Tool-supported Program Abstraction for Finite-state Verification

Matthew B. Dwyer, John Hatcliff, Robby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Hongjun Zheng
Kansas State University
Department of Computing and Information Sciences
Manhattan, KS 66506, USA
{dwyer,hatcliff,robbyjo,laubach,pcorina,robby,zheng}@cis.ksu.edu

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

ABSTRACT

Numerous researchers have reported success in reasoning about properties of small programs using finite-state verification techniques. We believe, as do most researchers in this area, that in order to scale those initial successes to realistic programs, aggressive abstraction of program data will be necessary. Furthermore, we believe that to make abstraction-based verification usable by non-experts significant tool support will be required.

In this paper, we describe how several different program analysis and transformation techniques are integrated into the Bandera toolset to provide facilities for abstracting Java programs to produce compact, finite-state models that are amenable to verification, for example via model checking. We illustrate the application of Bandera's abstraction facilities to analyze a realistic multi-threaded Java program.

Keywords

program analysis, model checking, abstract interpretation

1 INTRODUCTION

Finite-state verification techniques, such as model checking, are rekindling interest in program verification. Such techniques exhaustively check a finite-state model of a system for violations of a system requirement formally specified as an assertion or in a temporal logic (e.g., LTL [13]). This approach allows for a very high-level of confidence in system correctness to be achieved essentially automatically—once the model and property specification are constructed, the verification is fully automatic, albeit potentially time-consuming.

There are several obstacles to applying existing model checking [11, 14] directly to the reason about programs; for example, how to efficiently express modern object-oriented language features in the somewhat restrictive verifier input languages. Perhaps the greatest obstacle to scaling finite-state verification technology to reason about realistic programs, however, is the exponential increase in the size of a finite-state model as the number of program components grows. A variety of methods exist for curbing this 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.

In this paper, we describe an approach to user-guided abstraction of programs written in Java; this approach is implemented as part of the Bandera toolset [4]. Our goal is to enable non-experts in program verification to generate abstract program models that are compact enough for tractable verification. While fully-automatic verification is conceptually attractive, we believe that minimal input from users at key points in the abstraction process will provide significantly more flexibility in producing compact program models.

Our approach has been influenced by a large body of work on abstraction in program analysis and computer-aided verification. Specifically, we adopt the framework of abstract interpretation [5], in which an *abstraction* maps the values of a program data type to a set of abstract values and the operations on that data type to a function over abstract values; abstraction of program control is handled in Bandera using program slicing and partial evaluation techniques.

While there has been much foundational work on defining sound abstractions in transition system models, there has been very little work on developing tool support for effectively applying abstract interpretations for model-reduction to large code bases written in modern programming languages like Java. We believe that there are four central issues that need to be addressed in this setting: (*i*) providing facilities for easily *defining* new abstractions, (*ii*) providing tool support and methodologies for *selecting* appropriate abstractions, (*iii*) *generating* abstract programs from concrete ones, and (*iv*) *interpreting* the results of model-checking abstract programs. We use a combination of predicate abstraction [18] and manual techniques for creating the definitions of abstractions and we store these definitions in a library for use in subsequent program verification activities. Users are guided in selecting abstractions based on the property under analysis and analysis of program dependences. We use type inference [15] to determine the appropriate abstractions for all program data, based on the selections made by the user. An abstracted program is generated by compiling abstraction definitions into Java representations, and by systematically replacing concrete operations with calls to abstract operations in the Java abstraction representations. Finally, we make it easier to interpret the results of checking abstracted code by enhancing the JFP model-checker with a facility avoids generating infeasible counterexamples.

Our work focuses on adapting techniques from existing work, developing new techniques where necessary, and integrating these together to provide an abstraction capability that is broadly applicable to programs written in Java and is usable by non-experts. While our tools are Java specific, the underlying abstraction techniques are not; much of our toolset is capable of analyzing any program described in terms of JVM bytecodes. In addition, while much of our discussion focuses on verification of program properties, our tools allow for "unsafe" abstractions to be used; these can provide for additional compaction of finite-state models that are useful for bug finding.

In the next section, we give a brief overview of the Bandera system. Section 3 outlines the underlying principles on which our abstraction techniques are built. Sections 4 through 7 describe Bandera's tool support for defining abstractions, selecting abstractions appropriate for a given property, generating abstract programs, and interpreting the results of model-checking abstract programs. Section 8 discusses the experience of applying these techniques to a Java rendering of the DEOS realtime kernel. Section 9 gives related work and Section 10 concludes.

2 BANDERA

Bandera [4] 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 [11], SMV [14], and JPF [20], to provide state-of-the-art analysis engines for checking program-property correspondence. 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. There are several motivations for targeting multiple model checkers. Tools may have different performance, e.g., SPIN is purported to be an order of magnitude faster than JPF, or may provide a more expressive input language, e.g., JPF treats library code and provides support for recursion and garbage collection directly. By integrating these different tools behind a common interface, Bandera allows the user to exploit their individual strengths.

