Bandera : Extracting Finite-state Models from Java Source Code

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

Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, Hongjun Zheng
Kansas State University
Department of Computing and Information Sciences
Manhattan, KS 66506-2302
+1 785 532 6350
{dwyer,hatcliff,laubach,pcorina,robby,zheng}@cis.ksu.edu

ABSTRACT

Finite-state verification techniques, such as model checking, have shown promise as a cost-effective means for finding defects in hardware designs. To date, the application of these techniques to software has been hindered by several obstacles. Chief among these is the problem of constructing a finite-state model that approximates the executable behavior of the software system of interest. Current best-practice involves hand-construction of models which is expensive (prohibitive for all but the smallest systems), prone to errors (which can result in misleading verification results), and difficult to optimize (which is necessary to combat the exponential complexity of verification algorithms).

In this paper, we describe an integrated collection of program analysis and transformation components, called Bandera, that enables the automatic extraction of safe, compact finite-state models from program source code. Bandera takes as input Java source code and generates a program model in the input language of one of several existing verification tools; Bandera also maps verifier outputs back to the original source code. We discuss the major components of Bandera and give an overview of how it can be used to model check correctness properties of Java programs.

Keywords

program verification, model checking, model extraction, slicing, abstract interpretation, program specialization

1 INTRODUCTION

In recent years, finite-state verification techniques such as model checking have renewed interest in formal verification of computer systems. These techniques exhaustively check a finite-state model of a system for violations of a system requirement formally specified in some temporal logic (e.g., LTL [20]). This approach provides a level of confidence comparable to that of a machine-checked proof of the requirement's correctness without the extensive human guidance required by theorem provers—once the model and property specification are constructed, the verification is fully automatic, albeit potentially time-consuming. Tools supporting these techniques [14, 21] have matured to the point where hardware manufacturers frequently use them to validate their designs.

The transfer of this technology from research to practice has been much slower for software. One reason for this is the *model construction problem*: the semantic gap between the artifacts produced by software developers and those accepted by current verification tools. Most development is done with general-purpose programming languages (e.g., C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g., process algebras, state machines). In order to use a verification tool on a real program, the developer must extract an abstract mathematical model of the program's salient properties and specify this model in the input language of the verification tool. This process is both error-prone and time-consuming.

Another obstacle to the transfer of finite-state verification technology is the *state explosion problem*: the exponential increase in the size of a finite-state model as the number of system components grows. A variety of methods exist for curbing the state explosion when analyzing certain types of systems, and these methods have proven sufficient to make analysis of many hardware designs tractable. Unfortunately, software systems tend to have much more state than hardware components and thus must be more aggressively abstracted to produce tractable models.

Recent efforts have attacked the problem of model checking software in several ways. Some have taken a monolithic approach by building a dedicated model checker for a specific programming language, such as Erlang [18]. Others have built tools, such as JCAT [4] and Java PathFinder [13], that translate a program directly into a relatively expressive verifier input language, in this case Promela, the input language of the Spin model checker [14].

Although these tools can be useful, we see a number of significant limitations to these approaches. The monolithic approach makes it difficult to keep the checking engine state-of-the-art; new methods for curbing the state explosion must be recoded in the tool's dedicated engine. The translation approach often results in larger models, partly because of the mismatch between the semantics of the two languages, and partly because the translation does not consider the property being verified, thus the model cannot be customized for the property. Both approaches tend to lock the user into a single kind of checking technology, even though empirical studies have shown that the best analysis method varies by program [1]. Finally, these tools have limited support for the kinds of control/data abstraction used by human analysts when they build models by hand (e.g., removing irrelevant program features, reducing the cardinality of data types via symbolic abstraction).

Our goal is to overcome the major obstacles to finite-state verification of software by using a *component-based tool architecture for model extraction* based on the following design criteria:

- *Reuse of existing checking technologies.* Model checkers, especially the most widely used ones like Spin and SMV [21], are extremely sophisticated programs that have been crafted over many years by experts in the specific techniques employed by the tool. A re-implementation of the algorithms in these tools would likely yield inferior performance.
- *Automated support for the abstractions used by experienced model designers.* The most important single method for extracting tractable models of software is abstraction. Thus, our tool-set should go beyond simple translation and instead be structured like an optimizing compiler, employing complex transformations to optimize the "performance" (i.e., compactness) of the generated "code" (i.e., model). The transformations should be staged to improve their effectiveness and would rely on static analyses (e.g., object flow analysis, dependency analysis). Transformations commonly employed by human analysts and that should be supported include: slicing, abstract interpretation, and specialization.
- *Specialized models for specific properties.* Rather than construct a single model of the software system accurate enough to verify all relevant properties, the model should be customized (i.e., optimized) for a particular property. Although this is rarely done for hand-generated models (due to the effort required), generating a custom model for each property can significantly reduce the analysis effort and thus enhance scalability.
- *An open design for extensibility.* The tool-set should consist of a number of loosely connected components that communicate through a small set of well documented intermediate representations, thus allowing new abstraction techniques and checking engines to be added easily. For example, the back end of the tool-set should have a low-level intermediate representation that can easily be translated to the input languages of current model checkers.
- *Synergistic integration with existing testing and debugging techniques.* We want a tool-set that complements and can be used alongside existing development environments, which support testing and simulation. The supporting environment should encapsulate the details of the checking engines and allow counterexamples found to be displayed in a form that (*i*) is familiar and uniform (i.e., not specific to the model checker), and (*ii*) can be leveraged for testing, debugging, and simulation.

