Bandera : A Source-level Interface for Model Checking Java Programs

James C. Corbett
University of Hawai'i
Dept. of Info. and Comp. Science
Honolulu, HI 96822
+1 808 956 6107
corbett@hawaii.edu

Matthew B. Dwyer, John Hatcliff, Robby
Kansas State University
Dept. of Comp. and Info. Sciences
Manhattan, KS 66506-2302
+1 785 532 6350
{dwyer,hatcliff,robby}@cis.ksu.edu

ABSTRACT

Despite emerging tool support for assertion-checking and testing of object-oriented programs, providing convincing evidence of program correctness remains a difficult challenge. This is especially true for multi-threaded programs. Techniques for reasoning about finite-state systems have been developing rapidly over the past decade and have the potential to form the basis of powerful software validation technologies.

We have developed the Bandera toolset [1] to harness the power of existing model checking tools to apply them to reason about correctness requirements of Java programs. Bandera provides tool support for defining and managing collections of requirements for a program, for extracting compact finite-state models of the program to enable tractable analysis, and for displaying analysis results to the user through a debugger-like interface. This paper describes and illustrates the use of Bandera's source-level user interface for model checking Java programs.

Keywords

Java, model checking, program analysis, debugging and testing

1 INTRODUCTION

Testing multi-threaded programs is much more difficult than testing sequential programs. This is partly due to the fact that a multi-threaded program's execution is not determined solely by its input values; the relative speed of thread execution (which is typically beyond a developers control) can also affect a program's execution by reordering individual program operations, e.g., method calls. For this reason, it is not uncommon for multi-threaded Java programs to exhibit intermittent failures that are difficult to find and reproduce. Finite-state verification techniques, such as model checking, present a possible solution to this problem. These techniques *exhaustively* check a finite-state model of a system for violations of a system correctness requirement, or property, and are thus able to reason about all of the possible orderings of program operations that a system could exhibit. This can be a powerful means of insuring that a system operates correctly, or conversely, of revealing subtle program errors.

Bandera [1] is an integrated collection of program analysis and transformation components that enable 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 [2] and SMV [3], to provide state-of-the-art analysis engines for checking program-property correspondance. These tools vary greatly in the specification and system description languages that they accept and in the kind of feedback they provide to users about the results of the analysis. Bandera hides these details from the user and presents a single uniform interface oriented around the Java source text.

In this paper, we describe parts of the Bandera user interface related to user-guided finite-state model extraction and interpretation of analysis results. The next section gives a brief overview of these parts of Bandera. Section 3 presents a scenario depicting the use of Bandera for analyzing a simple multi-threaded Java program for freedom from deadlock. We conclude in Section 4.

2 THE BANDERA TOOLSET

Model checking is a technique for systematically searching the possible behaviors of a system for certain kinds of errors. First, the system (in our case, a Java program) is modeled as a finite-state transition system. Each state represents an abstraction of the program's state and each transition represents the execution of one or more statements transforming this state. Second, a property of the system is expressed as an assertion or in a temporal logic; the property describes some constraint on the permissible state/event sequences in the finite-state model. Third, a model checking tool algorithmically determines whether all paths through the finite-

```java
public class Monitor {
 public static void main(String argv[]) {
  Wrapper w = new Wrapper();
  (new Lo(w)).start();
  (new Hi(w)).start();
 }
}
class Wrapper {
 private Pair p;
 Wrapper(Pair x) {p = new Pair();}
 public synchronized void addl() {
  p.inclo();
 }
 public synchronized void addh() {
  p.inchi();
 }
}
```

```java
class Pair {
 private int hi = 0;
 private int lo = 0;
 private int gap = 0;
 public synchronized void inclo() {
  while (lo <= hi)
   try { wait(); }
   catch ( InterruptedException ex) {}
  lo++;
 }
 public synchronized void inchi() {
  hi++;
  if (hi-lo > gap) gap = hi-lo;
  notify();
 }
}
```

```java
class Hi extends Thread {
 private Wrapper w;
 Hi(Wrapper x) {w = x;}
 public void run() {
  for (int i=0; i<50; i++)
   w.addh();
 }
}
class Lo extends Thread {
 private Wrapper w;
 Lo(Wrapper x) {w = x;}
 public void run() {
  for (int i=0; i<50; i++)
   w.addl();
 }
}
```

Figure 1: Nested Monitor Deadlock in Java

state transition system satisfy the property. If not, the model checker displays a path through the transition system that violates the property; this path, called a counter-example, can be interpreted as a behavior of the system and used to understand the error.