Bandera consists of five major components:

- *property specification* is supported in Bandera through the use of global properties, e.g., deadlock, and application specific properties e.g., assertions and temporal logic formula. Users define *observations* of the execution state of a Java program as predicates over program locations and data values in the program. Assertions and temporal formula are then defined in terms of those observations.
- *program slicing* automates the elimination of program components that 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 [9] and is based on calculation of the program dependence graph.
- *program abstraction* is discussed below.
- *verifier code generation* transforms the sliced, abstracted program into the input format of a selected model checker. This component is also responsible establishing the correspondence between the states of the input model and the observable states of the program that are referenced in the property.
- *counter-example interpretation* involves the mapping of low-level verifier-specific counter-examples back to the Java source code. Facilities for navigating through the counter-example and displaying the values of both stack and heap allocated data are provided through a debugger-like interface.

To date these capabilities have proven useful in supporting the analysis of small to medium-sized Java programs (100-1500 lines). Case-studies are currently being conducted on several large Java programs (10k-100k lines).

3 DATA ABSTRACTION IN VERIFICATION

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

Linear temporal logic

Model-checking tools often accept state/action sequencing specifications written in linear temporal logic (LTL) [13], computational tree logic (CTL) – a branching-time logic [14], or automata-based formalisms. Bandera hides the differences in various specification languages by using a tool-independent language of temporal specification patterns for rendering temporal properties. To simplify our presentation here, we use LTL for coding temporal properties of programs.

The grammar below gives gives syntax of LTL.

$$\begin{array}{lcl} P & ::= & \top \mid \bot \mid p \\ \psi & ::= & P \mid \neg P \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \\ & & \Box\psi \mid \Diamond\psi \mid \psi_1 \ \mathcal{U} \ \psi_2 \end{array}$$

An LTL specification describes the intended behavior of a system on all possible executions. LTL formulas are built up from primitive propositions p, $\top$ (true), $\bot$ (false), and the usual propositional connectives, along with temporal operators $\Box$, $\Diamond$, $\mathcal{U}$. In this presentation, we consider primitive propositions p to be relational expressions between program variables and constants (e.g., the proposition `(x==0)` holds in states where variable `x` is zero). Requiring formulas to be in negation-normal form (negation is only applied to primitive propositions) simplifies automated reasoning about formulas. For the temporal operators, $\Box\psi$ holds when ψ holds at all points in the future, $\Diamond\psi$ holds when ψ holds at some point in the future, and $\psi_1 \ \mathcal{U} \ \psi_2$ holds when ψ_1 holds at all points up to the first point where ψ_2 holds.

Abstract interpretations
For verification of LTL properties, abstractions are required to preserve properties that are true of all system executions. This class of abstractions can naturally be described as abstract interpretations (AI) [5] over the system's execution semantics. The abstract interpretation 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 overapproximate the set of true executable behaviors of the system, i.e., each true executable behavior is covered by an abstract execution. Thus, when these abstract behaviors are exhaustively compared to an LTL specification and found to be in conformance, we can be sure that the true executable system behaviors conform to the specification.

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

For example, consider an LTL specification containing the primitive proposition `(x==0)`. Deciding this specification at a particular program state does not require complete information about the value of `x` — rather, we only need to know if `x` is zero or non-zero. In this case, an appropriate abstraction for `x` might be the classic *signs* AI which only keeps track of whether an integer value is negative, equal to zero, or positive. In the this case, the abstract domain is the powerset of the set of tokens $T = \{\mathit{neg}, \mathit{zero}, \mathit{pos}\}$. For the abstraction mapping α, we have $\alpha(n) = \{\mathit{neg}\}$ when $n < 0$, $\alpha(n) = \{\mathit{zero}\}$ when $n = 0$, and $\alpha(n) = \{\mathit{pos}\}$ when $n > 0$. Now, one needs abstract versions of each of the basic operations on integers. As an example, here is the definition of the abstract version of the addition operation.

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

Here, the operation is defined to have type $+_{abs} : T \times T \rightarrow \wp(T)$ and can be lifted to $\oplus_{abs} : \wp(T) \times \wp(T) \rightarrow \wp(T)$ by taking $\oplus_{abs}(S_1, S_2) \overset{def}{=} \cup_{t_1 \in S_1, t_2 \in S_2} +_{abs}(t_1, t_2)$. For example, $\oplus_{abs}(\{\mathit{zero}\}, \{\mathit{pos}, \mathit{neg}\}) = \{\mathit{pos}, \mathit{neg}\}$.

Note that the return of multiple tokens in cases such as $+_{abs}(\mathit{neg}, \mathit{pos})$ models the lack of knowledge about specific abstract values. We will see later that this imprecision is interpreted in the model-checker as a non-deterministic choice over the values in the set. Such cases are one source of "extra behaviors" that one gets as the abstract model over-approximates the set of true execution behaviors of the system. The over-approximation is correct or *safe* in the sense that each behavior of $+$ on concrete values is contained in the corresponding abstract behavior, i.e., for integers n_1 and n_2, $\alpha(+(n_1, n_2)) \subseteq \oplus_{abs}(\alpha(n_1), \alpha(n_2))$. Abstract interpretation frameworks are constructed so that if one establishes that the safety property above holds for each basic operator, then the set of complete concrete program execution are safely approximated by the set of abstract executions.

