# Foundations of the Bandera Abstraction Tools*

John Hatcliff[1], Matthew B. Dwyer[1], Corina S. Păsăreanu[2], and Robby[1]

[1] Department of Computing and Information Sciences, Kansas State University ***
[2] Kestrel Technology, NASA Ames Research Center †

**Abstract.** Current research is demonstrating that model-checking and other forms of automated finite-state verification can be effective for checking properties of software systems. Due to the exponential costs associated with model-checking, multiple forms of abstraction are often necessary to obtain system models that are tractable for automated checking.

The Bandera Tool Set provides multiple forms of automated support for compiling concurrent Java software systems to models that can be supplied to several different model-checking tools. In this paper, we describe the foundations of Bandera's data abstraction mechanism which is used to reduce the cardinality (and the program's state-space) of data domains in software to be model-checked. From a technical standpoint, the form of data abstraction used in Bandera is simple, and it is based on classical presentations of abstract interpretation. We describe the mechanisms that Bandera provides for declaring abstractions, for attaching abstractions to programs, and for generating abstracted programs and properties. The contributions of this work are the design and implementation of various forms of tool support required for effective application of data abstraction to software components written in a programming language like Java which has a rich set of linguistic features.

## 1 Introduction

Current research is demonstrating that model-checking and other techniques for automated finite-state verification can be applied directly to software in written in widely used programming languages like C and Java [1, 3, 4, 15, 28]. Although they may vary substantially in the specifics, in essence each of these techniques

* This work was supported in part by NSF under grants CCR-9703094, CCR-9708184, CCR-9896354, and CCR-9901605, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, by DARPA/IXO's PCES program through AFRL Contract F33615-00-C-3044, by NASA under grant NAG-02-1209, by Intel Corporation under grant 11462, 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.

*** 234 Nichols Hall, Manhattan KS, 66506, USA.
{hatcliff,dwyer,robby}@cis.ksu.edu
† Moffet Field, CA, 94035-1000, USA. pcorina@email.arc.nasa.gov

exhaustively checks a finite-state model of a system for violations of a system requirement formally specified by some assertion language or in some temporal logic (e.g., LTL [22]). Finite-state verification is attractive because it automatically and exhaustively checks all behaviors captured in the system model against the given requirement. A weakness of this approach is that it is computationally very expensive (especially for concurrent systems) due to the huge number of system states, and this makes it difficult to scale the approach to realistic systems.

The widespread adoption of Java with its built-in concurrency constructs and emphasis on event-based programming has led to a state of affairs where correct construction of multi-threaded reactive systems, which was previously a domain for experienced programmers, must be tackled by novice programmers. Moreover, Java is being used increasingly in embedded systems where it is more important to detect and remove errors before initial deployment. Thus, there is substantial motivation for building model-checking tools to assess the effectiveness of applying software model-checking to Java. Central to any such tool should be *abstraction mechanisms* that are employed to reduce the number of states encountered during the exploration of software models.

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin [16], dSpin [9], and JPF [3]). Both program slicing and user extensible data abstraction components are applied to form abstract program models customized the to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

Various forms of predicate abstraction [1, 3, 28] and data abstraction [8, 10, 27] have been used in model-checking, and there is a wide body of literature on these techniques. Given this rich resource upon which to build our abstraction facilities in Bandera, our particular choice of abstraction techniques was influenced by multiple requirements outlined below.

**(I)** *The abstraction facilities should be easy to understand and apply by software engineers with little technical background in formal semantics:* This is a basic requirement if Bandera is to be applied effectively by a broad spectrum of users.

**(II)** *The abstraction capabilities should integrate well with the dynamic features found in Java:* In Java programs, features such as dynamic thread/object creation and traversal of heap-allocated data are ubiquitous. Existing work on predicate abstraction and automated refinement [1, 28] has focused on software that relies on integer computation and pointers that are restricted to referencing static data. Automated counter-example-driven abstraction refinement for programs with dynamically allocated data/threads is still an open research area.

**(III)** *The abstraction process should scale to realistic software systems:* Methods for selecting abstractions and methods for constructing abstract programs should not degrade dramatically as the size of programs considered increases.

