Finding Feasible Counter-examples when Model Checking Abstracted Java Programs*

Corina S. Păsăreanu[1], Matthew B. Dwyer[1], and Willem Visser[2]

[1] Department of Computing and Information Sciences, Kansas State University, USA
[2] RIACS, NASA Ames Research Center, Moffett Field, USA
pcorina@cis.ksu.edu

Abstract. Despite recent advances in model checking and in adapting model checking techniques to software, the state explosion problem remains a major hurdle in applying model checking to software. It is well-accepted that automated techniques for abstracting programs will be necessary to overcome this problem. Most common abstraction techniques compute an upper approximation of the original program. Thus, when a specification is true for the abstracted program, it will also be true for the concrete program. However, if the specification is false for the abstracted program, the counter-example may be the result of some behavior in the abstracted program which is not present in the original program. We have extended an explicit-state model checker, Java PathFinder (JPF), to analyze counter-examples in the presence of abstractions, such as those introduced by the Bandera toolset. We enhanced JPF with an option to search for "feasible" (i.e. nondeterminism-free) counter-examples "on-the-fly", during model checking. Alternatively, an abstract counter-example can be used to guide the simulation of the concrete computation and thereby check feasibility of the counter-example. We demonstrate that these techniques can be effective in analyzing counter-examples from checks of several non-trivial multi-threaded Java programs.

1 Introduction

In the past decade, model checking has matured into an effective technique for reasoning about realistic components of hardware systems and communication protocols. The past several years have witnessed a series of efforts aimed at applying model checking techniques to reason about software design models (e.g., UML-statecharts [18]) and implementations (e.g., Java source code [8, 12, 24]). While the conceptual basis for applying model checking to software is reasonably well-understood, there are still unsettled questions about whether effective tool support can be constructed that allows for realistic software requirements to be

* This work was supported in part by NSF under grants CCR-9703094 and CCR-9708184, by NASA under grant NAG-02-1209, by Sun Microsystems under grant EDUD-7824-00130-US, and was performed for the Formal Verification of Integrated Modular Avionics Software Cooperative Agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

checked of realistic software descriptions in a practical amount of time. Most researchers in model checking believe that property-preserving abstraction of the state-space will be necessary to make checking of realistic systems practical (e.g., [6, 11, 19]). There are a variety of challenges in bringing this belief to reality. This paper addresses one of those challenges, namely, the problem of automating the analysis of counter-examples that have been produced from abstract model checks in order to determine whether they represent real system defects.

The work described in this paper involves the integration of two recently developed tools for model checking Java source code : Bandera [8] and Java PathFinder (JPF) [24]. Bandera is a toolset that provides automated support for reducing a program's state space through the application of program slicing and the compilation of abstract definitions of program data types. The resulting reduced Java program is then fed to JPF which performs an optimized explicit-state model check for program properties (e.g., assertion violations or deadlock). If the search is free of violations then the program properties are verified. If a violation is found the situation is less clear. Bandera uses abstractions that preserve the ability to prove *all paths* properties (e.g., such as assertions or linear temporal logic formulae). To achieve state space reduction, however, the ability to disprove such properties is sacrificed. This means that a check of an abstracted system may fail either because the program has an error or because the abstractions introduce *spurious* executions into the program that violate the property. The former are of interest to a user, while the latter are a potentially significant distraction to the user, especially if such spurious results occur in large numbers.

Several approaches have been proposed recently for analyzing the feasibility of counter-examples of abstracted transition-system models [5, 3, 4]. While our work shares much in common with these approaches, it is distinguished from all of them in three ways: (*i*) it treats the abstraction of both a program's control and data, as well as the run-time system scheduler and the property to be checked, (*ii*) the feasibility of a counter-example is judged against the semantics of a real programming language (i.e., Java), and (*iii*) we advocate multiple approaches for analyzing feasibility with different cost/precision profiles. We will demonstrate the practical utility of an implementation of our approaches by applying them to the analysis of counter-examples for several real multi-threaded Java applications.

Safe abstractions often result in program models where the information required to decide conditionals is lost and hence nondeterministic choice needs to be used to encode such conditionals (i.e., to account for both true and false results). Nondeterministic choice is also used to model the possible decisions that a thread (or process) scheduler would make. Such abstractions are safe for all paths properties since they are guaranteed to include all behaviors of the unabstracted system. The difficulty lies in the fact that they may introduce many behaviors that are not possible. To sharpen the precision of the abstract model (by eliminating some spurious behaviors) one minimizes the use of nondeterminism and it can be shown that the absence of nondeterminism equates to feasibility [23].