One of the tenets of the Bandera tools is that the basic propositions found in a given temporal specification should drive the selection of AIs as well as other model-reduction techniques such as slicing. In the case of data abstraction, one generally wants to select AIs that are "appropriate" for the specification in the sense that the specification's propositions can be decided based on the domain of the AIs. For example, if the specification contains the proposition `(x==0)`, the signs AI may be appropriate for `x` (because the proposition can be decided for any value of `x` from *neg*, *zero*, *pos*), but it would not be appropriate for a proposition `(x==2)` (because when `x` is *pos*, the proposition cannot be decided).

When abstracting properties, Bandera uses an approach similar to [12]. 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). Thus, this abstract disjunction under-approximates the concrete proposition insuring that the property holds on the original program if it does on the abstract program.

4 DEFINING AND COLLECTING ABSTRACTIONS

Bandera provides a simple declarative specification lan-

```
abstraction Signs abstracts int
begin
 TOKENS = {NEG, ZERO, POS};

 abstract(n)              operator + add
  begin                     begin
   n < 0   -> {NEG};         (NEG , NEG)  -> {NEG} ;
   n == 0  -> {ZERO};        (NEG , ZERO) -> {NEG} ;
   n > 0   -> {POS};         (ZERO, NEG)  -> {NEG} ;
  end                        (ZERO, ZERO) -> {ZERO};
                             (ZERO, POS)  -> {POS} ;
                             (POS , ZERO) -> {POS} ;
                             (POS , POS)  -> {POS} ;
                             (_,_) -> {NEG,ZERO,POS};
                            end
end
```

Figure 1: BASL definition of Signs AI (excerpts)

guage (named Bandera Abstraction Specification Language (BASL)) that allows users to define the three components of an AI described above. Figure 1 illustrates the format of BASL for abstracting base types by showing excerpts of the Signs AI specification. The specification begins with a definition of a set of tokens — the actual abstract domain will be the powerset of the token set. Although one can imagine allowing users to define arbitrary lattices for abstract domains, BASL currently does not provide this capability because we have found powersets of finite token sets to be easy for users to understand and quite effective for verification. Following the token set definition, the user specifies the abstraction function which maps concrete values (in this case, integers) to elements of the abstract domain. After the abstract function, the BASL specification for base types must contain a definition of an abstract operator for each corresponding basic concrete operator. Figure 1 gives the BASL rendering of the abstract operator $+_{abs}$ as defined in Section 3.

We noted in Section 3 that for the AI to be correct, each concrete/abstract operation pair must satisfy the operation safety property. Bandera supports this by allowing the user to supply only the BASL token set and abstract function definitions for integer abstractions, and it *automatically generates* safe abstract operator definitions using the elimination method based on weakest pre-conditions from [2]. Using this approach makes it extremely easy for even novice users to construct new AIs for integers. Given a binary concrete operator *op*, generation of the abstract operator op_{abs} applied to a particular pair of abstract tokens a_1 and a_2 proceeds as follows. The tool starts with the most general definition, i.e., it assumes that $op_{abs}(a_1, a_2)$ can output any of the abstract tokens (this trivially satisfies the safety requirement). Then, for each token in the output, it checks to see (using the theorem prover PVS [16]) if the safety property would still hold if the token is eliminated from the output. An abstract token can be safely eliminated from the output token set if the result of the concrete operation applied to concrete values cannot be abstracted to that abstract value.

For example, consider the derivation of $+_{abs}(neg, neg)$ in the Signs AI. We start by assuming $+_{abs}(neg, neg) = \{neg, zero, pos\}$ and we try to prove the following implications where $neg?(n)$, $zero?(n)$, and $pos?(n)$ are the predicates from the abstraction function associated with the respective abstract token (e.g., $pos?(n)$ holds iff $\alpha(()n) = \{neg\}$).

$$\begin{array}{rcl} neg?(n_1) \wedge neg?(n_2) & \Rightarrow & \neg neg?(+(n_1, n_2)) \\ neg?(n_1) \wedge neg?(n_2) & \Rightarrow & \neg zero?(+(n_1, n_2)) \\ neg?(n_1) \wedge neg?(n_2) & \Rightarrow & \neg pos?(+(n_1, n_2)) \end{array}$$

The theorem prover establishes that the second and third implications are true for any integer values n_1 and n_2. From this, our tool can infer that the output of $+_{abs}(neg, neg)$ should not include *zero* or *pos*, i.e., the output should be $\{neg\}$. Since there are three values in the Signs token set, generating the definition of a binary operator requires that a total of $3^3 = 27$ implications be submitted to PVS. In our experience, these simple proof obligations can always be discharged completely automatically using PVS's "grind" facility.

If one is interested only in bug-finding and not in actual verification (i.e., if one is willing to accept both false negative and false positive error reports), then unsafe abstractions that yield more compact models may be used. For example, when checking a program with an integer variable that appears never to hold a negative value, one might use an AI with a token set $\{zero, pos\}$ (which would be unsafe in general).