The main contribution of this paper is the description of Bandera: a component-based model extractor for Java programs designed to meet these goals. In particular, we describe the major components of Bandera:

Slicer The Bandera slicing component compresses paths in the program by removing control points, variables, and data structures that are irrelevant for checking a given property.

Abstraction Engine The Bandera abstraction engine allows the user to reduce the cardinality of data sets associated with variables. The tool also includes a language for specifying abstractions, which can be collected in an abstraction library for reuse.

Back End The back end of Bandera generates BIR: a low-level intermediate language based on guarded commands that abstracts common model checker input languages. The back end also contains a translator for each model checker supported.

User Interface Bandera has an advanced graphical user interface that facilitates interaction with the various components and displays counterexamples to the user in terms of the program source, like a debugger.

We also describe the application of the current implementation of Bandera (which handles a reasonably large subset of Java) to a non-trivial program.

In the next section, we give an overview of the process of model-checking software systems, and we summarize the main techniques used to construct models of software. Section 3 describes how Bandera provides automated support for these techniques: first, the user's view of Bandera is presented followed by a discussion of the internal architecture and a summary of the functionality of each Bandera component. Section 4 uses a small example to illustrate model checking of Java source code with Bandera. Section 5 discusses related work, and Section 6 concludes.

2 MODEL CHECKING SOFTWARE

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 desired property of the system is expressed in temporal logic [20]; 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-state transition system satisfy the property. If not, the model checker displays a path through the transition system violating the property; this path can be interpreted as a behavior of the system and used to understand the error.

Although this checking technique takes worst-case exponential time, it has been used to validate crucial properties of real software systems, e.g., [12]. Properties checked include freedom from deadlock, simple assertions, state-sequencing properties, and absence of null-pointer derefences. This success is partly due to ever improving methods for curbing the state explosion in model checking algorithms, but mostly due to the use of abstraction. The key to applying model checking to large software systems is not clever model *checkers* but clever model *builders* that abstract away most of the details of these programs, leaving only what is essential to verify a specific property.

Based on our experience, we believe that there are three main techniques that can be applied to build tractable models for verifying a given property: irrelevant component elimination, data abstraction, and component restriction.

I. Irrelevant component elimination: Many of the program components (classes, threads, variables, code) may not be relevant to the property being verified. For example, properties testing specific features of a program (e.g., selecting a certain menu item always brings up a particular dialog) are likely to be independent of most of the application code.

II. Data abstraction: After eliminating irrelevant components, some of the remaining variables, although relevant, might be recording more detail than is necessary for the property being verified. The range of such variables can often be safely abstracted to a much smaller set. For example, an application might store a set of items in a vector, but if the property being verified depends only on whether a particular item is in the vector, we could abstract the large number of vector states onto a small set $\{ItemInVector, ItemNotInVector\}$.

III. Component restriction: The two techniques above can often produce tractable models of software systems. If these methods fail, a *restricted model* of the program can usually be constructed by limiting the number of components and/or the ranges of variables (e.g., bounding the number of objects that can be created by an allocator, bounding the number of total execution steps). Restricted models do not capture all behaviors of the program, but since many design errors are manifest in small versions of a system, they can be useful for finding errors [5, 19].

Note that the *semantic gap* between systems and finite-state models is much wider for software than for hardware, and this may increase the need for component restriction. Model checkers have highly static input languages, whereas most software is written in high-level languages supporting a wide variety of dynamic constructs (e.g., unbounded heap allocation, unbounded recursion, dynamic creation of threads, polymorphism). If clever abstractions for these features cannot be defined, and if bounds on the degree of dynamism in these constructs cannot be inferred, we must construct a restricted model by imposing such bounds arbitrarily.

Besides creating challenges for model building, the semantic gap presents additional hurdles when diagnosing program errors. When the model checker finds a violation of a given property, the analyst must interpret the model checker's violating trace through the state transition system as a sequence of program statements. This can be difficult if the model was obtained from the program through many nontrivial transformations, as is normally the case. In addition, each program step usually corresponds to many steps in the transition system.

3 BANDERA

One of the primary goals of Bandera is to provide automated support for the model-construction and error trace interpretation techniques outlined in the previous section. Specifically, Bandera uses *slicing* to automate irrelevant component elimination, *abstract interpretation* to support data abstraction, and a model-generator that allows significant flexibility in setting bounds for various system components. Bandera also includes a collection of data structures for automatically mapping model-checker error traces back to the source level as well as facilities for graphical navigation of these traces.