Section 3 describes how program control and data, the property and scheduler behavior are abstracted in Bandera/JPF using nondeterminism.

JPF can perform a state-space search that is bounded by nondeterministic-choice operations; a property violation that lies within this space has a counter-example that is free of nondeterminism and is hence feasible. JPF can also perform simulation of the concrete program guided by an abstract counter-example. If a corresponding concrete program trace exists then the counter-example is feasible. Section 4 describes these two techniques for analyzing program counter-examples that were added to JPF. Section 5 describes several defective Java applications whose counter-examples were analyzed using these techniques. In Section 6 we discuss related and future work and we conclude in Section 7. In the next section, we give some brief background on Bandera and JPF.

2 Background

Bandera [8] is an integrated collection of program analysis and transformation components that allows users to selectively analyze program properties and to tailor the analysis to that property so as to minimize analysis time. Bandera exploits existing model checkers, such as SPIN [15], SMV [20], and JPF [24], to provide state-of-the-art analysis engines for checking program-property correspondence. Bandera provides support for reducing a program's state-space via:

- *program slicing* automates the elimination of program components that are irrelevant for the property under analysis. Our Java slicer treats multi-threaded programs [14] and is based on calculation of the program dependence graph.
- *data abstraction* automates the reduction in size of the data domains over which program variables range [13]. A type inference algorithm is applied to ensure that a consistent set of abstractions are applied to program data. This type-based approach to abstraction is complementary to predicate abstraction approaches that reduce a program by preserving the ability to decide specific user-define predicates; JPF's companion tool implements predicate abstraction programs written in Java [25].

Java PathFinder (JPF) is a model checker for Java programs that can check any Java program, since it is built on top of a custom made Java Virtual Machine (JVM), for deadlock and violations of user-defined assertions [24]. Inside the JVM special attention is paid to reducing the size of a state (rather than the speed of execution which is typical for commercial JVM implementations), since this is one of the major efficiency concerns of explicit-state model checkers.

Users have the ability to set the level of atomicity during model checking: either one byte-code at a time, one line of code at a time (the default) or by specifying atomic blocks by calling `beginAtomic()` and `endAtomic()` methods from a special class called `Verify`. When JPF discovers a counter-example it indicates how to execute code from the initial state of the program to reach the error. Each step in the execution contains the name of the *class* the code is from, the *file* the source code is stored in, the *line number* of the source file that is currently being executed and the a number identifying the *thread* that is executing. Using only thread numbers in each step, JPF can be run in a simulation mode to reproduce the error.

3 Program Abstraction

Given a concrete program and a property, the strategy of verification by using abstraction can be summarized as follows: *(i)* define an abstraction mapping that is appropriate for the property being verified and use it to transform the concrete program into an abstract program, *(ii)* transform the property into an abstract property, *(iii)* verify that the abstract program satisfies the abstract property, and finally *(iv)* infer that the concrete program satisfies the concrete property. In this section, we summarize foundational issues that underlie these steps.

3.1 Data Abstraction

The abstract interpretation (AI) [9] framework as described in a large body of literature establishes a rigorous semantics-based methodology for constructing abstractions so that they are *safe* in the sense that they over-approximate the set of true executable behaviors of the system (i.e., each executable behavior is covered by an abstract execution). Thus, when these abstract behaviors are exhaustively compared to a specification and found to be in conformance, we can be sure that the true executable system behaviors conform to the specification.

We present an AI, in an informal manner, as a collection of three components: (1) a domain of abstract values, (2) an abstraction function mapping concrete program values to abstract values, and (3) a collection of abstract primitive operations (one for each concrete operation in the program). Substituting concrete operations applied to selected program variables with corresponding abstract operations of an AI yields an abstract program [6].

For example, we may wish to abstract from everything but the fact that integer variable `x` is zero or not. In this case, an appropriate abstraction for `x` might be the classic *signs* AI [1] which only keeps track of whether an integer value is negative, equal to zero, or positive. The abstract domain is the set of tokens $\{neg, zero, pos\}$. The abstraction function maps negative numbers to *neg*, 0 to *zero*, and positive numbers to *pos*. Abstract versions of each of the basic operations on integers are used that respect the abstract domain values. For example, an abstract version of the addition operation for *signs* is:

$+_{abs}$	$zero$	pos	neg
$zero$	$zero$	pos	neg
pos	pos	pos	$\{zero, pos, neg\}$
neg	neg	$\{zero, pos, neg\}$	neg

Abstract operations are allowed to return sets of values to model lack of knowledge about specific abstract values. This imprecision is interpreted in the model checker as a nondeterministic choice over the values in the set. Such cases are one source of "extra behaviors" that one gets as the abstract model over-approximates the set of true execution behaviors of the system.

