Int J STTT (2002) 4: 1–15 / Digital Object Identifier (DOI) 10.1007/s10009-002-0088-z

Finding feasible abstract counter-examples

Corina S. Păsăreanu[1], **Matthew B. Dwyer**[2], **Willem Visser**[3]

[1] Kestrel, NASA Ames Research Center, Moffett Field, Calif., USA
[2] Department of Computing and Information Sciences, Kansas State University, Kan., USA
[3] RIACS, NASA Ames Research Center, Moffett Field, Calif., USA; E-mail: pcorina@email.arc.nasa.gov

Published online: █ 2002 – © Springer-Verlag 2002

Abstract. A strength of model checking is its ability to automate the detection of subtle system errors and produce traces that exhibit those errors. Given the high-computational cost of model checking most researchers advocate the use of aggressive property-preserving abstractions. Unfortunately, the more aggressively a system is abstracted the more infeasible behavior it will have. Thus, while abstraction enables efficient model checking it also threatens the usefulness of model checking as a defect detection tool, since it may be difficult to determine whether a counter-example is feasible and hence worth developer time to analyze.

We have explored several strategies for addressing this problem by extending an explicit-state model checker, Java PathFinder (JPF), to search for and analyze counter-examples in the presence of abstractions. We demonstrate that these techniques effectively preserve the defect detection ability of model checking in the presence of aggressive abstraction by applying them to check properties of several abstracted multi-threaded Java programs. These new capabilities are not specific to JPF and can be easily adapted to other model checking frameworks; we describe how this was done for the Bandera toolset.

Keywords: █ – CE^a

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 implementations (e.g., Java source code [11, 15, 33]). 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 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., [8, 14, 26]). 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 [11] and Java PathFinder [33]. 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

This paper is an expanded version of [29]. This work was supported in part by NSF under grants CCR-9703094 and CCR-9708184, by NASA under grant NAG-02-1209, by DARPA/ITO's PCES program through AFRL Contract F33615-00-C-3044, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564, and by the Formal Verification of Integrated Modular Avionics Software Cooperative Agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

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 distraction to the user, especially if spurious results occur in large numbers.

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 in the implementation of abstract operations, to model the lack of knowledge about specific abstract values. 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 [31].

Several approaches have been proposed recently for analyzing the feasibility of counter-examples of abstracted transition-system models [3, 7, 23]. While our work shares much in common with these approaches, it is distinguished from them in four ways: (i) it treats the abstraction of both the program's data and the property to be checked; (ii) the feasibility of a counter-example is judged against the semantics of a real programming language; (iii) we advocate multiple approaches for analyzing feasibility with different cost/precision profiles; and (iv) our work is oriented toward detecting defects in the presence of abstraction. Our work makes a contribution by adapting counter-example analysis techniques developed in simplified settings to support the analysis of systems written in modern programming languages. Concretely we have enhanced JPF with two new capabilities. JPF can now 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 deterministic 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. An important contribution of our work is that we provide convincing evidence that these techniques are effective in detecting feasible counter-examples under aggressive abstraction. We report the results of analyzing counter-examples produced from model checks of properties of seven non-trivial multi-threaded Java programs.

While our presentation focuses on tools for abstracting and model checking of Java programs, it is important to note that the approach developed here is not specific to Java or to JPF. To illustrate this, we describe how we enhanced the Bandera toolset to implement the choice-bounded search using the Spin [21] model checker.

Section 3 presents our approach for abstracting Java programs using nondeterminism. Section 4 describes the two techniques for analyzing program counter-examples that were added to JPF and their adaptation to Bandera. Section 5 describes several defective Java applications whose counter-examples were analyzed using these techniques. In Sect. 6 we discuss related work and we conclude in Sect. 7. In the next section, we give some brief background on Bandera and JPF.

2 Background

2.1 The Bandera tool-set

Bandera [11] 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 [21] and JPF [33], to provide state-of-the-art analysis engines for checking program-property correspondence.

To bridge the gap from Java to model checker input language Bandera is organized much like an optimizing compiler. Java programs are translated to intermediate representations that are amenable to different kinds of program analyses and transformations. A typical series of analyses and transformations is as follows:

1. *Compilation* of Java is performed using traditional parsing and semantic analysis techniques. The result of this process is a 3-address representation of the program at the level of granularity of Java Virtual Machine (JVM) byte-codes [25]; the specific representation used is Jimple provided by the SOOT compilation framework [32]. The property to be checked is translated from the Bandera Specification Language [9] (BSL) into a form that refers to Jimple variables and locations.
2. *Program slicing* automates the elimination of program components that are irrelevant for the property under analysis. Slicing criteria are automatically extracted from the observable predicates that are referenced in the property. Our Java slicer treats multi-threaded programs [20] and is based on calculation of the program dependence graph.
3. *Data abstraction* automates the reduction in size of the data domains over which program data range [18] by replacing implementation types (e.g., an integer) with abstract types (e.g., that records the sign of the integer value). A user identifies fields of Java classes that appear relevant to the property being checked (e.g., those that are referenced in the BSL specification) and selects abstract types for those fields. A type

inference algorithm is applied to calculate a consistent set of abstract types for the rest of a program's data. This type-based approach to abstraction is a specialized form of predicate abstraction (e.g., [4,5,34]). It is restricted to non-relational predicates, but has the advantage of allowing operations on abstract types to be calculated once and reused in abstracting different programs.