Figure 1 illustrates the inputs supplied to Bandera when checking a program (a collection of `.java` files) against some requirement and the internal structure of the Bandera tool set. The user formalizes a requirement as, for example, a temporal logic formula. To aid the formalization, Bandera provides a menu-driven selection

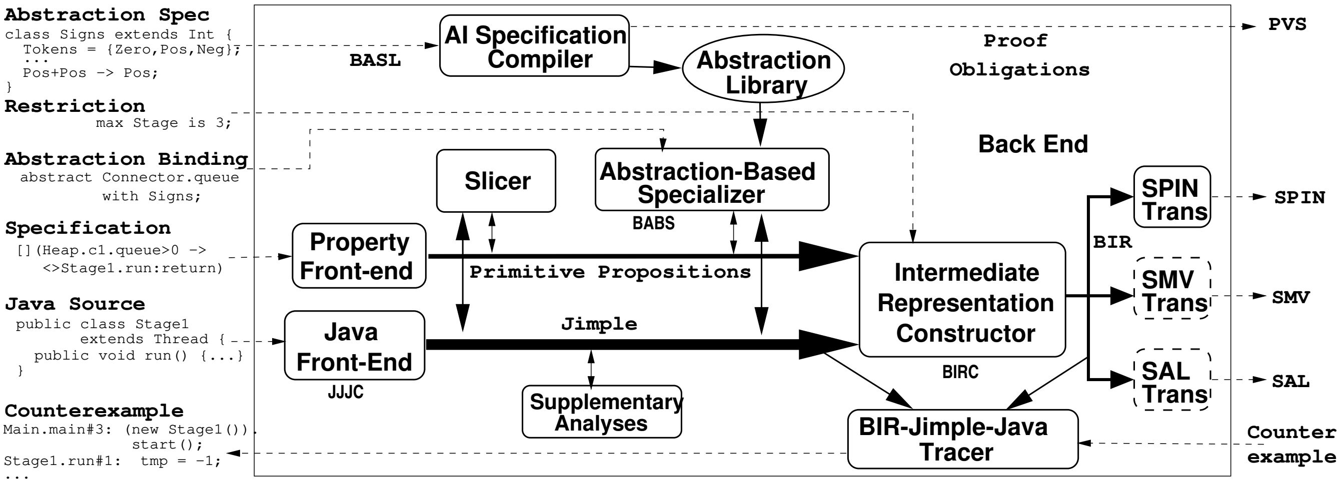

Figure 1: Bandera's Interfaces and Internals

of specification templates from an existing template library [7]. The user completes a specification by filling in the template's parameters with *primitive propositions* that describe the semantic features of the program that the user is interested in reasoning about. Currently, Bandera supports the definition of propositions that define a range of values for object fields that are defined as built-in types (e.g., `c1.queue>0`), and propositions that capture locations in the source code (e.g., `Stage1.run:return`— the return point of the `run` method of class `Stage1`). From these propositions, Bandera automatically derives a slicing criterion that can be used to slice away program components irrelevant to the property being checked. The user can subsequently, examine the remaining variables and fields of this sliced program to decide how the data for those components should be abstracted. Abstractions are selected from a library and bound to those variables that the user deems particularly needy of compaction (because of their contribution to the system's state space). As illustrated, it is also possible for user's to add new abstractions (e.g., the `Signs` abstraction specification). The abstraction engine then compiles the definitions of the chosen abstractions into the program, simplifies the program, and inlines all remaining method calls. Finally, this reduced program is fed to Bandera's back-end which generates a finite-state model in the input language of a chosen verification tool. Bandera interprets verifier output and when the property fails to hold on the generated model, the verifier-specific counter-example is mapped back to the user's original source code.

The architecture of Bandera is similar to an optimizing compiler. Compiler's use multiple intermediate languages to stage the transformation to machine code, Bandera uses multiple intermediate languages to stage the transformation from Java to model-checker input languages. The Bandera front-end translates Java to a a high-level intermediate language called Jimple; the Bandera back-end generates model-checker inputs from a low-level intermediate language of guarded commands called BIR (Bandera Intermediate Representation). Just as a conventional compiler relies on sophisticated static analyses and transformations to produce optimized code, Bandera relies on conventional data-flow, control-flow and dependency analyses, slicing and specialization transformations, as well as several supplementary analysis to produce compact models. Below we summarize the functionality of each of the Bandera components and intermediate representations.

Soot/Jimple

Bandera is built on top of the *Soot* compiler framework developed by the Sable group at the University of McGill [23]. In the Soot framework, Java programs are translated to the intermediate language *Jimple* — one of several intermediate languages supported by Soot. We have developed our own front-end called JJJC (Java-to-Jimple-to-Java Compiler) that maintains a tight correspondence between a Java source program and its Jimple representation. Given a node in a program's Jimple representation, JJJC can return the corresponding node in the Java abstract syntax tree (AST) for the program (and vice versa). These bi-directional mappings, together with similar mappings for our other intermediate language BIR, facilitate the mapping of model-checker counter-example traces back to source code traces. Downstream components such as the slicer and specializer that transform the Jimple representation must be careful to maintain the mappings. For instance, the specializer performs inlining which duplicates the bodies of some methods. Thus, after specialization, a single Java source node may map to multiple points in the Jimple representation. Similarly, the slicer removes nodes from the Jimple representation. Thus, the mapping from Java to Jimple must be modified to indicate the corresponding Jimple node has been sliced away.