Abstractions of program control can be induced by data abstractions. For example, a loop will collapse to a structure that can iterate zero or more times if its exit condition is completely abstracted. When abstracted control structures nest, further compression of the program's control is possible. Since control abstractions are driven by safe abstraction of data it follows that they are safe.

3.2 Property Abstraction

When abstracting properties, Bandera uses an approach similar to [16]. Informally, given an AI for a variable `x` (e.g. *signs*) that appears in a proposition (e.g.,`x>0`), we convert the proposition to a disjunction of propositions of the form `x==`a, where a are the abstract values that correspond to values that imply the truth of the original proposition (e.g., `x==`*pos* implies `x>0`, but `x==`*neg* and `x==`*zero* do not; it follows that proposition `x>0` is abstracted to `x==`*pos*). Thus, this disjunction under-approximate the truth of a concrete proposition insuring that the property holds on the original program if the abstracted property holds on the abstract program.

3.3 Scheduler Abstraction

Analyzing concurrent systems requires safe modeling of the possible scheduling decisions that are made in executing individual threads. Since software is often ported to operating system's with different scheduling policies, a property checked under a specific policy would be potentially invalid when that system is executed under a different policy.

To address this, the approach taken in existing model checkers is to implement what amounts to the most general scheduling policy (i.e., nondeterministic choice among the set of runnable threads). Properties verified under such a policy will also hold under any more restrictive policy. Fairness constraints are supported in most model checkers to provide the ability to more accurately model realistic scheduling policies (e.g., by eliminating starvation).

The Java language has a relatively weak specification for its thread scheduling policy. Threads are assigned priorities and a scheduler must ensure that "all threads with the top priority will eventually run" [2]. Thus, a model checker that guarantees progress to all runnable threads of the highest priority will produce only feasible schedules; JPF implements this policy.

3.4 Abstraction Implementation

In Bandera, generating an abstract program involves the following steps: the user selects a set of AIs for a program's data components, then type inference is used to calculate the abstractions for the remaining program data, then the Java

```java
public class Signs {
 public static final int NEG =0;
 public static final int ZERO=1;
 public static final int POS =2;
 public static int abs(int n){
  if (n < 0) return NEG;
  if (n == 0) return ZERO;
  if (n > 0) return POS;
 }
```

```java
public static int add(int a,int b){
 int r;
 Verify.beginAtomic();
 if(a==NEG  && b==NEG) r=NEG;
 else if(a==NEG  && b==ZERO)r=NEG;
 else if(a==ZERO && b==NEG) r=NEG;
 else if(a==ZERO && b==ZERO)r=ZERO;
 else if(a==ZERO && b==POS) r=POS;
 else if(a==POS  && b==ZERO)r=POS;
 else if(a==POS  && b==POS) r=POS;
 else r=Verify.choose(2);
 Verify.endAtomic(); return r; }}
```

Fig. 1. Java Representation of *signs* AI (excerpts)

class that implements each AI's abstraction function and abstract operations is retrieved from Bandera's abstraction library, and finally the concrete Java program is traversed, and concrete literals and operations are replaced with calls to classes that implement the corresponding abstract literals and operations.

Figure 1 shows excerpts of the Java representation of the *signs* AI. Abstract tokens are implemented as integer values, and the abstraction function and operations have straightforward implementations as Java methods. For Java base-types, the definitions of abstract operations are automatically generated using a theorem prover (see [13] for details). Nondeterministic choice is specified by calls to `Verify.choose(n)`, which JPF traps during model checking and returns nondeterministic values between `0` and `n` inclusive. Abstract operations execute atomically (via calls to `Verify.beginAtomic()` and `Verify.endAtomic()`) since they abstract concrete byte-codes (e.g. `Signs.add()` abstracts `iadd`).

