often used, in all areas of software engineering, to achieve a reduction in program size. Only comparatively recently has there been any activity in using it to reduce the size of systems before model checking. Specifically, it was noticed that slicing [41], can be a useful way of reducing program size to allow more efficient model checking [31, 4]. The best exponent of using slicing to reduce Java programs for model checking is the BANDERA tool [6], where they use the variables occurring in an LTL formula in their slicing criteria. We believe this is a very interesting avenue for further research and are currently in the process of interfacing JPF with the BANDERA tool.

Within JPF we are currently using static analysis techniques 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 [27] 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.

3.3.3 Runtime Analysis

Runtime analysis is conceptually based on the idea of executing the 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 demonstrate whether the generated

trace violates such properties). Note 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. These algorithms typically will not guarantee that errors are found since they work on an arbitrary trace. They also may yield false positives. 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. 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 [38] developed at Compaq. Another example is a locking order analysis called LockTree which we have developed. Both these algorithms have been implemented in JPF. Below we describe these two algorithms, and then describe how they are integrated in JPF to run stand-alone, or integrated with model checking to reduce the state space.

The Eraser algorithm detects data races. A data race occurs when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The program is 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 works by maintaining for each variable x a set L_x of those locks active when threads access the variable. Furthermore, for each thread t is maintained a set L_t of those locks taken by the tread at any time. Whenever a thread t accesses the variable x, the set L_x is refined to the intersection between L_x and L_t ($L_x = L_x \cap L_t$), although the first access just assigns L_t to L_x. A race condition may be potential if L_x ever becomes empty. The algorithm described in [38] is relaxed to allow variables to be initialized without locks, and to be read by several threads without locks, if no-one writes.

The LockTree algorithm looks for potential deadlocks by detecting differences in the order in which threads take locks. A classical deadlock situation can be defined as one thread T_1 accessing two Locks K and L, in that order, while another thread accesses them in the reverse order. The deadlock may then occur if T_1 takes K, and then T_2 takes L. Now none of the threads can continue. If we define deadlock in this limited way, a program is deadlock free if all locks are accessed in the same order. The LockTree algorithm searches for the violation of such an ordering between locks. It maintains a tree of lock orders for each thread, and compares these trees at the end of an execution. This is in contrast with the Eraser algorithm which does the analysis on-the-fly.

Runtime analysis can be used in two modes within JPF. It can first of all be used stand-alone in simulation mode. Second, runtime analysis can be used to guide the model checker. We have made experiments where the Eraser module in JPF generates a so-called *race window* consisting of the threads involved in a race condition. The model checker is then launched, focusing on the race window by forcing the scheduler always to pick threads in the window before other threads. In the near future, we plan to perform runtime analysis during the model checking itself.

4 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 4.1) in which we 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 4.2) 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.

4.1 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 our group if we could locate the error using model checking. This resulted in an effort described in [18], and 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 [19]. 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 1. The figure shows a