Slicer

Given a program P and some statements of interest $C = \{s_1, \ldots, s_k\}$ from P called the *slicing criterion*, a program slicer will compute a reduced version of P by removing statements of P that do not affect the computation at the criterion statements C. When checking a program P against a specification ϕ, Bandera uses slicing to remove the statements of P that do not affect the satisfaction of ϕ. Thus, the specification ϕ holds for P if and only if ϕ holds for the reduced version of P (*i.e.*, the reduction of P is sound and complete with respect to ϕ) [11].

In recent work [11], we showed that the slicing transformation can driven by generating a slicing criterion C_ϕ based only on the primitive propositions in ϕ. As an example, consider the LTL response specification from Figure 1. This property states that along all paths, if execution reaches a point where the `queue` field of the `Heap.c1` object has a value greater than 0, then eventually the return point of the `run` method of class `Stage1` will be reached. From this specification, Bandera generates a slicing criterion, containing the return statement of method `Stage1.run` as well as all statements that contain assignments to `Heap.c1.queue`. The slicing algorithm guarantees the preservation of all program components that can affect (1) the values assigned to `Heap.c1.queue` as well as (2) the relative order of execution of `Stage1.run`'s return and assignments to `Heap.c1.queue`.

Building a slicer for Java requires a significant amount of effort. Fortunately, except for issues surrounding Java's concurrency primitives we were able to carry out most of the development using previously developed slicing techniques based on program dependence graphs, e.g., [17]. In recent work, we gave a formal presentation of slicing that includes additional notions of dependence that arise in Java's concurrency model [8]. These includes dependencies due to possibly infinite delays in waiting for locks or notification *via* Java's `notify/notifyall`, data dependencies due to access/definition of shared variables, and dependencies between program statements and the monitor delimiters that enclose them.

The effectiveness of slicing for reducing program models varies depending on the structure of the program. In some systems that we have considered, slicing removes entire threads and dramatically reduces the state space. In other cases, where program components are tightly coupled or where large sections of the program are relevant to the specification, the slicing reduction is only moderate. However, since slicing is cheap compared to the overall cost of model-checking and since it is totally automatic, we almost always use Bandera with the slicing option enabled.

Abstraction-Based Specializer

The Bandera Abstraction-Based Specializer (BABS) provides automated support for reducing model size *via* data abstraction. This is useful when a specification to be checked does not depend on the program's concrete values but instead depends only on *properties* of those values. For example, as described in the previous section, a vector can be abstracted to $\{\mathit{ItemInVector}, \mathit{ItemNotInVector}\}$ if the specification depends only on a particular item being in the vector.

Given an appropriate definition of an abstraction, the specialization engine will transform the source code into a specialized version where all concrete operations and tests on the relevant vector objects (e.g., method calls on the vector class) are replaced with abstract versions that manipulate tokens representing the abstract values $\{\mathit{ItemInVector}, \mathit{ItemNotInVector}\}$. Since information is lost in this transformation, operations and tests that relied on the lost information can no longer be determined completely in the abstract program. For instance, in the vector example the length of the abstracted vector cannot be determined. Values that cannot be determined are represented with a special token $\top$. When a $\top$ token flows into the test of a conditional expression in the abstract program, the specialization engine inserts a flag that informs the downstream model construction components to implement the test as a non-deterministic choice between the true and false branches.

The user guides BABS in incorporating abstractions by binding variables to entries from an *abstraction library*. The library entries are indexed by concrete type, and each entry implements an abstract version of its corresponding concrete type. Since abstractions are incorporated on a per variable basis, two different variables of the same concrete type can have different abstract types. For example, if I_1 and I_2 are both `int` abstractions, then variable `int x` may be bound to I_1 and variable `int y` may be bound to I_2. After the user has chosen abstractions for relevant variables, a type inference phase propagates this information throughout the program and infers abstraction types for the remaining variables and for each expression in the program. Type inference also informs the user when there is an abstraction type error.

Each abstraction library entry is automatically generated from a high-level description written in the Bandera Abstraction Specification Language (BASL). In its most general form, a BASL specification consists of a declaration of a finite set of abstract tokens, an abstraction function that maps each concrete Java value to an abstract token, and an abstract operation for each operation (method) of the concrete type (class). A rule-based format that incorporates pattern matching simplifies the definition of abstract operations. We have used BASL to define abstractions for integers, e.g., general forms of range abstractions, such as an abstraction that preserves an integer's *sign*, and modulo abstractions, such as an *even-odd* abstraction. We have recently designed an extension to BASL that supports abstractions for arrays, strings, and general objects.