4 Finding feasible counter-examples

We have seen in the previous section that, if a specification is true for the abstracted program, it will also be true for the concrete program. However, if the specification is false for the abstracted program, the counter-example may be the result of some behavior in the abstracted program which is not present in the original program. It takes deep insight to decide if an abstract counter-example is feasible (i.e. corresponds to a concrete computation). We have developed two techniques that automate tests for counter-example feasibility: model checking on *choose-free* paths and abstract counter-example guided concrete simulation.

4.1 Choose-free state space search

We enhanced the JPF model checker with an option to look only at paths that do not refer to instructions that introduce nondeterminism (i.e. a `Verify.choose()` call). When a *choose* occurs the search algorithm of the model checker backtracks. The approach exploits the following theorem from [23]: **Theorem.**

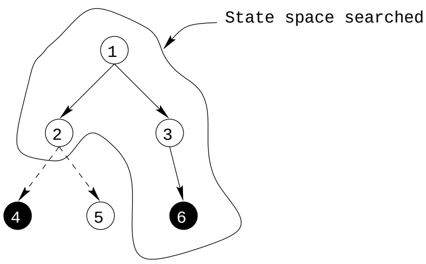

Fig. 2. Model Checking on Choose-free Paths

> *Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.*

The theorem ensures that paths that are free of nondeterminism correspond to paths in the concrete program (a more general definition of deterministic paths can be found in [10]). It follows that if a counter-example is reported in a *choose-free* search then it represents a feasible execution. If this execution also violates the property, then it represents a feasible counter-example.

Consider an abstracted program (whose state space is sketched in Figure 2). Black circles represent states where some assertion is violated. Dashed lines represent transitions that refer to *choose*, while solid lines refer to instructions other than *choose*. Model checking on choose-free paths will report only the error path 1-3-6, although path 1-2-4 leads to a state where the assertion is false (and it may correspond to an execution in the concrete program).

The abstractions we use correspond to *free* abstraction relations from [11]. When looking only at the choose-free paths, JPF examines (on-the-fly) paths that under-approximate the behavior of the concrete program. These paths correspond to the ones introduced by *constrained* abstraction relations [11]. Both free and constrained abstractions are used to build *mixed transition systems* for model checking full CTL. We use these abstractions in a different way : free abstract transitions for verifying properties and constrained abstraction transitions when looking for defects.

We also note that our technique could be implemented in any model checker, but the design of JPF made this modification particularly easy. JPF is essentially a special-purpose JVM that interprets each byte code in the compiled version of a Java program. Since *choose* operations are represented as static method calls, trapping and processing those operations specially only required modification of the code for the static method invocation byte-code. We made sure that the search on choose-free paths does not introduce deadlocks (choose instructions are interpreted as infinite self loops).

Consider checking the fragment of code on the left of Figure 3 against the assertion at line `4`, where initially `Global.done` is false; the abstracted code (using *signs* for `i`) is shown to the right of the original. In the abstracted program, nondeterminism is introduced through method `lt` that implements the abstract operation for `<`: after one pass through the `while` loop, the abstract value of `i` becomes *pos* and the value returned by `Signs.lt(i,Signs.POS)` can

```
    class App{
     public static void main(...){
[1]   new AThread().start(); ...
[2]   int i=0;
[3]   while(i<2){...
[4]     assert(!Global.done);
[5]     i++;
     }}}
    class AThread extends Thread{
     public void run(){ ...
[6]   Global.done=true;
     }}
```

```
class App{
 public static void main(...){
  new AThread().start(); ...
  int i=Signs.ZERO;
  while(Signs.lt(i,Signs.POS)){...
   assert(!Global.done);
   i=Signs.add(i,Signs.POS);
  }}}
class AThread extends Thread{
 public void run(){ ...
  Global.done=true;
 }}
```

Fig. 3. Simple Example of Concrete (left) and Abstracted (right) Code

be either true or false. However, the abstract program does expose a choose-free counter-example: if the thread that is an instance of `AThread` executes line 6 before the main thread begins the execution of the `while` loop, the assertion in line 4 is violated when the body of the loop is executed for the first time (and the abstract value of `i` is *zero*). This counter-example does not contain nondeterministic choices, since the value returned by `Signs.lt(i,Signs.POS)`, when `i` is *zero*, is uniquely true.

4.2 Abstract counter-example guided concrete simulation

In Bandera, the generation of an abstracted program is automatic and is done in such a way that there is a clear correspondence between the concrete and abstracted program: for each line in the concrete program, there is a single line in the abstracted program. Since byte-codes execute atomically, for each "concrete" byte-code, there is a set of "abstract" byte-codes that execute atomically in JPF. This property of Bandera abstraction, together with the fact that all Java variables have known initial values, allows for simulation of the concrete program, based on an abstract counter-example.