4. *Jimple simplifications* are applied to perform traditional compiler optimizations that may reduce the state space (e.g., calculation of variable liveness information, local name packing, and method inlining).
5. *Transition system generation* converts the resulting sliced, abstracted, and simplified Jimple representation to a model checker independent transition system notation called BIR (Bandera Intermediate Representation). BIR is essentially a guarded-assignment language that includes support for object-oriented features, such as, the definition of collections of structures for representing heap-allocated data and the definition of locks and wait sets that can be associated with structure instances. BIR also provides support for determining the visibility of each transition relative to the specification being checked and collapsing consecutive invisible transitions into atomic steps.
6. *Model checker input generation* defines representations for BIR types and implementations of BIR guards and assignments. This translation is greatly simplified due to the fact that the majority of BIR's constructs map almost directly to constructs found in model checker input languages, such as Promela (the input language of Spin). The object-oriented features of BIR (e.g., support for collections, locks, and querying the inheritance hierarchy) require more care in translating. The generated model is checked by invoking the appropriate tool.
7. *Counter-example display* is invoked when a violation of the property is detected. The low-level counter-example is mapped back to a sequence of BIR transitions. A simulator for BIR allows for moving forward and backward through the counter-example and querying values of BIR state components, which are mapped back to Java variables and instances for display to the user.

We discuss data abstraction in more detail in Sect. 3 and the adaptation of the final three steps of this process to treat abstract counter-examples in Sect. 4.

2.2 Java PathFinder

Java PathFinder [33] is a model checker for Java programs that can check any Java program, since it is built on top of a custom made JVM. JPF's companion tool [34] implements predicate abstraction for programs written in Java.

In JPF special attention is paid to reducing the memory usage, rather than execution speed as is typical of commercial JVMs, since this is the major efficiency concern in explicit-state model checking. Users have the ability to set the granularity of atomic steps during model checking to: byte-codes, source lines (the default) or explicit atomic blocks (through calls to `beginAtomic()` and `endAtomic()` methods from a special class called `Verify`). To model nondeterministic behavior, a special method is called that the model checker will trap during execution. A JPF counter-example 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, a number identifying the *thread* that is executing and if the executed code involved a *nondeterministic choice* it also includes which choice was made. Using only thread numbers and nondeterministic choice numbers in each step JPF can simulate the erroneous execution.

Recent enhancements to JPF include both the addition of new capabilities as well as improvements in implementation (and therefore performance). JPF now supports temporal logic property checking in Linear Time Temporal logic (LTL) in addition to the default mode where it checks for deadlocks and user-defined assertions. Furthermore, new heuristic search capabilities have been added (for deadlock and assertion checking only) that allows breadth-first search (BFS) where all the successor states of the current state are put into a priority queue depending on a user-specified heuristic function (default heuristic is BFS). For example, when searching for a deadlock in a Java program one might specify a heuristic that will favor exploration of states in which more threads are blocked, since the more threads are blocked the closer one *should* be to a state where all threads are blocked (i.e., a deadlock).

JPF is integrat ed with Bandera after step 4 of Sect. 2.1. The Jimple representation of the program can either be converted directly to byte-codes or decompiled to Java which is then compiled to byte-codes.

3 Program abstraction

Given a concrete program and a property, the strategy of verification by using abstraction involves: (i) defining an abstraction mapping that is appropriate for the property being verified and using it to transform the concrete program into an abstract program; (ii) transforming the property into an abstract property; (iii) verifying that the abstract program satisfies the abstract property; and finally (iv) inferring 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) [12] 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 domain of abstract values, an abstraction function mapping concrete program values to abstract values, and 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 [8].

For example, to abstract from everything but the fact that integer variable `x` is zero or not one could use the *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 a source of "extra behaviors" introduced in the abstract model due to its over-approximation of the set of behaviors of the original system.

3.2 Property abstraction

When abstracting properties, Bandera uses an approach similar to [22]. 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-approximates 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 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 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.

```
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;
  }
  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(7);
    Verify.endAtomic();
    return r;
  }
}
```

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

For Java base-types, the definitions of abstract operations are automatically generated using a theorem prover (see [18] for details). Nondeterministic choice is specified by calls to `Verify.choose(bits)`, which JPF traps during model checking and returns nondeterministic values between the abstract values encoded in the bit-vector `bits`. Specifically, `Verify.choose(7)` denotes a nondeterministic choice between the values `NEG`, `ZERO` and `POS`. 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 have enhanced a model checker with an option to perform an adaptive depth-bounded search that backtracks whenever an instruction that introduces nondeterminism is encountered (i.e., a `Verify.choose()` call). The approach exploits the result from [31, Theorem 5], which states that *every path in the abstracted program where all assignments are deterministic has a corresponding path in the concrete (unabstracted) program.*

The result is a corollary of the fact that if an abstract system is *deterministic*, then it is equivalent to the concrete system (i.e., there is a simulation equivalence between the concrete and abstract systems). The result in [31] is stated for predicate abstraction applied to programs described as guarded-assignment transition systems where only the assignments may be nondeterministic as a result of abstraction. The theorem applies to our approach since: (i) predicate abstraction subsumes type-based abstraction, where the abstract domain is finite [31]; and (ii) Java programs can be modeled by a guarded-assignment transition system enriched with types for modeling object references, locks, and wait-sets [25] (this is what Bandera's BIR model does).[1] As discussed in Sect. 3, our abstraction approach introduces nondeterminism in expression evaluation; consequently both assignments and conditionals may be nondeterministic. Compiling Java to a byte-code representation effectively "normalizes" a program by adding Boolean temporary variables for all the expressions in conditionals, thus insuring that in the abstracted program, only the assignments may be nondeterministic as a result of abstraction.