Abstracting non-Base Types

BASL also includes formats for specifying AIs for classes and arrays. Class abstractions are defined component-wise: the BASL format allows the user to assign AIs to each field of the class. The current format does not allow the user to specify abstract versions of the class's methods. Instead, abstract versions are derived by transformation described in Section 6. In the BASL array format, the user specifies an integer abstraction for the array index and an abstraction for the component type. For example, consider an application that keeps track of information about k widgets using an array `WidgetInfo wi[k]`. If one wants to verify properties about a particular widget (e.g., the widget in array position 5), an appropriate array abstraction might involve abstracting (a) the index using the token set $\{belowfive, five, abovefive\}$, and (b) the component type `WidgetInfo` using a class abstraction the preserves relevant fields from `WidgetInfo`. In essence, this collapses the concrete array `wi` with k components down to an array `awi` with three components; the intuition is that during abstract interpretation, `awi[belowfive]` will summarize (i.e., union) all the widget information for the concrete components 0 through 4, `awi[five]` will contain information for component 5, and `awi[abovefive]` will summarize all the widget information for the concrete components 6 through k. Manually created abstractions with this structure have been used in earlier work by Dams on verification of air traffic control software [6].

A Library of Abstractions

After creating a BASL specification, the user submits it to an abstraction library manager which compiles one or more representations of the AI (discussed in Section 6) and enters those representations in an abstraction database. During the process of abstraction selection (described in the next section), the user binds one or more variables that are relevant for the property being checked to AIs chosen from the library. A GUI allows browsing of the library which is organized by concrete types: selecting a particular type from a list of concrete types returns a list of possible abstractions for that type. Since they are so widely applied, abstractions for integers are further organized into several different families including the *range*, *set*, *modulo*, and *point* families which we discuss below.

A *range* AI tracks concrete values between lower and upper bounds l and u but abstracts values less than l and greater than u by using a token set of the form $\{belowl, l, ..., u, aboveu\}$. Both the signs AI and the array index AI above are trivial range abstractions where $l = u = 0$ and $l = u = 5$, respectively. A *set* AI can often be used instead of a range AI when no operations other than equality are performed (e.g., when integers are used to simulate an enumerated type). For example, a set AI that tracks the concrete values 3 and 5 would have the token set $\{three, five, other\}$. A *modulo-k* AI merges all integers that have the same remainder when divided by k. The even-odd abstraction with token set $\{even, odd\}$ is a modulo-2 abstraction. Finally, the token set for the *point* AI includes a single token *unknown.* The point abstraction function maps all concrete values to this single value; this the effect of throwing away all information about the data domain. Each concrete base type has a point AI associated with it, and these are widely used in situations where a particular variable's value has no significant impact on the property be verified.

5 SELECTING ABSTRACTIONS

Bandera's support for definition and reuse of abstractions as well as the ability to automatically integrate abstraction definitions into the source text (discussed in the next section) enhances confidence in the correctness of the resulting abstract program. While correct abstraction is necessary, it is not sufficient for a practical software model checking technology. For a given program and property, it will often be necessary to customize the abstraction of program data in order to enable tractable model checking.

It is possible to define a notion of optimal abstraction with respect to a program-property pair, i.e., where any finer abstraction adds irrelevant information and any coarser abstraction introduces infeasible behaviors. For example, a program in which the only conditionals that `x` appears in are of the form `(x==0)` or `(x>0)` would optimally use a signs abstraction for `x`. One could use a range(0,1) abstraction where the *pos* value is decomposed into *one* and *aboveone.* Both of the new values yield true for an `(x>0)` test and false for an `(x==0)` test just as the *pos* value did and hence they provide no new information. Alternately, one could use a set(0) abstraction which collapses *neg* and *pos* values to a *nonzero* token. This over-abstraction with respect to proposition `(x>0)` runs the risk of introducing infeasible paths on which a positive value appearing at an `(x>0)` conditional in the concrete program yields a transition to the false branch in the abstract program.

As illustrated in the above example, the fundamental tension in selecting abstractions is between a desire to compress the state space, via reduction of data domains, and to preserve data properties that are relevant to the property being checked. We provide tool support that allows the user to focus attention on local variables and class fields (referred to simply as variables in the sequel) and the properties of the values stored in those variables that are most relevant to the property being checked. In addition, we allow users to define only the abstractions they view as essential and then the tools calculate abstractions for the rest of the program data.

Program Dependence-based Selection Methodology

While the user is free to make any abstraction selection they desire, we believe that selections should be driven by relationships between the data and control points mentioned in the property and the data and control points in the program that can influence their execution. A methodology that exploits such relationships was introduced in [7] and consists of four steps:

I. Start with point AI: Initially all variables are abstracted to point.

II. Identify variables referenced in the property: The propositions in the property to be checked may refer to variables (e.g., a proposition `(x>0)`). As discussed in the previous sections, those variables must be abstracted in a way that preserves the ability to decide the proposition.

III. Select controlling variables: In addition to variables mentioned explicitly in the specification, consider variables on which they are control and data dependent. Conditional expressions that reference those controlling variables suggest additional variables that should be abstracted.

IV. Select variables with broadest impact: When confronted with multiple controlling variables to abstract, select the ones that appear most often in a conditional.

This methodology is supported in Bandera through the use of two program analysis techniques : calculation and browsing of the program dependence graph and abstract type inference.