This is done in JPF by executing the steps in the abstract trace. For clarity, we'll discuss the simulation in terms of the execution of lines of Java source code, but JPF can also perform simulation on a byte-code level. Each step contains information about the thread to be run next and the line of the counter-example. At each step of the concrete execution, JPF checks that the concrete line to be executed corresponds to the abstract line in the counter-example. If the lines match throughout the simulation then the abstract trace is feasible, otherwise, the abstract trace is spurious. To check whether the feasible trace is a counter-example, we have also to check if it violates the property.

Consider again the example from Figure 3 where the result of model checking the abstracted program is a counter-example where `Global.done` is set true after the loop in the main thread is executed two times. This means that the assertion is reachable (and violated) by the (abstract) trace

`1-2-3-4-5-3-4-5-3-4`

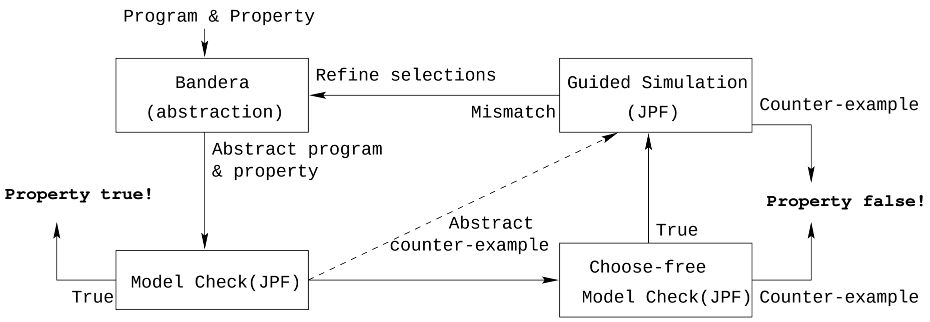

Fig. 4. Model Checking and Refinement

in the main thread. While this is clearly possible in the abstract program (since, after the abstract value of `i` becomes *pos*, the condition at line `3` can be non-deterministically true or false), it is not possible in the concrete program. To see this, we simulate the steps from the abstract trace on the concrete program: after executing the loop two times, the value of `i` is 2 so the exit condition of the loop is true and the loop is exited. At this point a line mismatch is detected and the simulation stops.

It is possible to detect the infeasibility of an abstract trace earlier, using a technique similar to forward analysis (e.g.[22]): when simulating each step on the concrete program, we also check the correspondence between concrete and abstract values. This can be done in JPF by abstracting the values of variables (e.g., via calls to `Signs.abs()`) in the concrete simulation and comparing them to the abstract values in the counter-example.

4.3 Methodology

Our methodology for model checking and abstraction involves the integration of the above two techniques as illustrated in Figure 4. The input (concrete) program and the specification are abstracted (using abstractions from Bandera's library) as described in Section 2 and the transformed program is fed to a model checker. If the result of model checking is true, then the specification is true for the concrete program. If the result is false, we re-run the model checker to search only choose-free paths in the model. If the model checker finds a choose-free counter-example, it is reported to the user otherwise we perform counter-example guided simulation. If the simulation succeeds, a counter-example is reported, but if a mismatch is detected then abstractions need to be refined. The refinement involves modifying the selection of abstractions guided by the counter-example reported in the first run of the model checker.

4.4 Discussion

In general, the result of model checking an abstract program is false either because the concrete program does not satisfy the property (in which case the

```
[1] x=1;                      |    x=Signs.POS;
[2] y=x+1;                    |    y=Signs.add(x,Signs.POS);
[3] assert(x<y);              |    assert((x==Signs.NEG && y==Signs.ZERO)
                              |          ||(x==Signs.NEG && y==Signs.POS)
                              |          ||(x==Signs.ZERO && y==Signs.POS));
```

Fig. 5. Example of Spurious Error Introduced by Property Abstraction

counter-example is feasible and indicates a real defect), or because the abstraction is not suitable for checking the property. In the latter case, the abstract counter-example can be one of the following:

– not feasible as a result of over-approximation of the behavior of the concrete program (e.g. the spurious counter-example of the program in Figure 3);
– feasible but not defective; as a result of the under-approximation of the property to be checked. This case is illustrated by the code in Figure 5, where both `x` and `y` are abstracted to *signs*. The predicate in the assertion is abstracted in such a way that if the assertion is true in the abstracted program, it follows that it is true in the concrete program. Abstract trace `1-2-3` violates the assertion, since after step 2, both `x` and `y` are *pos*. However, in the concrete program, the assertion is true.