Bandera supports the user in describing correctness properties, creating efficient models, and interpreting counter-examples. In this paper, we focus on the interfaces to Bandera's slicing and abstraction components for model creation, and on the counter-example simulation interface. The reader is refered to [1] (in this volume) for more details about Bandera's other components.

Property-directed Java Slicer

The Bandera slicing component compresses paths in the program by removing statements, variables, and data structures that are irrelevant for checking a given property. Bandera extracts a description of the program variables and statements that are directly related to the property under analysis. This description forms the criterion for slicing the program based on both traditional data and control dependences as well as dependences that capture inter-thread synchronization. Slicing is fully-automatic and is guaranteed to preserve all program behavior that is relevant to the requirement under analysis.

Type-based Abstraction of Java

Bandera includes a component that systematically compiles abstractions into the program text. Users can abstract local variables and class fields by selecting from a library of abstraction definitions that are appropriate for the type of the variable or field. These abstractions effectively reduce the range of values that a variable (field) can range over. This can dramatically reduce the number of states in the model and speed analysis.

Bandera abstractions are designed to be sound with respect to verification of properties of all program executions (as opposed to properties that are required to hold only on some execution). Unsound approximations can also be used when one is only interested in finding possible program defects.

Counter-example Simulation

When a model check is unsuccesful it produces a counter-example, which is a sequence of system states that violate the property under analysis. Bandera includes a component that records the sequence of state transitions and allows forward (backward) stepping through the counter-example and display of the values of program variables at each state.

3 MODEL CHECKING A JAVA PROGRAM

We illustrate the interfaces to Bandera by way of a simple example. Figure 1 shows the source code for a simple multi-threaded Java program. This program exhibits a not-uncommon problem with composing synchronized objects in Java. `Pair` defines an object whose synchronization protocol enforces the condition that the `lo` field remains below the `hi` field. The `Wrapper` class adapts the interface of a `Pair` to a new signature; in Java this might be required, for example, to conform to an interface definition. The synchronized methods of the `Wrapper` object, however, interfere with the `Pairs` synchronization protocol. This results in a composite sub-system that can lead to a so-called *nested monitor deadlock*. In the rest of this section, we illustrate one scenario for using Bandera to check whether this program is free from deadlock.

Property-directed Slicing

Analyzing a freedom from deadlock property requires the preservation of all program statements that may affect whether a thread blocks, e.g., `synchronized` statements, `wait`, and `notify` calls. Figure 2 shows how Bandera presents the sliced source code the example as faded text; note also that the `gap` field of the `Pair` object has been removed.