From each operation in a BASL specification, the BASL compiler generates a rather sophisticated Java implementation that includes code to propagate abstract tokens as well as code to generate specialized versions of the operation if the operation cannot be completely symbolically executed. For abstractions over primitive types, the BASL compiler also generates declarations and correctness proof obligations for the theorem prover PVS [22]. We are currently experimenting with generating BASL definitions using a variant of predicate-abstraction and PVS. Interestingly, our use of PVS to prove abstraction correctness and to generate abstractions has required only the "grind" facility of the theorem-prover.

The design of BASL and the transformations implemented by BABS are grounded in the rigorously developed framework of *abstract interpretation* [3]. In previous work, we have formalized and proven the correctness of a simplified version of the abstraction-based specialized engine [9]. Despite this solid formal foundation, BASL is designed with a compact notation that shields the user from the technical machinery of abstract interpretation, thereby allowing even non-expert users to add abstractions to the Bandera's abstraction library.

Back End

The Bandera back end is like a code generator, taking the sliced and abstracted program and producing verifier-specific representations for targeted verifiers. The back end components communicate through BIR, the Bandera Intermediate Representation, an intermediary between compiler-based representations and verifier-based representations. As shown in Figure 1, the back end has one fixed component called BIRC (Bandera Intermediate Representation Constructor) that accepts a restricted form of Jimple and produces BIR. For each supported verifier, there is also a translator component that accepts the program represented in BIR and generates input for that verifier. A translator for Spin is complete, and translators for SMV and Stanford's forthcoming SAL model checker are under construction.

BIR is a guarded command language for describing state transition systems. The main purpose of BIR is to provide a simple interface for writing translators for target verifiers—to use Bandera with a new verifier, one must only write a translator from BIR to the input language of the verifier. The guarded command style of BIR meshes well with the input languages of existing model checkers.

BIR contains some higher-level constructs to facilitate modeling Java, such as threads, Java-like locks (supporting wait/notify), and a bounded form of heap allocation. Rather than choose an implementation of these constructs and remove them from BIR (e.g., model a lock as a boolean variable), we allow the translators to implement these constructs in whatever way is most efficient in the verifier input language. BIR also provides other kinds of information that can aid translators in producing more compact models. For example, a guarded command can be labeled *invisible*, indicating that it can be executed atomically with its successor. The set of local variables that are live at each control location can be specified (dead variables can be set to a fixed value for Spin or left unconstrained for SMV).

BIRC translates a subset of Jimple to BIR. Java locals and instance variables are mapped onto BIR state variables and record fields. The Jimple statement graph is traversed to construct a set of guarded commands for each thread. Each guarded command is marked as visible/invisible based on the kind of data accesses (e.g., operations on locals are invisible). BIRC also accepts a set of expressions used to define primitive propositions in the model (e.g., a thread is at a specific statement, a variable has a given value). BIRC embeds these proposition definitions into the BIR and insures that any program statement that changes the value of one of these primitive propositions will cause a visible state change in the model.

The Spin translator accepts a BIR representation and produces a Promela model of the system, suitable for input to the Spin model checker. Promela is quite expressive and has the same basic execution model as BIR (asynchronous processes executing guarded commands), thus the translation is straightforward. Promela `atomic` blocks are used to collapse sequences of invisible commands. Locks are implemented using a `struct` containing the lock state, owner, acquisition count, and wait set.

Supplementary Analyses

The effectiveness of each of the Bandera components can be improved with the aid of additional static analysis results. Bandera is designed to allow the integration of a variety of analyses whose results can be used to boost the precision of the primary model extraction transformations so as to produce more accurate and compact models. Examples of these analyses include a lock safety analysis [8] whose results can be used to refine the inter-thread dependences used in slicing, and a highly-parameterized object flow analysis that computes information about the object values flowing to selected program points that can also be used to refine