In our experience this second case is rare, since in Bandera user's are guided to make abstraction selections that are able to decide both the truth and falsity of the propositions used in the property to be checked. Only when such a selection is impossible can a feasible, but not defective, counter-example arise.

We note that both choose-free model checking and abstract counter-example guided concrete simulation can be directly applied to a executable program slice. If a trace is feasible in the sliced program, it is also feasible in the original program [14].

5 Experience with Defective Java Applications

To illustrate the potential benefits of the techniques described in the previous section, we applied them to several small to medium-size multi-threaded Java applications. These applications used both lock synchronization and condition-based synchronization (i.e., `wait/notify`). The systems are: **RAX** (Remote Agent experiment) [25], a Java version of a component extracted from an embedded spacecraft-control application, **Pipeline** [7], a generic framework for implementing multi-threaded staged calculations, **RWVSN**, Lea's [17] generic readers-writers synchronization framework, and **DEOS** [21, 25], the scheduler from a real-time executive for avionics systems that was translated from C++. Table 6 gives some basic measures of the size of the system; *SLOC* stands for the number of source lines of code. Most of these programs use the basic features of Java and its concurrency constructs, however, the **RWVSN** application uses abstract classes, inheritance, and `java.util.Vector`.

Program	*SLOC*	*Classes*	*Methods*	*Fields*	*Threads*
RAX	55	4	8	7	3
Pipeline	103	5	10	7	5
RWVSN	590	5	43	10	5
DEOS	1443	20	91	92	6

Fig. 6. Java Program Size Measures

The **RAX** and **DEOS** examples had known errors that we checked for. For the **Pipeline** and **RWVSN** examples we seeded faults in the program. For example, we dropped a negation (`!`) in one program and changed `<=` into `<` (simulating an off-by-one error) in the other. It is interesting to note that not all seeded faults could be detected given the properties we checked for, so we altered the faults until we generated a property violation.

In the remainder of this section we describe several model checks and the automated analysis of the resulting counter-examples. Full details for the examples and model checks is available at `http://www.cis.ksu.edu/~pcorina/case-studies`.

5.1 Description of experiments

We model checked the **RAX** example to detect deadlocks using two different abstractions. Figure 7 shows excerpts from the original and the generated abstract Java program. The abstraction of the program was driven by our selection that the `Event.count` field should be abstracted with *signs*. Bandera's abstraction type inference determined that the local `count` variables in the `FirstTask.run()` method should also be abstracted. Running JPF on this abstracted system detects a deadlock and produces a 74 step counter-example. Analysis of this counter-example reveals that it is spurious. After 39 steps in the counter-example the trace reaches the conditional at line `15`. In the real system, the branch condition is false, but due to the nondeterminism of `Signs.eq()` for positive parameters the abstract system enters the conditional. JPF is able to find a 40 step choose-free counter-example. It is clear that the presence of spurious counter-examples is closely related to the property being checked, the program and the abstraction's selected. We reran our model checks changing the abstraction for `Event.count` field to record information about the evenness or oddness of its values. This produced a 128 step counter-example, but JPF was unable to find a choose-free counter-example. At this point, we ran JPF in simulation mode guided by the 128 step counter-example and while this counter-example did contain nondeterministic choices it was shown to be feasible.

The **Pipeline** example consists of an application that uses the methods of a `Pipeline` class to manage execution of a multi-threaded staged computation. The application constructs and starts execution of a pipeline, calls `stop()` to end execution of the pipeline, and calls `add()` to provide input to the computation. We model checked a precedence property for the **Pipeline** system stating that

```
[ 1]class Event{
[ 2] int count=0;
[ 3] public synchronized void wait_for_event(){
[ 4]  try{wait();}
[ 5]  catch(InterruptedException e){};
     }
[ 6] public synchronized void signal_event(){
[ 7]  count = count + 1;
[ 8]  notifyAll();
     }}
[ 9]class FirstTask extends Thread{
[10] Event event1,event2;
[11] int count=0;
[12] public void run(){
[13]  count = event1.count;
[14]  while(true){
[15]   if(count == event1.count)
[16]    event1.wait_for_event();
[17]   count = event1.count;
[18]   event2.signal_event();
      }}}
```

```
class Event {
 int count = Signs.ZERO;
 public synchronized void wait_for_event(){
  try {wait();}
  catch(InterruptedException e){};
 }
 public synchronized void signal_event(){
  count = Signs.add(count,Signs.POS);
  notifyAll();
 }}
class FirstTask extends Thread {
 Event event1,event2;
 int count = Signs.ZERO;
 public void run () {
  count = event1.count;
  while (true){
   if(Signs.eq(count,event1.count))
    event1.wait_for_event();
   count = event1.count;
   event2.signal_event();
  }}}
```