Java class `Event` from which event objects can be instantiated. The class 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.

```
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 1. The RAX Error in Java

The shown program has theoretically infinitely many reachable states due to the repeated increment of the count variable in the events. We use abstraction to remove those `count` 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, we put `Abstract.addBoolean("EQ",count==event1.count)` in the definition of the `Planner` class. 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 [19]. 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.

4.2 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 can 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 [34]. Since the slice of DEOS is fairly large, ± 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) ever attempted by JPF. One change that was required in the Java version of the model checking was that we had to create an invariant that would show when the error occurred, since the Promela version used an LTL formula, which our current system does not support. This invariant is fairly complex, 92 lines of Java, and was created by one of the developers of the DEOS system.

As with the Spin version we started off by limiting the search-depth of the model checker, since the original system has 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. In fact, much faster than Spin found the error, but the Promela and Java versions are not identical and hence one should read nothing into this result (for example, the order of nondeterministic choices are different). As in the Promela version, large parts of the system is 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.

5 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 showed that model checking could 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 showed that augmenting model checking with abstract interpretation, static analysis and runtime analysis can lead to the efficient analysis of complex (Java) software. Although the combination of some these techniques are not new, to the best of our knowledge, our use of automatic 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.

Since we are drawing on different techniques and the synergy between these techniques it should be clear that many areas for future research exists. 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 need 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 100k 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.

References

[1] C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *FMCAD'96*, volume 1166 of *LNCS*, November 1996.

[2] D. Bjøner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall International, 1982.

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

[4] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *CHARME'99*, Bad Herrenalb, Germany.

[5] M. Colón and T. Uribe. Generating Finite-state Abstractions of Reactive Systems using Decision Procedures. In *CAV'98*, volume 1427 of *LNCS*, July 1998.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Int. Conf. on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.

[7] C. Cornes, J. Courant, J. Filliatre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Munoz, C. Murthy, C. Parent, A. Saibi, and B. Werner. The Coq proof assistant reference manual, version 5.10. Technical report, INRIA, Rocquencourt, France, February 1995.

[8] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.

[9] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, volume 1633 of *LNCS*, 1999.

[10] C. Demartini, R. Iosif, and R. Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.

[11] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proc. of the 6th SPIN Workshop*, volume 1680 of *LNCS*, 1999.

[12] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[13] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer, Dordrecht, Netherlands, 1988.

[14] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *CAV'97*, volume 1254 of *LNCS*, 1997.

[15] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[16] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *Proc. of 5th and 6th SPIN Workshops*, volume 1680 of *LNCS*, September 1999.

[17] K. Havelund. Mechanical Verification of a Garbage Collector. In *FMPPTA'99*, number 1586 in LNCS, April 1999. San Juan, Puerto Rico.

[18] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proc. of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[19] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proc. of the 4th SPIN workshop, Paris, France*, November 1998. To appear in IEEE Transactions of Software Engineering.

[20] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, April 2000. Special issue for selected papers from SPIN'98 workshop.

[21] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME'96*, volume 1051 of *LNCS*, 1996.

[22] K. Havelund and J. Skakkebaek. Practical Application of Model Checking in Software Verification. In *Proc. of the 5th and 6th SPIN Workshops*, volume 1680 of *LNCS*, 1999.

[23] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, CWI, March 1994.

[24] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 1969.

[25] G. Holzmann. State Compression in Spin. In *Proc. of the 3rd Spin Workshop*, Enschede, Netherlands, April 1997.

[26] G. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[27] G. Holzmann and D. Peled. An Improvement in Formal Verification. In *FORTE'94*, Berne, Switzerland, October 1994.

[28] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, Dec 1998.

[29] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[30] R. Melton, D. Dill, C. N. Ip, and U. Stern. Murphi Annotated Reference Manual, Release 3.0. Technical report, Stanford University, Palo Alto, California, USA, July 1996.

[31] L. I. Millett and T. Teitelbaum. Slicing Promela and its Application to Model Checking, Simulation, and Protocol Understanding. In *Proc. of the 4th SPIN Workshop*, 1998.

[32] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[33] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *CAV'96*, number 1102 in LNCS, New Brunswick, NJ, July/August 1996.

[34] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proc. of the 22nd Int. Conf. on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.

[35] D. M. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[36] H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *14th IEEE Int. Conference on Automated Software Engineering*, October 1999.

[37] H. Saïdi and N. Shankar. Abstract and Model Check while you Prove. In *CAV'99*, volume 1633 of *LNCS*, July 1999.

[38] S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[39] M. Spivey. *The Z Notation: A Reference Manual, 2nd edition*. Prentice Hall Int. Series in Computer Science, 1992.

[40] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series, Prentice-Hall, 1992.

[41] F. Tip. A Survey of Program Slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[42] W. Visser, K. Havelund, and J. Penix. Adding Active Objects to SPIN. In *Proc. of the 5th SPIN Workshop*, Trento, Italy, July 1999.

[43] W. Visser, S. Park, and J. Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. In *Proc. of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.