Intuitively, the evaluation of an abstract operation is deterministic (i.e., its implementation does not refer to *choose* instructions), if its outcome is a unique abstract value. A transition (or assignment) is deterministic if all evaluated operations are deterministic, while a path is deterministic if it involves only deterministic transitions. For every deterministic transition in the abstract program, there is a corresponding transition in the concrete program; hence, every deterministic abstract path has a corresponding path in the concrete program. For example, consider assignment `x=x+1;` in some concrete program, where the initial value of variable `x` is 0, and assume that we decide to abstract `x` with *signs.* The abstracted assignment is deterministic, since $+_{abs}$ applied to *zero* and *pos* is deterministic, and its unique outcome is *pos*; for the corresponding abstracted transition that changes the state of `x` from *zero* to *pos*, there is a concrete transition, that changes the value of `x` from 0 to 1. Assume now that the (concrete) initial value of `x` is -1. Then, the corresponding abstracted assignment is nondeterministic, and its outcome may be *zero*, *pos* or *neg*. For a corresponding abstract transition that changes the value of `x` from *neg* to *pos* or from *neg* to *neg*, there is no concrete transition.

Saïdi uses [31, Theorem 5] to judge a counter-example feasible and drive abstraction refinement, whereas we use it to bias the model checker to search for feasible counter-examples. By construction, the sub-space explored by this search constitutes a deterministic abstract model of a portion of the concrete program's behavior. 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 [13]; we should also note that determinism corresponds to *completeness* in abstract interpretation (see e.g., [19]), which states that no loss of precision is introduced by the abstraction. 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 Fig. 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).

[1] The statement of [31, Theorem 5] requires also that no loss of precision is introduced by abstracting the properties. We are interested here only in determining path feasibility, and for this purpose the property under consideration is irrelevant (see also Sect. 4.4).

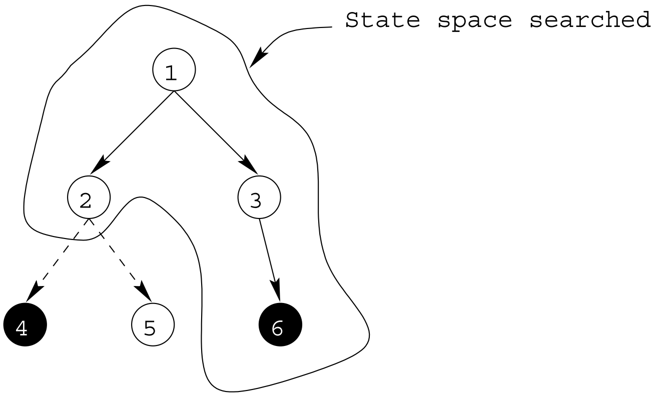

Fig. 2. Model checking on choose-free paths

To illustrate this more concretely, consider checking the fragment of code on the left of Fig. 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 `Signs.lt()` that implements the abstract operation for integer `<`. 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 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.

4.1.1 JPF Implementation

Our technique could be implemented in any model checker, but the design of JPF made these 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. Specifically, whenever the next instruction to be executed in a thread is a *choose* method call, this thread is considered not to be enabled. We made sure that the search on choose-free paths does not introduce deadlocks (choose instructions are interpreted as infinite self-loops).

We also implemented a choose-free heuristic search within JPF. This heuristic favors exploration from states that have the least number of nondeterministic choice statements currently enabled. This has the effect of trying to explore the choose-free state-space before exploring parts of the state-space that requires a *choose* statement (i.e., nondeterministic choice) to be executed. Note that unlike in the previous choose-free search described, here the exploration does not stop when no choose-free path is found, but continues to explore the whole abstract state-space. It is however easy to detect when a counter-example produced during a heuristic search is choose-free or not by just checking whether any choose statements were executed on the path. Consider again the abstracted program whose state space is sketched in Fig. 2. During choose-free heuristic search, JPF first explores the state space that is bounded by the transitions that introduce nondeterminism (so it will report the same error path 1-3-6). Once all choose-free path prefixes are considered without finding a counter-example, the algorithm proceeds to explore the rest of the state space. This

```
     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(...) {
[1]      new AThread().start();
         ...
[2]      int i=Signs.ZERO;
[3]      while(Signs.lt(i,Signs.POS)) {
           ...
[4]        assert(!Global.done);
[5]        i=Signs.add(i,Signs.POS);
         }
       }
     }
     class AThread extends Thread {
       public void run() {
         ...
[6]      Global.done=true;
       }
     }
```

Fig. 3. Simple example of concrete (left) and abstracted (right) code

means that, in Fig. 2, if the only erroneous state would have been state 4, JPF would have discovered it (and it would have reported error path 1-2-4). The only drawback to this heuristic search is that storing the states that comprise the frontier of the choose-free subspace can be expensive.

4.1.2 Bandera implementation

Bandera represents *choose* operations as explicit BIR operators. The implementation of a BIR `ChooseExpr` depends on the model checker being targeted. For example, an assignment to a local variable, `x`, of Boolean typed `ChooseExpr` would be implemented in Promela as:

```
if
:: true -> x = 0;
:: true -> x = 1;
fi;
```

When a choose-free search is desired the implementation of `ChooseExpr`s is modified to force the thread executing the *choose* into a self-loop. This causes the model checker to explore extensions of the trace in other threads. For example, the assignment above would be implemented in Promela as:

```
loc: goto loc;
```

This basic strategy can be used in any of the model checkers that we have studied.