Fig. 7. RAX Program with Deadlock (excerpts)

"no pipeline stage (i.e., thread) will terminate until the stop method is called". Since JPF does not currently support checking of temporal properties, we encoded this using a boolean variable, `stopCalled`, set to true when the `stop()` method had been called and embedded `assert(stopCalled)` at the return point of the stage `run` methods. This example was abstracted by identifying a loop index variable that controlled the number of times the `add()` method was called and abstracting it to *signs*. Type inference determined that 5 additional fields and local variables also needed abstraction. Checking the property on the abstracted system detected an error on a 168 step counter-example. JPF found a 69 step choose-free counter-example that is similar to the example in Figure 3 in that it occurred on the first iteration of an abstracted loop.

RWVSN consists of an application that extends Lea's `RWVSN` class [17] to implement an object with a readers-writers synchronization policy. That object is then shared by several threads that read and write through the `RWVSN` interface. We checked that access by a reader excluded access by a writer by setting a boolean variable, `in_writer`, in the writer's critical section and resetting it upon exit, and embedding `assert(!in_writer)` in the reader's critical section. Abstraction was applied to 3 integer fields of the `RWVSN` class abstracting them to *signs*. Checking the property on the abstracted system detected an error in 179 steps. JPF found a 76 step choose-free counter-example.

The **DEOS** system has been the subject of several recent case studies in model checking code [21, 25, 13]; we performed the abstraction and analysis as described in [13]. The property being checked is an assertion that encodes a test for *time partitioning* in the scheduler component of the system. We used dependence analysis driven by the location of the assert statement and the data values it referenced to identify a single field (out of 92) as influencing the property. We selected the *signs* AI for that field and type inference determined that 2 more fields should be abstracted. Checking the property on the abstracted system detected an error in 471 steps. JPF found a 312 step choose-free counter-example.

5.2 Discussion

While these programs represent a range of different patterns of concurrency (e.g., clients and server, pipelines, and peer-groups) and the larger examples are real applications, we do not claim that our results generalize to a broader class of multi-threaded Java programs. We do, however, believe the results suggest that the counter-example analysis techniques we have developed have merit and can significantly reduce the burden users face when analyzing counter-examples from checks of abstracted systems.

The data clearly show that counter-examples can be reduced significantly in length; this alone makes it easier to diagnose the program fault. The fact that counter-examples are guaranteed to be feasible helps focus the user's attention on only those counter-examples for which analysis will lead to fault detection.

It should come as no surprise that a choose-free model check is faster than a typical model check since it is essentially a depth-bounded model check. Most model checkers can do depth-bounded search and in fact this often allows for detection of significantly shorter counter-examples. The key difference lies in the fact that a choose-free search uses an adaptive depth-bound that is based on encountering nondeterministic choice operators. This guarantee of not executing a choice operator is what assures counter-example feasibility. Without that a naive depth-bounded search may include execution of a choice operator.

Finally, we observe that choose-free search can be an effective way to exploit more aggressive abstraction approaches. The application of source-level predicate abstraction techniques to the **DEOS** and **RAX** is described in detail in [25]. In that work a predicate abstraction and an invariant for **DEOS** and 4 different predicate abstractions for **RAX** were used to produce abstract models that preserved both truth and falsity of the properties being checked. In contrast, the checks described in this paper sacrifice precision for more aggressive abstraction, and state-space reduction, while choose-free search enables the recovery of feasible counter-examples.

6 Related and Future Work

We have discussed foundational and program abstraction related work in the body of the paper. Here we link our approach to tool support for the analysis of abstract counter-examples from the verification community.