**(IV)** *The abstraction process should be decoupled from model-checker engines:* Bandera encapsulates existing model-checkers and does not modify their functionality. Thus, any abstraction process implemented by Bandera needs to be accomplished outside of any encapsulated model-checkers.

Bandera addresses Requirement I by providing a easy-to-use and flexible mechanism for defining abstractions. Complex domain structures are avoided by embracing only domains that can be represented as powersets of finite sets of "abstract tokens". These domains (along with appropriate abstract versions of operators) form *abstract types* that users can associate with program variables. An abstract type inference takes as input an abstract type environment that gives abstraction selections for a small set of relevant program variables and then propagates this information through the entire program. This significantly reduces the amount of effort required by users to specify how a system should be abstracted.

Bandera addresses Requirement II by taking advantage of the fact that the type-based data abstraction process described above can be applied in a component-wise manner to fields of different classes. This allows components of heap-allocated data to be abstracted without considering more complicated forms of, for example, shape analysis or predicate abstraction.

Bandera addresses Requirement III by precomputing definitions of abstract operations (using a theorem prover) and then compiling those definitions into the Java source code to form an abstract program. Thus, the repeated (expensive) calls to a theorem prover used in predicate abstraction approaches are not needed during abstract program construction or the model-checking process.

Bandera addresses Requirement IV by transforming the Java source code of program such that concrete operations on types to be abstracted are replaced by corresponding calls to generated abstraction operations. Since the abstraction process takes places at the source level, existing model-checking engines do not have to be modified to incorporate abstraction mechanisms.

In summary, we have arrived at the form of abstraction used in Bandera by balancing a number of competing goals. Bandera's abstraction facilities are less automatic than the automated predicate abstraction and refinement techniques of other tools, but they are much less expensive and can be used immediately in the presence of concurrency and dynamically allocated data without any technical extensions. Moreover, we have found the facilities to be effective in reasoning about a variety of real Java systems.

The tool-oriented aspects of Bandera's abstraction facilities have been described in detail elsewhere [10]. In this paper, we focus on technical aspects of the facilities. Section 2 reviews the methodology that users typically follow when applying Bandera. Section 3 describes the program/property syntax and semantics for a simple flowchart language which we will use to present technical aspects of Bandera's abstraction facilities. Section 4 presents a formal view of Bandera's abstraction definitions and how decision procedures are used to automatically construct definitions of abstract operators and tests. Section 5 describes Bandera's abstract type inference mechanism that is used to bind abstraction declarations to program components. Section 6 outlines how Bandera uses the abstraction bindings calculated above to generate an abstracted program. Section 7 describes how chosen program abstractions should also give rise to appropriate abstractions of the property being checked. Section 8 summarizes related work, and Section 9 concludes.

## 2 The Bandera Abstraction Facilities

Bandera is designed to support a semi-automated abstraction methodology. The goal of this methodology is to minimize the amount of information that user's need to supply to perform an abstract model check of a given property on a Java system. The main steps in applying Bandera are:

1. Identify the portion of the system to be analyzed;
2. Formalize the property in the Bandera Specification Language;
3. Compile the system and property;
4. Define and select the abstractions to be used;
5. Generate an abstracted system model and property;
6. Execute a model check on the abstracted system model; and
7. Analyze any generated counter-examples for feasibility.

Bandera provides various forms of automated support for Step 1. Once the system has been closed, in Step 2 the user formalizes properties to be checked using the Bandera Specification Language (BSL) [5] — a language for defining propositions/predicates on a program's control points and data state and for expressing temporal relationships between declared propositions. In Step 3, Bandera compiles a closed Java unit and the property specification down to a three-address intermediate form called Jimple – part of the Soot [29] Java compiler infrastructure. After transformations in Steps 4 and 5 and other transformations such as slicing have completed, Jimple is transformed to a lower-level intermediate representation called the Bandera Intermediate Representation (BIR). A detailed presentation of BIR's semantics and the translation of Java to BIR and BIR to Promela, the Spin model-checker's input language, is available in [18].