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. 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 by executing the steps in the abstract trace. For clarity, we discuss the simulation in terms of the execution of lines of Java source code, but simulation can also be performed at a finer level (e.g., byte-code). 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, we check 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 Fig. 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]
```

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 nondeterministically 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., [7]). During simulation at each step on the concrete program, we check the correspondence between concrete and abstract values. This can be done 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.

Our simulation technique works because we analyze programs that do not exchange data with their environment and Java defines default initial values for all data (thus a program has a single initial state). More general simulation techniques, that handle multiple initial states, are discussed in Sect. 6.

4.2.1 JPF Implementation

In simulation mode JPF can execute a pre-determined path by looking at which thread is executing and the nondeterministic choice taken by the thread (if one exists). Steps in the path contain additional information, such as the class name and line number of the executing thread, that can be used to determine the execution context. When executing the concrete system using the error-path generated from analysis of the abstract system we check whether the class name and line number expected by the error path is matched by the execution in the system. A mismatch indicates that the abstract path is not feasible in the concrete system. Since JPF states store variable values explicitly it is easy to extend the checking of abstract/concrete state correspondence to include data; this requires the abstraction functions (e.g., `Signs.abs()` from Fig. 1).

4.2.2 Bandera implementation

As mentioned in Sect. 2 Bandera includes a simulator for systems represented in BIR. Assuming location correspondence between the original and abstracted program, as above, Bandera's transition system generation phase will produce BIR for the original and abstracted program whose locations correspond. The original BIR simulator assumed that a sequence of transitions was available to drive the simulation, consequently there was no provision for determining the enabled transitions in a state and selecting from them. To determine feasibility of the abstract BIR trace, we modified the simulator so that at each state in the abstract trace we can:

– determine which concrete BIR transitions are enabled;
– detect whether the next abstract BIR transition corresponds to one of those transitions;
– if not, then the abstract counter-example is infeasible;
– otherwise, the concrete simulation is extended by the corresponding transition.

BIR states store variable values explicitly so, as with JPF, extending checking of abstract/concrete state correspondence to include data is straightforward.

4.3 Methodology

Our methodology for model checking and abstraction involves the integration of the above two techniques as illustrated in Fig. 4. The input (concrete) program and the specification are abstracted (using abstractions from Bandera's library) as described in Sect. 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 rerun 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. For a discussion of abstraction refinement, see Sect. 6.

We note that using JPF's choose-free heuristic search has the advantage that we do not have to run the model checker twice to get a result. This changes the methodology above in the following way. The abstracted program is fed to a model checker, to perform choose-free heuristic search. If the result is true, then the specification is true for the original program. If the result is false and the counter-example is choose-free, then it is reported to the user; otherwise, we perform the counter-example guided simulation.

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 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 due to over-approximation of the behavior of the concrete program (e.g., the spurious counter-example of the program in Fig. 3).

feasible but not defective due to under-approximation of the property to be checked.

The latter case is illustrated by the code in Fig. 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

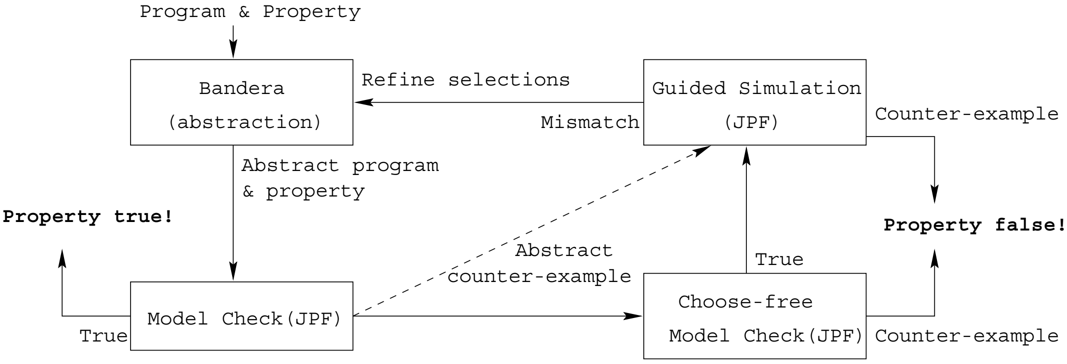

Fig. 4. Model checking and refinement

```
[1] x=1;
[2] y=x+1;
[3] assert(x<y);
```

```
[1] x=Signs.POS;
[2] y=Signs.add(x,Signs.POS);
[3] 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

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 users 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 is it necessary to check whether the feasible counter-example is defective or not.

We note that both choose-free model checking and abstract counter-example guided concrete simulation can be directly applied to an executable program slice. If a trace is feasible in the sliced program, it is also feasible in the original program [20]. We also note that the techniques presented here can be applied for checking safety properties expressed in any universal temporal logic.

4.5 Abstraction and nondeterminism

When building safe abstractions, we use *explicit* nondeterminism (i.e., special instructions to be interpreted by verification tools as nondeterministic choice). We also use explicit nondeterminism when modeling the environment in which a program executes. For example, during our analysis of the DEOS kernel presented in Sect. 5, we used nondeterminism to model the behavior of the kernel's environment, which consists of the user applications running on the kernel and the hardware. We distinguish (syntactically) (i.e., we use two different special instructions) between *internal* nondeterminism, introduced by data abstractions to model lack of knowledge about abstracted variable values, and *external* nondeterminism, introduced because of the lack of knowledge about the environment in which a program executes.

Implicit nondeterminism is used to model the possible decisions that a thread scheduler would make. 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.

The choose-free search technique is set to be bounded only by the internal nondeterminism introduced by data abstraction. This means that during choose-free search, a model checker analyzes a program's behavior that may be nondeterministic because of the thread scheduler or the environment (but not because of data abstraction).