In SLAM [3], sequential C programs are abstracted into *boolean programs*; symbolic execution is used to map abstract counter-examples on concrete executions. Our simulation technique works because we analyze *closed* programs (i.e. programs that do not interact with their environment) and because Java defines default initial values for all data. This means that there is a single initial state for the program. A more general technique [5] is to allow for multiple initial states and to perform a symbolic simulation of the concrete system using predicates that characterize the program data values. This method was implemented in the SMV model checker. INVEST [4] and interactive abstractions [22] use theorem proving to rule out spurious counter-examples. Backward analysis is used to obtain information to refine the abstractions. Unlike our approach, these tools/techniques are not concerned with property abstraction or scheduling information.

We believe that the methods described in these papers are complementary to our techniques and may be combined with ours in order to achieve better results. For example, we can use backward analysis to obtain feedback for refinement of abstractions. Backward analysis computes pre-images of the violating abstract state over the given trace. For the spurious counter-example of Figure 3, after the body of the loop is executed two times, the value of the loop condition is true, which means that the concrete value of `x` is believed to be less than 2. The analysis would discover that this happens because the value of `x` before the assignment at line `5` is believed to be less than 1 (which is not true in the concrete program, where the value of `x` is exactly 1). It means that the new abstraction to be selected for variable `x` has to include a new token for 1 (e.g. *signs* abstraction should be replaced with *range(0..1)* abstraction [13]).

We note that both choose-free search and counter-example guided simulation techniques could be implemented in most model checkers. For example, Bandera [8] implements a form of adaptive depth-bounded search in Promela models generated for Spin that can be adjusted to perform choose-free search. Path simulation simply requires the ability to associate the steps of the concrete and abstract program and to simulate the concrete program. One can already do this by hand using Spin's simulation facilities, but automating the process would greatly ease its use.

We also note that, although we set our presentation in the context of Bandera's abstraction, other forms of data abstraction, like JPF's predicate abstraction, would also be treated properly. By that we mean that a path through the predicate abstracted code that is choose-free or that can be mapped to a concrete execution is feasible.

7 Conclusion

In this paper, we have suggested two approaches for analyzing counter-examples produced by model checks of abstracted programs. These approaches have the advantage of being very fast (i.e., choose-free search is depth-bounded and the cost of simulation is related to the length of the counter-example). Based on experimentation with an implementation of these techniques in a Java model checking tool we have also found the techniques to be capable of detecting guaranteed feasible counter-examples in nearly every case. This enables aggressive abstractions to be applied without losing the ability to detect errors, thereby minimizing the need for refinement of abstractions. This implementation treats not only abstraction of program data, but also of thread scheduling policies, and the property to be checked. Finally, we believe that these techniques can be combined with other counter-example analysis methods to provide a suite of tools that vary cost and in their ability to precisely analyze counter-examples. Such a tool suite would be a useful addition to any model checking tool.

References

1. S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.
2. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 2nd edition, 1998.
3. T. Ball and S.K. Rajamani. Checking temporal properties of software with boolean programs. In *Proc. Workshop on Advances in Verification*, July 2000.
4. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10th International Conference on Computer-Aided Verification*, June 1998.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th International Conference on Computer-Aided Verification*, July 2000.
6. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
7. J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Rob by. Bandera : A source-level interface for model checking Java prog rams. In *Proc. 22nd International Conference on Software E ngineering*, June 2000.
8. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Păsăreanu, Robby, and H Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering*, June 2000.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
10. D. Dams, R. Gerth, G. Dhmen, R. Herrmann, P. Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In D.L.Dill, editor, *Proc. 6th International Conference on Computer-Aided Verification*, volume 818 of LNCS, pages 455–467. Springer Verlag, June 1994.
11. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.
12. 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.
13. M.B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C.S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. Technical report, Department of CIS, Kansas State University, August 2000.
14. J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proc. 6th International Static Analysis Symposium*, September 1999.

15. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
16. Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. LNCS, 1450, 1998.
17. D. Lea. *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns*. The Java Series. Addison-Wesley, 2nd edition, 1999.
18. J. Lilius and I.P. Paltor. vUML: A tool for verifying UML models. In *Proc. 14th IEEE International Conference on Automated Software Engineering*, 1999.
19. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajiani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
20. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
21. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS real-time scheduling kernel. In *Proc. 22nd International Conference on Software Engineering*, June 2000.
22. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, March 1999.
23. H. Saidi. Model checking guided abstraction and analysis. In *Proc. 7th International Static Analysis Symposium* , 2000.
24. W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, September 2000.
25. W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *Proc. 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.