Leveraging Program Dependence Information

Bandera includes a slicing component [10] that calculates the program dependence graph (PDG) for the given Java program. Slicing is driven by the propositions in the property to be checked (e.g., for a proposition `(x>0)` all definitions of `x` would be included in the slice criterion). This automates the second step of the methodology described above as the variables referenced in the property's propositions are linked to the corresponding program components. One consequence of slicing is that all program variables that can not influence

```
int x,y,z;
boolean even;
...
if ((y % 2) == 0) { // conditional 1
  even = true;
}
...
if (even && z>0) {  // conditional 2
  x = x + 1;
}
...
```

Figure 2: Abstraction Selection Example

the truth or falsity of the propositions will be eliminated.

Bandera provides the ability to visualize, navigate, and query the program dependence graph. The basic functionality of this approach was inspired by GrammaTech's CodeSurfer [8], but has been adapted for multi-threaded Java programs and to serve the purpose of abstraction selection.

The visualization and navigation capabilities allow the user to designate a statement of interest in the program and then selectively follow predecessor links that correspond to different forms of dependence including data, control, and synchronization specific dependences [9]. In this way, once a user has identified a variable that is a candidate for abstraction they can browse the PDG for variables that influence the candidate's value. Querying the PDG is performed by specifying a group of program statements that will form the root of a search in the PDG. Queries will search upward in the PDG to designated target nodes along whatever set of dependence edges the user chooses. The results of a query are presented as a set of paths in the PDG that the user can navigate.

Step three of the methodology is supported by a PDG query that locates all conditionals that influence the root of the search through data and control dependences. Consider the program fragment in Figure 2 where we are interested in a property that refers to proposition `(x>0)`. In this case, all assignments to `x` would be included in the roots of the search. The result will be a set of paths through the PDG that connect a sequence of conditionals and end in the selected program statement (e.g., conditional 1, conditional 2, `x = x + 1;`). From these paths it is possible to identify variables that appear in conditionals that directly control the execution of the statement (e.g., the definition of `x` is control dependent on the tests of `even` and `z` in conditional 2) as well as those that that indirectly control the execution of the statement (e.g., the definition of `even` that reaches conditional 2 is control dependent on the test of `y` in conditional 1). Furthermore, the query results give the boolean expressions in the conditionals that suggest which abstraction should be chosen for the variables involved (e.g., modulo-2 for `y`, signs for `z`, and even remains unabstracted). These results are easily processed to determine the most frequently occurring variables in support of prioritizing abstraction selection based on impact as outlined in step four of the methodology.

Checking and Propagating Abstract Types

PDG-based tool support allows a user to identify a small set of variables that strongly influence the behavior of the program relative to the property being checked and to select abstractions for them. Bandera analyzes those selections to determine wether they are in conflict, i.e., two interacting variables are abstracted in incompatible ways, and to determine the abstract types for the rest of the variables in the program. This analysis is performed by executing a type inference algorithm that is adapted to calculate abstract types for all variables, fields, and expressions in the program. The next section, describes how type inference results are used in compiling the abstract operator definitions into the program text. Here we focus on how the results of type inference can be used to refine the selection of abstractions in the program.

Abstraction selections conflict when two abstract values appear as operands in an expression and there is no meaningful way to transfer information between those values. For example, if the program sketched in Figure 2 had an assignment `y = z;` it would be flagged as an abstract type conflict. Since it would be unclear how to convert a *pos* value for `z` to an *even* or *odd* value for `y`. Similarly, if there were another variable `v` and with assignments from `y` and `z` in the program, it would be unclear wether to abstract `v` with signs or modulo-2. While it is possible to introduce coercions into the program, to resolve the first problem, and to prioritize abstractions, to resolve the second, we believe that is important to notify the user that abstraction conflicts exist and to give them the option of adjusting their abstraction selections.

As suggested by the example above, with `v`, type inference will calculate the abstract type of every program variable that is data dependent on variables for which the user has made explicit abstraction selections. This process also identifies variables whose abstractions are independent of the explicit selections. Such variables arise because they influence the program through control or synchronization dependences, rather than data dependences, and are consequently preserved in the sliced program. The user has three options in abstracting such variables: abstract them to the point to minimize the state space of the generated system, abstract them to their concrete type to maximize the precision of the extracted model, or choose abstractions from the library. The first of these options represents step one in the methodology described above; the latter options provide extra flexibility in selecting abstractions.

6 GENERATING AN ABSTRACT PROGRAM

Generating an abstract program involves two separate steps. First, given a selection of AIs for a program's data components, the BASL specification for each selected AI is retrieved from the abstraction library and compiled into a Java class that implements the AI's abstraction function and abstract operations. Second, the given concrete Java program is traversed, and concrete literals and operations are replaced with calls to classes from the first step that implement the corresponding abstract literals and operations.