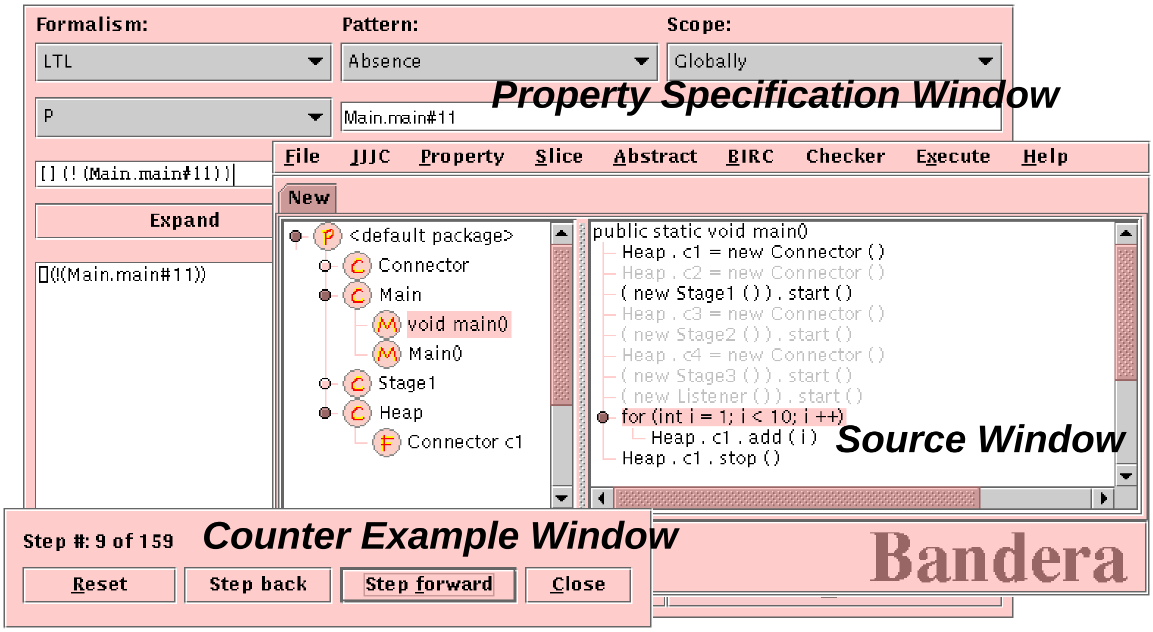

Figure 2: Bandera User Interface

dependences as well as to guide enable precise inlining of methods.

4 MODEL CHECKING JAVA PROGRAMS

The Bandera tool-set provides both a command-line and a graphical user interface, called the *Bandera User Interface* (BUI), for configuring the model extraction process. Figure 2 illustrates several of the BUI's features including: source-level code views of the results of applying the different Bandera components, e.g., sliced code is faded, support for writing formal specifications based on specification patterns [7], e.g., the global-absence property, and support that allows users to view counter-example information by navigating through the source-code statements. In addition the BUI allows users to control the abstractions used in model extraction and to configure the run-time options for the verifier that they select. We view the BUI as an interactive interface for configuring model extractions, configuration options set with the BUI can be dumped for input to the command-line interface so that an extraction can be run repeatedly as the code changes.

In the remainder of this Section, we describe the application of Bandera to check several properties of a multi-threaded Java program.

Applying Bandera

Bandera's collection of analysis and transformation components can applied in a variety of different ways to extract a finite-state model from Java source code. From initial experience using Bandera on several Java programs we have found that is always desirable to slice and that it is relatively easy to pinpoint variables for abstraction. Furthermore, even though selection of optimal abstractions, i.e., ones that give maximal reduction but preserve the property being checked, is a difficult problem, initial experience suggests that even naive abstraction choices are often effective. Thus, our basic model extraction scenario involves slicing based on the property followed by abstraction and model generation.

We applied the basic model extraction scenario to analyze properties of the program, given in Figure 3, that implements a pipelined computation, where each pipeline stage executes as a separate thread. Stages interact through `Connector` objects that provide methods for `add`ing and `take`ing data; these methods are typical of one form of conditional `wait-notify` synchronization in Java programs. The `main` method constructs `Connectors`, then creates and `starts` `Stage`*i* and `Listener` objects that implement the stages.

This code is slightly different from the Java code one might typically implement to solve this problem. These differences reflect limitations in the current Bandera tool-set that will be lifted in the next few months as additional tool capabilities are integrated. Currently each instance of a `Thread` object is converted to a separate class. Program specialization is used to unroll loops that contain constructor calls on `Thread` objects, and their subtypes, and to specialize the per-object `run` methods based on values of constructor parameters. In the code in Figure 3, this results in four copies of the `Stage` class with `run` methods that explicitly reference their input and output `Connectors`, which have been converted to static fields using techniques from [10]. For non-`Thread` objects, dynamic allocation is supported up to a fixed number of instances, but garbage collection of instances

```
class Heap {
 static Connector c1,c2,c3,c4;
}

class Main {
 static public void main (
       String argv[]) {
  Heap.c1 = new Connector();
  Heap.c2 = new Connector();
  (new Stage1()).start();
  Heap.c3 = new Connector();
  (new Stage2()).start();
  Heap.c4 = new Connector();
  (new Stage3()).start();
  (new Listener()).start();
  for (int i=1; i<10; i++)
   Heap.c1.add(i);
  Heap.c1.stop(); ⋆
 }
}
```

```
class Connector {
 public int queue = -1;
 public synchronized int take() {
  int value;
  while ( queue < 0 )
   try { wait(); }
   catch (InterruptedException ex) {}
  value = queue;
  queue = -1;
  return value;
 }
 public synchronized void add(int o) {
  queue = o;
  notifyAll();
 }
 public synchronized void stop() {
  queue = 0;
  notifyAll();
 }
}
```

```
class Stage1 extends Thread {
 public void run() {
  int tmp = -1;
  while (tmp != 0)
   if ((tmp=Heap.c1.take()) != 0)
    Heap.c2.add(tmp+1);
  Heap.c2.stop();
 } ⋆⋆
}
class Stage2 extends Thread {...}
class Stage3 extends Thread {...}
class Listener extends Thread {
 public void run() {
  int tmp = -1;
  while (tmp != 0)
   if ((tmp=Heap.c4.take()) != 0)
    System.out.println(
      "output is " + tmp);
 }
}
```

Figure 3: Threaded Pipeline in Java

is not supported.