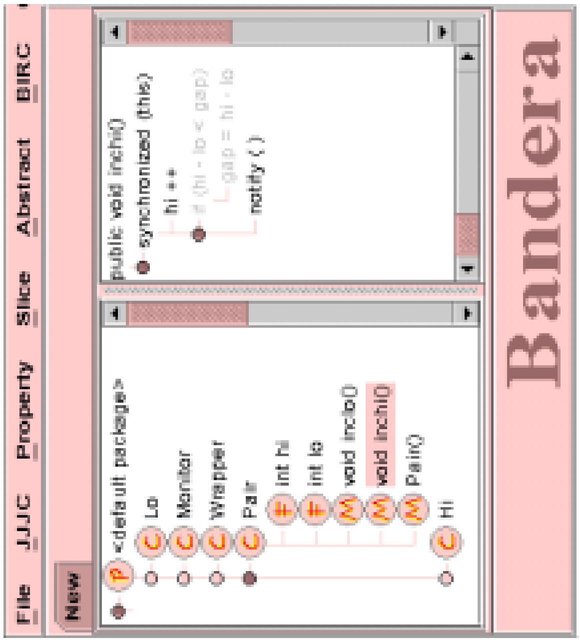

Figure 2: Sliced Source

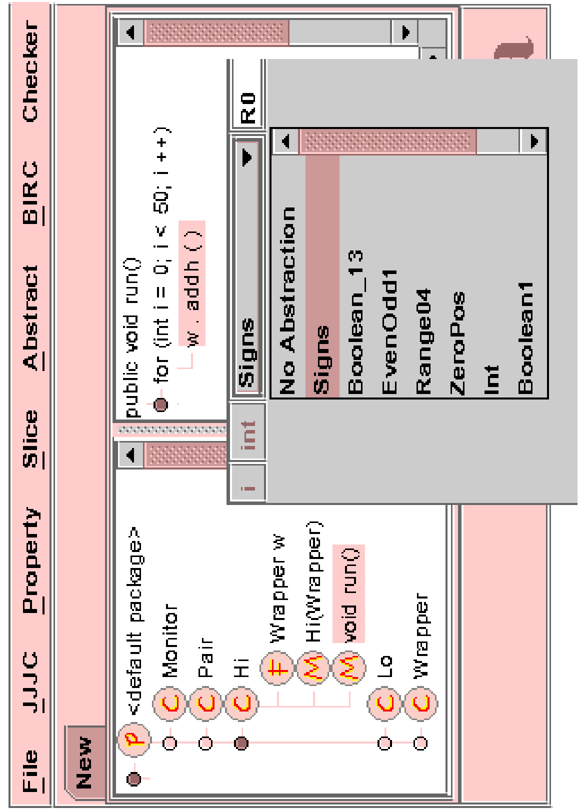

Figure 3: Abstraction Selection

Abstraction of Program Data

After slicing the user may choose to further reduce program's model by applying abstractions to selected program variables, class fields, and whole classes. For our example, we have chosen to abstract the loop index variable, `i`, in the `run` methods of the `Lo` and `Hi` classes. Figure 3 illustrates how users may select from a library of abstractions for the `int` type to be applied to `i`. The `Signs` abstraction will replace `POS`itive (`NEG`ative) values with a single representative abstract value and the value `0` with the abstract value `R0`. Bandera will also substitute abstract implementations for the `int` operations in the program. In the example, `i++` will be abstracted to the constant `POS` since the abstraction engine determines that since `i` is either `R0` or `POS` then adding a `POS` (the abstraction of the `1` in `++`) always yields a `POS`. Similarly, the test `i` $<$ `50` will yields two cases: `R0` $<$ `POS` on the first iteration, which evaluates to true, and `POS` $<$ `POS` on subsequent iterations, which can be either true or false. In the latter case non-deterministic choice is used in defining the model. This has the effect of modeling the `for` statement as a loop that is guaranteed to iterate at least once, but can continue for arbitrarily long. This yields a model that includes the set of all real program executions and is, thus, sound with respect to freedom from deadlock. It also has the effect of dramatically speeding the performance of most model checkers.

Counter-example Simulation

Just as a debugger can aid a user in diagnosing the behavior of a program execution, Bandera provides an interface that aids the user in diagnosing the reason for a failed analysis. Model checker generated counter-examples are expressed in terms of a sequence of operations defined on Bandera's low-level intermediate representation. This sequence of operations can be simulated and the current execution location and values of, potentially abstracted, variables are mapped back to the original Java source code for display to the user.

Figure 4 illustrates how the source location associated with a step in the counter-example is displayed and how the values of program variables that are in-scope at that step are displayed. The thread, `Hi.run`, and statement in that thread, `w.addh()`, corresponding to the current location are highlighted. In addition the *Step* window shows all of the fields that are in scope in any active thread; clicking on a field prints its value in the right-hand pane. Clicking on a field of reference type, denoted with a filled circle, expands to show the fields of the referenced object. Successive clicking on references allows for navigation of the state of the heap. Object values are printed as a unique instance number, e.g., `Wrapper#0`, the object's lock, `Holding`, and the object's wait set, `Waiting`. Figure 4 shows that object `Wrapper#0` has its lock held by thread `Lo.run`; this is preventing the current thread `Hi.run` from executing the synchronized method call `w.addh`. Figure 5 shows that `Lo.run` is in