Compiling BASL specifications to Java classes

```
public class Signs {
 public static final int NEG  = 0;  // bit-mask 1
 public static final int ZERO = 1;  // bit-mask 2
 public static final int POS  = 2;  // bit-mask 4

 public static int abstract(int n) {
  if (n < 0) return NEG;
  if (n == 0) return ZERO;
  if (n > 0) return POS;
 }

 public static int add(int arg1, int arg2) {
  if (arg1==NEG  && arg2==NEG)  return NEG;
  if (arg1==NEG  && arg2==ZERO) return NEG;
  if (arg1==ZERO && arg2==NEG)  return NEG;
  if (arg1==ZERO && arg2==ZERO) return ZERO;
  if (arg1==ZERO && arg2==POS)  return POS;
  if (arg1==POS  && arg2==ZERO) return POS;
  if (arg1==POS  && arg2==POS)  return POS;
  return Bandera.choose(7);
}
```

Figure 3: Compilation of BASL Signs AI (excerpts)

Figure 3 shows excerpts of Java representation of the BASL signs specification in Figure 1. Abstract tokens are implemented as integer values, and the abstraction function and operations have straightforward implementations as Java methods. The most noteworthy aspect of the implementation is the representation of set values. Recall from Section 3 that when arguments to an abstract operator do not provide enough information to produce a single token as a result, a set of two or more tokens (representing multiple possible outputs) is returned. Instead of representing such sets directly (e.g., as a bit vector), it is common practice in model-checking and abstraction interpretation to have the operator return a single value non-deterministically chosen from the set of possible return values. This is valid when the meaning of a particular program is taken to be the collection of all possible traces or executions of the program. In Figure 3, the `Bandera.choose(bits)` method denotes a non-deterministic choice between the token values encoded in the bit-vector `bits`. Specifically, `Bandera.choose(7)` denotes a non-deterministic choice between the tokens `NEG`, `ZERO`, and `POS`. The `Bandera.choose` method has no defined concrete execution semantics. Instead, when Bandera compiles the abstracted program down to the input of given a model-checker, it is implemented in terms of the model-checker's built-in constructs for expressing non-deterministic choice.

Replacing concrete operators

Traversing a given concrete program and replacing each operation with a call to a corresponding abstract version is relatively straightforward. The only challenge lies in resolving *which* abstract version of an operation should be used when multiple AI's are selected for a program. This problem is solved by the abstract type inference process described in the previous section: in addition to propagating abstract type information to each of the program variables, type inference also attaches abstract type information to each node in the program's syntax tree. For example, consider the code fragment `(x + y) + 2` where the user selected variable `x` to have type Signs and `y` was not selected for abstraction. This code fragment will be transformed into:

```
Signs.add(Signs.add(x, Signs.abstract(y)),
          Signs.POS);
```

For the innermost concrete `+` operation, the user selection of signs for `x` forces the abstract version of `+` to be `Signs.add`. Assuming no other contexts force `y` to be abstracted, `y` will hold a concrete value, and thus a coercion (`Signs.abstract`) is inserted that converts `y`'s concrete value to a signs abstract value. For the outermost `+`, since the left argument has an abstract type of signs, the constant `2` in the right argument is "coerced" at translation time to a signs abstract constant.

Abstract type inference is implemented by solving a system of type constraints and has additional features than cannot be fully illustrated here. For example, to provide the capability for coercions to resolve conflicts as described in Section 5, it allows the user to define a lattice of abstract types ordered by user-defined coercions (*i.e.*, an abstract type T_1 lies below another type T_2 if a coercion is defined from T_1 to T_2). Such lattices for integers generally have a bottom type of concrete integers (these can be coerced to integer abstraction) and the *point* abstraction from Section 3. The lattice structure makes it possible to define a notion of *best (most precise) abstract typing* – this is the typing returned by the inference algorithm.

7 ABSTRACT MODEL CHECK RESULTS

In general, our abstraction technique computes an over-approximation of the original program. Thus, when a specification is true for the abstracted program, it will also be true for the concrete program. However, if the specification is false for the abstracted program, the counterexample may be the result of some behavior in the abstracted program which is not present in the original program.

We implemented a simple technique that helps in interpretation of results in the presence of abstraction and refinment guided by counterexamples. The approach exploits the following theorem from [19]: *Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.*

We enhanced the JPF model checker with an option to look only at paths that do not refer to instructions that introduce nondeterminism (i.e. *choose*). When the next instruction introduces nondeterminism, the search algorithm of the model checker backtracks. The theorem ensures that the paths that are free of nondeterminism indeed correspond to paths in the concrete program. It follows that if a counterexample is reported then it represents a real counterexample (i.e. it can be matched by a computation in the concrete program).

Consider an abstracted program (whose state space is sketched in figure 4) and the invariant property $\Box p$. White circles represent states where predicate p holds, while black circles represent states where p is false. Dashed lines represent transitions that refer to *choose*, while solid lines refer to instructions other than *choose*. Model checking on choose-

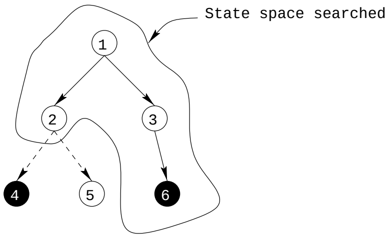

Figure 4: Model checking on choose-free paths

free paths will report only the error path 1-3-6, although path 1-2-4 leads to a state that invalidates the property (and it may correspond to an execution in the concrete program).

We note that our technique could be implemented in any model checker, but the design of JPF made this modification particularly easy. JPF is essentially a special-purpose JVM that interprets each byte code in the compiled version of a Java program. Since *choose* operations are represented as static method calls, trapping and processing those operations specially only required modification of the code for the static method invocation byte code.