We checked two kinds of properties for this system related to the proper shutdown of pipelined computations. The properties specify (*i*) the eventual shutdown of a pipeline stage in response to a call to `stop` on the pipeline's input `Connector` and (*ii*) that a stage does not prematurely shutdown. For `Stage1` property (*i*) is expressed as a global, response property in LTL as:

$\Box$(`Heap.c1.stop()` $\rightarrow \Diamond$ `Stage`i`.run:return`)

and property (*ii*) is expressed as a global, precedence property in LTL as:

$\Diamond$ `Heap.c1.stop()` $\rightarrow (\neg$ `Stage`i`.run:return` $\mathcal{U}$
(`Heap.c1.stop()` $\wedge \neg$ `Stage`i`.run:return`))

The primitive propositions in these formula are expressed as collections of locations in the source code and are indicated by the ⋆s in Figure 3. The BUI allows users to point-and-click to identify such locations, in Figure 2 the last line of `main` was selected (its line number, 11, appears in the LTL formula).

Table 4 shows data for checking several of the properties described above using Bandera. The table gives the total time required to extract the model from code, to check that model on the property using Spin, the result of the model check, and the number of states searched. The times given are user plus system time, rounded to the nearest second, for Bandera 0.3 and Spin 3.3.5 running on a 450 Mhz Pentium II Xeon running Linux. We used two variations of the program: the code in Figure 3 is the basic version (b) and a defective variant was created by inverting `Stage1`'s loop exit condition (d). The properties are instances of the response (r) and precedence (p) properties from above indexed by the stage number. Three different extractions were tried using no transformations (n), slicing (s) and slicing and abstraction (a). We identify a problem by using the letter for the problem, property and extraction, e.g., "b,r1,n" is the response property for `Stage1` checked on a model extracted from the basic program with no transformations.

The data shows the benefits of property-directed slicing and abstraction in model extraction. We performed a single transformation-less extraction, which essentially transliterates the program to Promela and checks it with Spin. Sliced extractions are driven completely automatically off of the temporal logic specification. Slicing on property (r1) is able to eliminate all of the components shown in the faded font in Figure 3. As the data indicates this yields dramatic performance improvement, but that the improvement is dependent on the property being checked. The response property for `Stage2` requires that the `run` method for that class and the `Heap.c2` variable be included in the slice. Only one abstraction was applied for this program. The `queue` field of the `Connector` object is abstracted to `Signs` which manipulates tokens `Neg`, `Zero`, `Pos` and $\top$. The collapsing of data states with this abstraction further reduces the state space and check times. In fact, for the properties we checked, the models extracted with abstraction do not depend on the number or size of the values `added` to the pipeline in the `main` loop. Since slicing is driven off the propositions in the specification, and not its structure, the models for (b,r1) and (b,p1) are the same although the structure of the property has a small impact on state space and check time. Finally, the extracted models for (d,r1) illustrate that the model extraction process preserves defects in the source code.

In summary, this data illustrates how Bandera can support scaling the application of verification tools. Since the complexity of the algorithms used in Bandera are typically much lower than the complexity of most finite-state verification algorithms, in practice, this should allow properties of significantly larger systems to be verified.

Problem	Extract Time (s)	Check Time (s)	Check Result	States
b,r1,n	24	2674	true	7338120
b,r1,s	13	4	true	3478
b,r1,a	15	4	true	895
b,r2,s	13	56	true	528059
b,r2,a	16	11	true	27519
b,p1,s	13	4	true	2507
b,p1,a	15	4	true	331
d,r1,s	13	3	false	88
d,r1,a	15	2	false	17

Figure 4: Bandera Pipeline Data

Extraction of compact models is crucial for practical application of finite-state verification to software. Concise presentation of diagnostic information provided by verifiers is just as important. For example, the 159 statement long counter-example that is being navigated through in Figure 2 corresponds to a 1780 step counter-example in the underlying Promela model.

Here we reported data for a `Connector` with a capacity of one value implemented as a single integer. We have applied Bandera to versions of this program where the `Connector` is implemented as a circular buffer of fixed size and as an instance of `java.util.Vector`. In both these latter cases, the arrays that store the actual values could be sliced away, leaving only the fields that determine emptiness; these fields appear in the condition used for `waiting` to `take` values from the `Connector`.

5 RELATED WORK

There has been a significant amount of activity in the past years on attempting to provide tool support for the translation of software system descriptions to the input languages of verification tools. There are three major efforts in this area: JavaPathFinder [13], JCAT [4], and Feaver [16].