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) [34], a Java version of a component extracted from an embedded spacecraft-control application, **Pipeline** [10], a generic framework for implementing multi-threaded staged calculations, **RWVSN**, Lea's [24] generic readers-writers synchronization framework, **DEOS** [28, 34], the scheduler from a real-time executive for avionics systems that was translated from C++, **BoundedBuffer** [27], a bounded buffer implementation in Java that is amenable to simultaneous use by multiple threads, **NestedMonitor** [27], a version of the bounded buffer implementation that uses semaphores instead of Java condition-based synchronization, and **ReplicatedWorkers** [17], a parameterizable parallel job scheduler.

The following table gives some basic measures of the size of the system; *SLOC* stands for the number of source lines of code.

Program	*SLOC*	*Classes*	*Threads*
RAX	55	4	3
Pipeline	103	5	5
RWVSN	590	5	5
DEOS	1443	20	6
BoundedBuffer	127	5	5
NestedMonitor	214	6	3
ReplicatedWorkers	954	11	5

Most of these programs use a rich set of Java features and concurrency constructs, including abstract classes, inheritance, and `java.util.Vector`.

The **RAX**, **DEOS**, **BoundedBuffer**, and **NestedMonitor** examples had known errors that we checked for. For the **Pipeline**, **RWVSN** and **ReplicatedWorkers** examples we seeded faults in the program. For example, we dropped a negation (`!`) in two programs 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 our experiments, we encoded the properties to be checked as assertions (rather than temporal logic formulae), in order to use JPF's new heuristic-search capabilities.

5.1 Description of experiments

We now describe several model checks for the abstracted systems and the automated analysis of the resulting counter-examples. Figure 6 gives the data for each of the model checking runs, using JPF to perform choose-free search, depth-first search, breadth-first search, and choose-free heuristic search. For each run, we report the abstraction that was used, the size of the counter-example, the total of user and system time to execute the checking and the memory used in verification. All model checks were performed on a SUN ULTRA5 with a 270 MHz UltraSparc IIi and 512 MB of RAM. The results are slightly different from [29], due to the use of the updated (and improved) version of the JPF tool. Full details for the examples and model checks are available at [16].

5.1.1 **RAX**

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, since there are two event objects allocated; this amounts to four abstracted variables in the system.

Running JPF (using depth-first search) on this abstracted system detects a deadlock and produces a 103-step counter-example. Analysis of this counter-example reveals that it is spurious. After 35 steps in the counter-

Program		*Choose-free search*	*Depth-first search*	*Breadth-first search*	*Choose-free heuristic search*
RAX(*signs*)	Size:	30	103	30	30
	Memory:	2.5M	2.3M	2.5M	3M
	Time:	12.3 s	12.4 s	11.8 s	11.4 s
RAX(*even-odd*)	Size:	None found	72	30	30
	Memory:		2.8M	4.4M	8.6M
	Time:		12.7 s	13 s	16.9 s
Pipeline(*signs*)	Size:	55	55	22	22
	Memory:	2.2M	67.1M	1.6M	1.6M
	Time:	11 s	3:22.3 s	10.8 s	10.9 s
RWVSN(*signs*)	Size:	70	27056	64	64
	Memory:	11.7M	182.9M	91M	94.8M
	Time:	49.2 s	11:29.6 s	4:17.4 s	4:18.4 s
DEOS(*signs*)	Size:	192	294	Out of memory	Out of memory
	Memory:	317.8M	228.2M		
	Time:	19:38.5 s	11:59.5 s		
BoundedBuffer(*signs*)	Size:	None found	5303	56	56
	Memory:		43.3M	159.7M	128.9M
	Time:		1:29.5 s	7:29.7 s	5:54.6 s
BoundedBuffer(*range(0..1)*)	Size:	353	5918	56	56
	Memory:	19.4M	57.6M	53.5M	30.6M
	Time:	2:46 s	3:51 s	5:12.7 s	1:23. s
NestedMonitor(*signs*)	Size:	22	111	22	22
	Memory:	1.9M	1.5M	711.1k	1.6M
	Time:	23.3 s	6.7 s	5.5 s	3.1 s
ReplicatedWorkers(*signs*)	Size:	423	423	Out of memory	Out of memory
	Memory:	3.6M	3.6M		
	Time:	27.1 s	27.8 s		

Fig. 6. Data for experiments CE[b]

CE[b] By 'M' and 'k' in this figure do you mean 'Mb' and 'kb', respectively ? Please confirm if this is the case.

Editor's or typesetter's annotations (will be removed before the final TEX run)

```
[ 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();
         }
       }
     }
[19] class SecondTask extends Thread {
[20]   Event event1,event2;
[21]   int count=0;
[22]   public void run() {
[23]     count = event2.count;
[24]     while (true) {
[25]       event1.signal_event();
[26]       if (count == event2.count)
[27]         event2.wait_for_event();
[28]       count = event2.count;
         }
       }
     }
```

```
[ 1] class Event {
[ 2]   int count = Signs.ZERO;
[ 3]   public synchronized void wait_for_event() {
[ 4]     try {wait();}
[ 5]     catch(InterruptedException e){};
     }
[ 6]   public synchronized void signal_event() {
[ 7]     count = Signs.add(count,Signs.POS);
[ 8]     notifyAll();
       }
     }
[ 9] class FirstTask extends Thread {
[10]   Event event1,event2;
[11]   int count = Signs.ZERO;
[12]   public void run () {
[13]     count = event1.count;
[14]     while (true){
[15]       if (Signs.eq(count,event1.count))
[16]         event1.wait_for_event();
[17]       count = event1.count;
[18]       event2.signal_event();
         }
       }
     }
[19] class SecondTask extends Thread {
[20]   Event event1,event2;
[21]   int count=Signs.ZERO;
[22]   public void run() {
[23]     count = event2.count;
[24]     while (true) {
[25]       event1.signal_event();
[26]       if (Signs.eq(count,event2.count))
[27]          event2.wait_for_event();
[28]       count = event2.count;
         }
       }
     }
```

Fig. 7. RAX Program with deadlock (excerpts)

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 30-step choose-free counter-example. Running JPF using the heuristic searches (i.e., both breadth-first search and choose-free heuristic search) discover the same counter-example (also 30-steps long). We ran JPF in simulation mode guided by this 30-step counter-example and it was shown to be feasible.