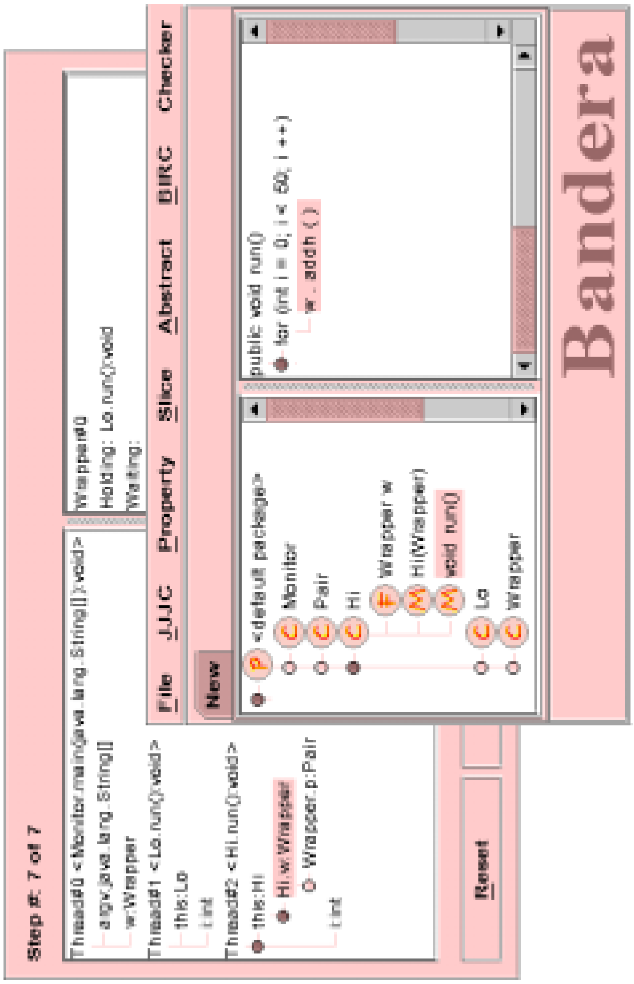

Figure 4: Counter-example Location Display

the wait set of object `Pair#4`, the object contained in `Wrapper#0`. This is the classic symptom of a nested monitor deadlock : a thread holds the lock on the outer object and is waiting for a condition to hold on the inner object, but in doing so it locks out any other thread from causing that condition to become true.

4 CONCLUSIONS

We believe that a source-level interface to model checking tools can provide a valuable software verification and validation capability. There are many technological and methodological issues to be explored in making this kind of technology usable by practicing software developers and in scaling it to large complex Java applications. The Bandera web-site (`http://www.cis.ksu.edu/~santos/bandera`) contains up-to-date information on our efforts to make software model checking practical.

ACKNOWLEDGEMENTS

The authors would like to thank the rest of the Bandera group : Corina Pasareanu, Shawn Laubach, Hongjun Zheng, Oksana Tkachuk, Daqi Li, and Venkatesh Ranganath. This work was supported in part by NSF under grants CCR-9407182, CCR-9703094, CCR-9708184 and CCR-9901605, by NASA under grant NAG-02-1209, and by Sun Microsystems under grant EDUD-7824-00130-US.

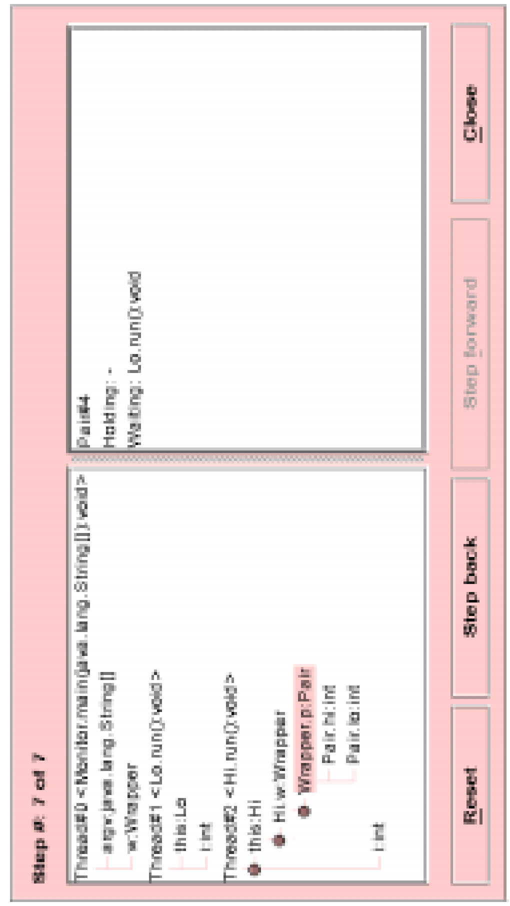

Figure 5: Counter-example Value Display

REFERENCES

[1] 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 *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[2] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[3] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.