The first two tools transliterate Java programs to Promela programs. Since Promela is a very rich model description language the semantic gap is not nearly as great as for some other model checker input languages, e.g., SMV. Each of these tools handle a significant portion of Java including dynamic object allocation, object references, exception processing, and inheritance. The weakness of these tools is their inability to significantly compress the Promela program based on the property to be checked, so as to enable tractable model checking for non-trivial programs. This is one of the major design goals of Bandera.

Feaver is a system for extracting Promela programs from annotated C programs for checking with Spin. Feaver allows the user to configure the extraction process by defining pairs of C and Promela code patterns. When the C pattern is detected, the Promela pattern is instantiated and output. In this way, the user can control the abstraction process, but there is no assurance about the information that is encoded in the model since the patterns have no semantics attached to them. Nevertheless, users can debug their patterns over a period of time and the resulting model extraction process has proven effective for production C programs in telecommunications applications.

Huch [18] has built a dedicated model-checker for a subset of Erlang – an untyped higher-order concurrent functional language with asynchronous communication primitives. For state space reduction, he uses a single abstract interpretation that approximates the set of data constructors that flow into expressions at run-time. When one restricts the recursion in programs to a reasonable generalization of tail-recursion, the abstract interpretation is guaranteed to give a finite-state model. Given such a model, properties such as mutual exclusion and absence of deadlock and livelock can be checked. Given the untyped nature of the language, it is not clear to us how would one obtain more effective abstract interpretation by using multiple abstractions tailored to given specifications rather than the single uniform abstraction used by Huch. However, the techniques used to treat recursion might be used for Java. Huch gives some of the same advantages and disadvantages of this monolithic approach that we noted in the introduction. For example, he notes that he must recode model-checking optimizations such as partial-order reductions in his system. However, he argues that working at the level of Erlang instead of *e.g.*, a generated Promela description, will allow easier detection of the cases where such optimizations can be applied.

Several model checking tools have begun to apply traditional compiler optimizations, such as dead code elimination and live-variable analyses, to optimize the checking process [6, 15]. Bandera incorporates a sophisticated infra-structure for implementing such analyses and transformations. We plan to use this infra-structure to enable importation of techniques developed in compiler research to experiment with model extraction for programs with recursive data and methods, e.g., by exploiting shape analyses [2].

6 CONCLUSIONS

We have described the design and implementation of Bandera: a tool for model checking Java source code. Bandera uses a component-based architecture for model extraction designed to maximize scalability, flexibility, and extensibility. We have discussed how several technologies that have been well-studied in software engineering and programming languages research can be adapted and integrated to provide an effective model extraction capability. The current implementation can handle a realistic, albeit limited, class of Java programs and we have illustrated the potential for model extraction to enable finite-state verification of source code.

We are currently working to enrich the features supported by Bandera in anticipation of a general public release of the tool-set in the summer of 2000 (see `http://www.cis.ksu.edu/~santos/bandera` for additional information about Bandera and the current state of the tool-set). This work involves, for example, handling more features of Java (e.g., interfaces, user-thrown exceptions), integrating support for abstracting entire classes, rather than abstracting the individual fields of the class, and supporting additional verifiers (e.g., Stanford's forthcoming SAL model checker and CMU's bounded satisfiability model checker).

By providing automated support for extracting compact finite-state models of source code, tools like Bandera lower the barriers to applying model checking to software. We hope this will facilitate the transfer of this technology from research to practice and provide developers with a very powerful tool for error detection.

ACKNOWLEDGEMENTS

The authors would like to thank David Schmidt and George Avrunin for numerous discussions about the foundations for and design of Bandera. 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.

REFERENCES

[1] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 1996.

[2] J. C. Corbett. Constructing compact models of concurrent Java programs. In M. Young, editor, *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, March 1998.

[3] 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.

[4] 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.

[5] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design*, October 1992.

[6] Y. Dong and C. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE/PSTV'99*, Nov. 1999.

[7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[8] 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 *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, Sept. 1999.

[9] J. Hatcliff, M. B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *LNCS 1490*. Principles of Declarative Programming 10th International Symposium, PLILP'98, Sept. 1998.

[10] J. Hatcliff, M. B. Dwyer, S. Laubach, and N. Muhammad. Specializing configurable systems for finite-state verification. Technical Report 98-4, Kansas State University, Department of Computing and Information Sciences, 1998.

[11] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 2000. to appear.

[12] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using SPIN. In *Proceedings of the 4th International SPIN Workshop*, Nov. 1997.

[13] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999. to appear.

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

[15] G. J. Holzmann. Engineering a model checker : The Gnu i-protocol case study revisited. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.

[16] G. J. Holzmann and M. H. Smith. Software model checking : Extracting verification models from source code. In *Proceedings of FORTE/PSTV'99*, Nov. 1999.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[18] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 261–272, Sept. 1999.

[19] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

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

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 1th International Conference on Automated Deduction (LNCS 607)*, 1992.

[23] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, Nov. 1999.