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 72-step counter-example (using depth-first search) and a 30-step counter-example (using both heuristic searches), but JPF was unable to find a choose-free counter-example. At this point, we ran JPF in simulation mode guided by the 72-step and the 30-step counter-examples and while these counter-examples did contain nondeterministic choices, they were shown to be feasible.

5.1.2 Pipeline

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 "no pipeline stage (i.e., thread) will terminate until the stop method is called". 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 five additional fields and local variables also needed abstraction. JPF found a choose-free counter-example that is similar to the example in Fig. 3 in that it occurred on the first iteration of an abstracted loop. We ran JPF in simulation mode to analyze the counter-examples produced during depth-first search and the heuristic searches, and they were shown to be feasible.

5.1.3 RWVSN

RWVSN consists of an application that extends Lea's `RWVSN` class [24] 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 three integer fields of the `RWVSN` class abstracting them to *signs*. JPF found a choose-free counter-example; the counter-example produced during depth-first search was analyzed and found not feasible, while the counter-examples produced during heuristic searches were shown to be feasible.

5.1.4 DEOS

The **DEOS** real-time scheduling kernel has been the subject of several recent case studies in model checking code [18, 28, 34]; we performed the abstraction and analysis as described in [18]. The property being checked is an assertion that encodes a test for *time partitioning* in the scheduler component of the system. We used JPF to detect a subtle implementation error related to the kernel's time partitioning requirement that was originally discovered and fixed during the standard formal review process. That requirement is that "application processes are guaranteed to be scheduled for their budgeted time during a scheduling unit". The requirement was encoded as a method that observes the state of the kernel and asserts that budgets are allocated in each scheduling unit. Calls to this method are inserted whenever the kernel schedules an application process; this guarantees the detection of property violations. To analyze the DEOS kernel, additional code was written to simulate the behavior of user applications and the hardware environment (e.g., a tick generator thread simulates a hardware clock for time related processing in the kernel). We modeled the environment using explicit external nondeterminism.

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 two more fields should be abstracted. Checking the property on the abstracted system detected a choose-free counter-example (i.e., a counter-example that does not contain internal nondeterminism introduced by the abstraction). The counter-example found using depth-first search turned out to be feasible when simulated. JPF ran out of memory during the heuristic searches.

5.1.5 BoundedBuffer

The **BoundedBuffer** program uses a synchronization object that monitors the access to a bounded buffer into which producer threads put items and consumer threads get items. The monitor class maintains an array of objects, two indices into that array representing the beginning and the end of the active segment of the array and a counter of the items stored into the buffer. A constant `SIZE` defines the maximum number of items that may be stored in the buffer. Calls to `put` items into the buffer are guarded by a check for a full buffer using the Java conditional `wait/notify` idiom; calls to `get` items into the buffer are guarded similarly by a check for an empty buffer. As in [27], we ignored the details of what items are stored in the buffer and how these items are stored.

We analyzed an instance of the problem with two producer and two consumer threads. The abstraction of the program was driven by our selection that the constant `SIZE` should be abstracted with *signs*. Bandera's abstraction type inference determined that variable `count`, that stores the number of items currently stored in the buffer, should also be abstracted. Running JPF (using depth-first search) on this abstracted system detects a deadlock and produces a counter-example; analysis of this counter-example reveals that it is spurious. No choose-free counter-example were found. Using the heuristic searches, JPF found counter-examples that were shown to be feasible.

We reran our model checks changing the abstraction for `SIZE` with *range(0..1)* abstraction [18], which tracks concrete values 0 and 1, but abstracts the values less than 0 and greater than 1, by using the set of tokens {*below0*, *zero*, *one*, *above1*}; this produced a choose-free counter-example. We ran JPF in simulation mode guided by the counter-example produced using depth-first search and it was shown to be not feasible; the counter-examples reported by the heuristic searches were found to be feasible.

5.1.6 NestedMonitor

The **NestedMonitor** program is an implementation of the bounded buffer that may deadlock because of nested monitor calls. We analyzed an instance of the program with one producer thread and one consumer thread.

As with the previous example, we abstracted the constant `SIZE` (that records the number of items that are stored in the buffer) using *signs*. Checking the abstracted system yielded feasible counter-examples with each of the techniques.

5.1.7 ReplicatedWorkers

The **ReplicatedWorkers** system is a parameterizable job scheduler, where the user configures the computation to be performed in each job, the degree of parallelism and several pre-defined variations of scheduler behavior. An instance of this framework is a collection of similar computational elements, called workers. Each worker repeatedly accesses data from a shared work pool, processes the data, and produces new data elements which are returned to the pool. Users define the number of workers, the type of work data, and computations to be performed by a worker on a item. We checked for proper termination (i.e., that the computation can not terminate unless the work pool is empty or a worker signals termination). We encoded this with an assertion that uses a variable, `GlobalDone`, set to true when a worker thread signals termination. To analyze the framework, additional code was written to simulate the behavior of a driver that invokes the framework's operations and of stubs that implement the work done by workers. Similar to DEOS, we modeled the environment using explicit external nondeterminism.