Our methodology of model checking and refinement is as follows. The input (concrete) program and the specification are abstracted using abstractions from a library, and verified using 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, the generated error trace has to be analyzed to see if it does not correspond to some spurious behavior introduced by abstractions. Instead of trying to match the error trace to a concrete computation, we re-run the model checker to search only choose-free paths in the model. If the model checker finds an error trace then a guaranteed feasible counterexample is reported. Otherwise, the abstractions are too coarse and the counterexample from the first model check is used to guide the (re)selection of abstractions.

8 AN EXAMPLE

We have applied our abstraction tools to a variety of small Java programs. In this section, we describe how our tools enabled analysis of a complex property of a medium-sized multi-threaded Java program.

The Honeywell DEOS Operating System

The DEOS system is a micro-kernel based operating system designed to provide both time and space partitioning for applications running on it. This system has been the topic of several recent software model checking experiments [17, 20, 21] and we chose it partly as a means of comparing results on an identical problem, and also as a significant test case for the feasibility of our methods.

As described in [20], the system consists of more than 1000 lines of C++ code. To apply existing software model checking tools the C++ code was translated to Java; this was nearly a direct rewriting of the C++ code since it was written in a ”safe” style that avoided pointer arithmetic. To analyze properties of this Java version of 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). The DEOS code was known to have an extremely subtle bug related to time partitioning. That property is ”that all application process are guaranteed to be scheduled for their budgeted time during a scheduling unit”. While it is possible to encode this property as an LTL formula, for this example the property 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.

The Java implementation of DEOS and its environment code constitutes a non-trivial multi-threaded Java application; it measures 1443 lines of code, spread over 20 classes, with a total of 91 methods, 41 instance fields, and 51 static fields. The program consists of 6 threads at run-time. Model checking of the system is infeasible given current hardware limitations; it is unclear exactly how large the systems state space is, but we know that it exceeds four gigabytes. Thus, some form of abstraction is needed to reduce the state space to a tractable size. The authors of [21] describe the application of predicate abstraction to a version of the DEOS kernel written in Promela, SPIN’s [11] modeling language, and tool support for applying predicate abstraction to the Java version is under development. We describe how Bandera’s type-based abstraction can be applied to the Java DEOS kernel.

Abstraction Selection and Compilation

The first step is to apply program slicing to eliminate irrelevant program components. The Java DEOS kernel is derived from a version of the C++ code that was already sliced, by hand, to make inspection of the code easier. Consequently, when we slice the program with the property violation assertion as the criterion there is very little reduction. This process does, however, calculate the PDG which is used to identify candidate variables and fields for abstraction.

Querying the PDG for potential candidates resulted in 42 influencing paths being identified; the longest such path had 5 control dependence edges, which identify conditionals of interest, as well as several data dependences. It is important to note that many of the 42 paths share some common subpaths, e.g., the path of length 5 overlaps with its prefixes of length 4, 3, 2 and 1 control dependence, so the quantity of paths could be reduced by further processing. In total, the query results identified 29 different conditional expressions spread across 16 methods that influence the assertion statement; 32 different local variables, instance or static fields appeared in those conditionals.

Within an hour we were able to analyze this information and identify one conditional that appeared at the root of multiple paths. Figure 5 shows that conditional (i.e.,

```
class Thread {
 public void startThread(...) {
   ...
  itsLastExecution  = itsPeriodicEvent.currentPeriod();
  itsLastCompletion = itsPeriodicEvent.currentPeriod();
 }
 public void startChargingCPUTime() {
  int cp = itsPeriodicEvent.currentPeriod();
  ...
  if (cp==itsLastExecution) {
  ...
 }
 int itsLastExecution, itsLastCompletion;
 ...
}

class StartOfPeriodEvent {
 public void pulseEvent(...) {
 countDown = countDown - 1;
 if ( countDown == 0 ) {
  itsPeriodId = (itsPeriodId + 1);
 ...
 }
 public int currentPeriod() {
  return itsPeriodId;
 }
 private StartOfPeriodEvent(...){
  ...
  countDown = 1;
  itsPeriodId = 0;
 }
 int  itsPeriodId, countDown;
 ...
}
```

Figure 5: Fragments of Java DEOS Code

`cp==itsLastExecution`). Tracing the data dependences from that conditional through the local variable `cp`, instance field `itsLastExecution`, and the method calls to `itsPeriodicEvent.currentPeriod()`, we found that the `itsPeriodId` instance field of the `Thread` class is the source of the values used in that conditional. Navigating the data dependences for `itsPeriodId` shows that there are only two definitions of this field: an assignment to zero and an increment. This means that the variable is effectively unbounded and that abstracting it to a finite domain has the potential to significantly reduce the state space of the program. We note here that the application of predicate abstraction to the Promela version of DEOS described in [21] identifies this same variable as being problematic. Their analysis exploits some rather deep insights into the design and implementation patterns used to efficiently implement certain features of the DEOS system. In contrast, our analysis is less application specific relying only on analysis of program syntax and data dependences.

Since the conditional involves an equality test we looked at the assignment statements to help identify useful abstractions for `itsPeriodId`. We chose the signs abstraction which allows the initial value, 0, to be distinguished from all subsequent incremented values. It should be noted that this is an extremely weak abstraction for the purpose of this conditional since any *pos* values flowing into the test may be equal or not; a non-deterministic choice will be used to model the conditional in that case.