The *signs* abstraction was applied to the variable that stores the number of work items in the pool. Checking the abstracted system yielded a choose-free counter-example. The counter-example reported by the depth-first search was found to be infeasible and during heuristic searches, JPF ran out of memory.

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.

There are three criteria that we believe are relevant when considering the effectiveness of the techniques presented in this paper: guarantees of counter-example feasibility, length of counter-example, and memory/time requirements of the analysis.

The data clearly show that counter-examples can be reduced significantly in length with respect to depth-first search; this alone makes it easier to diagnose the program fault. Choose-free search and both heuristics are effective in finding such short counter-examples. In some cases, the heuristics are more effective in reducing counter-example length. This is due to the fact that choose-free search operates depth-first in the subspace bounded by choose operations.

While breadth-first search can produce shorter counter-examples than choose-free search, they are not guaranteed to be feasible. This is why we investigated the choose-free heuristic, which is a hybrid of the two that performs a breadth-first traversal of the subspace bounded by choose operations. The price of the heuristic searches is that they explicitly store the search frontier, unlike the depth-first searches. This can lead to significant memory consumption, depending on the problem. For the two largest examples we considered, **DEOS** and **ReplicatedWorkers**, these searches exhausted memory while choose-free search completed successfully. More experiments with large software systems is needed to understand the relative effectiveness of choose-free search and its heuristic variant.

We believe that the guarantee of feasibility is important since it will focus the user's attention on only those counter-examples for which analysis will lead to fault detection. In general, one would prefer a shorter feasible counter-example to a longer one. Another possible variation of choose-free search that is guaranteed to produce the shortest choose-free counter-example, as the heuristic search does, but without the cost associated with a breadth-first traversal is to perform an exhaustive depth-first search of the choice bounded sub-space recording the shortest counter-example encountered and overwriting it when a shorter one is found.

We note that the heuristic searches can be used only for checking for assertion violations and for deadlock, while choose-free search can be used when checking properties written in temporal logic.

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 [34]. In that work a predicate abstraction and an invariant for **DEOS** and four 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 work

In our previous work [18], we focused on the specification, generation, selection and compilation of abstractions for Java programs. In this paper, we detail techniques for analyzing counter-examples and provide evidence for their usefulness on several non-trivial Java programs.

The abstractions we use correspond to *free* abstraction relations from [14]. When looking only at the choose-free paths, the model checker 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 [14]. 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 abstract transitions when looking for defects.

Most existing work on counter-example analysis (e.g., [3, 7, 23, 30, 31]) is oriented towards the goal of verification; counter-example analysis drives abstraction refinement for the purpose of proving a property. In contrast, our work is oriented toward defect detection, and we view the integration of our work with abstraction refinement techniques as an interesting research topic for the future. Our biasing of the model checker yields a complete coverage of the sub-space of guaranteed feasible paths in the system rather than simply assessing the feasibility of a single counter-example from an unbiased model check.

The simulation technique from [7] can handle programs with multiple initial states and it uses forward analysis to perform a symbolic simulation of the concrete system using predicates that characterize the program data values. Unlike our simulation technique, the method from [7] does not keep a correspondence between concrete and abstract transitions; hence, rather than determine the next concrete state, it must compute (at each step of the simulation) the set of *all* possible next concrete states. This method, which is implemented in the symbolic model checker NuSMV [6], is dependent on the computability of the inverse of the abstraction function and also on the decidability of whether a set of states is empty or not. Similar restrictions apply to the approaches presented in [23, 30], where theorem proving is used to rule out spurious counter-examples. Backward analysis, that computes pre-images of the violating abstract state over the given trace, is used to obtain information to refine the abstractions. In SLAM [3], sequential C programs are abstracted into *Boolean programs* (programs in which variables and procedure parameters are always Boolean). Feasibility of abstract counter-examples is checked using symbolic execution, in which a heuristic decision procedure is used to try and decide whether the abstract path is feasible or not. Unlike our approach, these tools and techniques are not concerned with property abstraction.

We 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 several approaches for finding feasible abstract counter-examples when model checking software. These include adaptations of state space search algorithms to focus on the choose-free subspace and abstract counter-example guided simulation of the concrete program.

Based on experimentation with an implementation of these techniques in a Java model checking tool we have found the combination of techniques to be capable of detecting guaranteed feasible counter-examples in every case. This enables users to apply aggressive abstractions to their programs to speed analysis without sacrificing the ability to detect errors.

Our approach treats both abstraction of program data and the property to be checked. Furthermore, it takes into account the differences between abstraction in the environment and abstraction in the system under analysis. Thus, our approach is well-adapted to finding errors in real software.

We have demonstrated that choose-free search and abstract counter-example guided concrete simulation are not tightly bound to a particular model checking framework by adapting them from JPF to Bandera.

Finally, we believe that our light-weight counter-example analysis techniques can be combined with other counter-example analysis methods to provide a suite of methods that vary in cost and in their ability to precisely analyze counter-examples.