Type inference was initialized with a single type binding of signs to `itsPeriodId`. With a single setting, type inference will find no conflicts and its purpose is to propagate abstract type information throughout the program. For the Java DEOS code type inference determines that the local `cp` and instance fields `itsLastCompletion` and `itsLastExecution` in class `Thread` must also be signs abstrated. The rest of the program's fields and variables are free to be bound to any desired type; we chose to leave them as their concrete types. We note that the use of predicate abstraction in [21] did not treat the field `itsLastCompletion`; this was due to the fact that abstraction was performed by hand. In that case, overlooking `itsLastCompletion` did not affect the ability to detect the time partitioning related defect; it does, however, emphasize the need for automated support in program abstraction.

Abstract Model Checks

Once abstract types were inferred, Bandera generated the abstract program which is very similar to the original program except that all operations applied to signs abstracted fields and variables are implemented with the abstract operations (e.g., for `+`, `==`).

The abstract Java program was fed to JPF which finds a counter-example; we could just as easily have used another model checker such as SPIN. At this point we suspect that the tools have found the time partitioning defect. It may be, however, that the abstractions have introduce behaviors in the abstract program that are infeasible in the original program and that such a behavior violates the time partitioning property. Since the counter-example consists of a sequence of 464 statements, its analysis would be time consuming and we choose to rerun JPF using the option to search for feasible counter-examples. It finds one, which is 318 steps long, that does illustrate the defect.

We note that for this non-trivial Java program all of the different tool components executed in under a minute, whereas model checking the original program never completes. This is strong evidence that tool-supported abstraction can significantly expand the size and complexity of programs for which model checking is practical.

9 RELATED AND FUTURE WORK

We have discussed foundational and program-oriented abstraction related work in the body of the paper. Here we link our approach to work on transition-system based techniques from the verification community.

Our strategy of applying abstractions in verification is similar to [3], where explicit abstraction mappings over the domains of the program variables are used for generating an abstract model. While in [3], the user has to provide the abstraction mapping, together with the abstract operations, we infer them automatically from a given set of predicates using predicate abstraction [18]. Unlike the approach described in [18], we use predicates to abstract operator definitions rather than complete transition systems. Predicate abstraction has been adapted to program code [21] and we view the question of how to combine our type-based approach with predicate-based approaches for program abstraction as an interesting research topic. One direction we are pursuing is to maintain multiple abstract variables for a given concrete variable. We have already explored extensions to BASL to accommodate this.

Our approach to interpreting counter-examples is efficient, but it can miss counter-examples on which non-determinism occurs. Other approaches, such as those based on symbolic execution [1] and theorem proving [2], are more expensive but more complete alternatives. We plan on pursuing comparisons between these approaches to understand the cost and precision tradeoffs.

10 CONCLUSIONS

We described a collection of program analyses that aid users in abstracting Java programs to reduce the cost of reasoning about program properties using model checking techniques. We presented an overview of the Bandera toolset's abstraction capabilities and discussed how they enabled checking of properties of a medium-size concurrent Java program.

One contribution of this work is the development of structured and navigable visualizations for exploiting intermediate results of program transformation components to guide the user in deciding how to abstract a program. In addition, we adapted several techniques from abstraction of transition system models for use in abstracting program source code. Bandera provides two ways to encode abstractions into the program. The first encodes abstract operator definitions directly into a model checker's input and the second, described in this paper, transforms the program's source code to incorporate definitions of abstract operators. This allows our program abstraction approach to be used by any tool that operates on Java code. We have implemented and done preliminary experiments with a method for producing *guaranteed feasible* counter-examples in model checks of abstract programs. The main contribution of our work, however, is the integration of these different techniques into a coherent program abstraction toolset that has the ability to greatly extend the range of programs to which model checking techniques can be effectively applied.

ACKNOWLEDGEMENTS

The authors would like to thank the the members of the Bandera group for numerous helpful discussions. This work was supported in part by NSF under grants CCR-9703094, CCR-9708184 and CCR-9901605, by NASA under grant NAG-02-1209, by Sun Microsystems under grant EDUD-7824-00130-US, and was performed for the Formal Verification of Integrated Modular Avionics Software Cooperative Agreement, NCC-1-399, sponsored by Honeywell Technology Center and NASA Langley Research Center.

REFERENCES

[1] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[2] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10th International Conference on Computer-Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, June 1998.

[3] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.

[4] 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.

[5] 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.

[6] D. Dams. Personal communication. Apr. 2000.

[7] M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 1998.

[8] GrammaTech. CodeSurfer : User Guide and Technical Reference. `http://www.grammatech.com/csurf-doc/manual.html`, 2000.

[9] 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.

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

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

[12] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. *Lecture Notes in Computer Science*, 1450, 1998.

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

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

[15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1998.

[16] 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.

[17] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos real-time scheduling kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[18] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Scienece*, pages 72–83, June 1997.

[19] H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, Lecture Notes in Computer Science, 2000.

[20] W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Sept. 2000. to appear.

[21] W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, Aug. 2000. to appear.