References

1. Abramsky, S., Hankin, C.: Abstract interpretation of declarative languages. Ellis Horwood, Chichester, UK, 1987
2. Arnold, K., Goslin, J.: The Java programming language. Addison-Wesley, Reading, Mass., USA, 1998
3. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.), Model Checking Software, Proc. 8th International SPIN Workshop, Lecture Notes in Computer Science, vol. 2057. Springer, Berlin Heidelberg New York, 2001, pp. 103–122
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01), ACM SIGPLAN Notices 36(5):203–213, 2001
5. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.), Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Lecture Notes in Computer Science, vol. 2031. Springer, Berlin Heidelberg New York, 2001, pp. 268–283
6. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. Int J Software Tools Technol Transfer 2(4):410–425, 2000
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.), Proc. 12th International Conference on Computer-Aided Verification (CAV'00), Lecture Notes in Computer Science, vol. 1855. Springer, Berlin Heidelberg New York, 2000, pp. 154–169
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans Program Lang Syst 16(5):1512–1542, 1994
9. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby.: Expressing checkable properties of dynamic systems: the Bandera specification language. Int J Software Tools Technol Transfer, (to appear)TSc
10. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby.: Bandera: a source-level interface for model checking Java programs. In: Ghezzi, C., Jazayeri, M., Wolf, A. (eds.), Proc. 22nd International Conference on Software Engineering (ICSE'00), pp. 762–765. ACM, New York, 2000

11. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Ghezzi, C., Jazayeri, M., Wolf, A. (eds.), Proc. of the 22nd International Conference on Software Engineering (ICSE'00), pp. 439–448. ACM, 2000
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages, pp. 238–252, 1977
13. Dams, D., Gerth, R., Döhmen, G., Herrmann, R., Kelb, P., Pargmann, H.: Model checking using adaptive state and data abstraction. In: Dill, D.L. (ed.), Proc. 6th International Conference on Computer-Aided Verification (CAV'94), Lecture Notes in Computer Science, vol. 818. Springer, Berlin Heidelberg New York, 1994, pp. 455–467
14. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans Program Lang Syst 19(2):253–291, 1997
15. Demartini, C., Iosif, R., Sisto, R.: A deadlock detection tool for concurrent Java programs. Software Pract Exper 29(7):577–603, 1999
16. Dwyer, M.B., Hatcliff, J., Păsăreanu, C.S., Robby.: Bandera case studies. Available at: `http://www.cis.ksu.edu/santos/bandera/#case-studies`, 2001
17. Dwyer M.B., Wallentine, V.: Object-oriented coordination abstractions for parallel software. In: Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), 1997
18. Dwyer, M.B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C.S., Robby, Visser, W., Zheng, H.: Tool-supported program abstraction for finite-state verification. In: Proc. 23rd International Conference on Software Engineering (ICSE'01), 2001
19. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.), Proc. 8th International Static Analysis Symposium (SAS'01), Lecture Notes in Computer Science, vol. 2126. Springer, Berlin Heidelberg New York, 2001, pp. 356–373
20. Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S., Zheng, H.: A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In: Cortesi, A., Filé, G.(eds.), Proc. 6th International Static Analysis Symposium (SAS'99), Lecture Notes in Computer Science, vol. 1694. Springer, Berlin Heidelberg New York, 1999, pp. 1–18
21. Holzmann, G.J.: The model checker SPIN. IEEE Transa Software Eng 23(5):279–294, 1997
22. Kesten, Y., Pnueli, A.: Modularization and abstraction: the keys to practical formal verification. In: Brim, L., Gruska, J., Zlatuska, J. (eds.), Proc. 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98), Lecture Notes in Computer Science, vol. 1450. Springer, Berlin Heidelberg New York, 1998, pp. 54–71
23. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: Margaria, T., Yi, W. (eds.), Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Lecture Notes in Computer Science, vol. 2301. Springer, Berlin Heidelberg New York, 2001, pp. 98–112
24. Lea, D.: Concurrent programming in Java[tm], 2nd edn. Design principles and patterns. The Java Series. Addison-Wesley, Reading, Mass., USA, 1999
25. Lindholm, T., Yellin, F.: The Java virtual machine specification. Addison-Wesley, Reading, Mass., USA, 1999
26. Loiseaux, C., Graf, S., Sifakis, J., Bouajiani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods Syst Des 6(1):11–44, 1995
27. Magee, J., Kramer, J.: Concurrency: state models and Java programs. Wiley, New York, 1999
28. Penix, J., Visser, W., Engstrom, E., Larson, A., Weininger, N.: Verification of time partitioning in the DEOS real-time scheduling kernel. In: Ghezzi, C., Jazayeri, M., Wolf, A. (eds.), Proc. 22nd International Conference on Software Engineering (ICSE'00), pp. 488–497. ACM, New York, 2000
29. Păsăreanu, C.S., Dwyer, M.B., Visser, W.: Finding feasible counter-examples when model checking abstracted Java programs. In: Margaria, T., Yi, W. (eds.), Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Lecture Notes in Computer Science, vol. 2031. Springer, Berlin Heidelberg New York, 2001, pp. 284–298
30. Rusu, V., Singerman, E.: On proving safety properties by integrating static analysis, theorem proving and abstraction. In: Cleaveland, R. (ed.), Proc. 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Lecture Notes in Computer Science, vol. 1579. Springer, Berlin Heidelberg New York, 1999, pp. 178–192
31. Saïdi, H.: Model checking guided abstraction and analysis. In: Palsberg, J. (ed.), Proc. 7th International Static Analysis Symposium (SAS'00), Lecture Notes in Computer Science, vol. 1824. Springer, Berlin Heidelberg New York, 2000, pp. 377–396
32. Valle-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. CASCON'99, 1999
33. Visser, W., Brat, G., Havelund, K., Park, S.: Model checking programs. In: Proc. 15th IEEE International Conference on Automated Software Engineering (ASE'00), pp. 3–12. IEEE Computer, New York, 2000
34. Visser, W., Park, S., Penix, J.: Applying predicate abstraction to model check object-oriented programs. In: Proc. 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice, August 2000