In the remainder of this section, we provide a brief overview of Steps 4–7. We emphasize the toolset components related to the definition of abstractions and the encoding of abstract system models and properties.

### 2.1 Defining and Selecting Abstractions

Users select abstractions by considering the semantics of predicates appearing in the property to be checked and the program expressions that can exert either

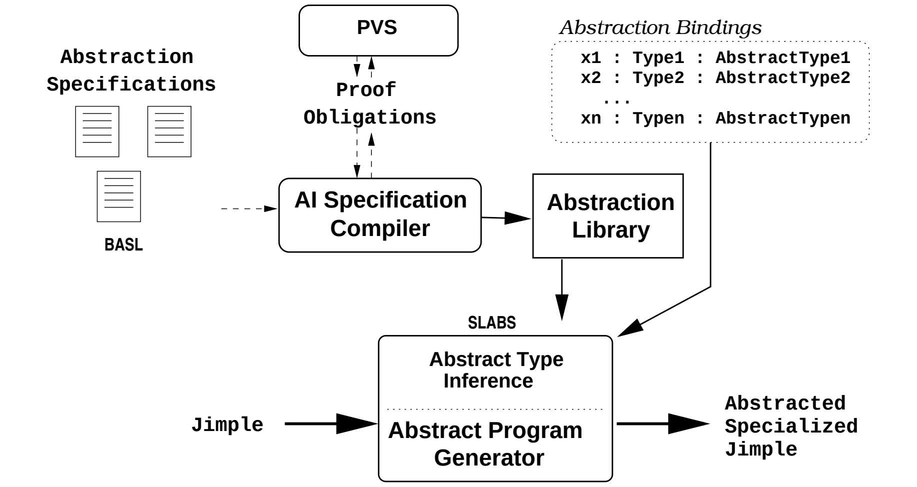


**Fig. 1.** Architecture of Bandera's Abstraction Facilities

control or date influences on those predicates. In [10] we describe tool support in Bandera that allows users to query and explore the program dependence graph that is generated by using a property's predicates to derive a slicing criterion. While strategies for exploiting this tool support to identify the program variables that should be abstracted and the semantics that should be preserved by such an abstraction are relevant for users of Bandera, in this paper, we assume that such a determination has been made. The user carries out the balance of the abstraction process by interacting with the Source Level Abstraction (SLABS) Bandera tool components displayed in Figure 1.

Bandera includes an *abstraction library* containing definitions of common abstractions for Java base types from which users can select. If necessary, the user specifies new abstractions using the rule-based Bandera Abstraction Specification Language (BASL). For Java base types, the user need only define an abstract domain, and abstract versions of concrete operations are generated automatically using the decision procedures of the PVS theorem prover [24]. The abstraction definitions are then compiled to a Java representation and added to the library.

The user declares how program components should be abstracted by binding class fields to entries from the abstraction library. It is usually only necessary to attach abstraction to a relatively few variables since the toolset contains an abstract type-inference phase that automatically propagates the abstract type information to remaining program components. When abstract type inference is complete, the concrete Jimple program representation is transformed to abstract Jimple by replacing concrete constants and operations by abstract tokens and operations drawn from the compiled abstraction representations in the abstraction library. We describe these steps in more detail in the subsections below.

```
abstraction EvenOdd abstracts int
  begin
   TOKENS = {EVEN, ODD};

   abstract(n)               operator * mul          operator > gt
    begin                     begin                    begin
     n % 2 == 0 -> {EVEN};    (ODD, ODD) -> {ODD} ;     (_, _) -> {true, false}
     n % 2 == 1 -> {ODD};     (_, _) -> {EVEN} ;       end
   end                       end
  end
```

**Fig. 2.** BASL definition of EvenOdd AI (excerpts)

**Defining Abstractions** Bandera provides BASL – a simple declarative specification language that allows users to define the three components of an AI described above.

Figure 2 illustrates the format of BASL for abstracting base types by showing excerpts of the even-odd 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 abstraction function, the BASL specification for base types must contain a definition of an abstract operator for each corresponding basic concrete operator.

Abstract operator definitions can be *generated automatically* from the BASL token set and abstraction function definitions for integer abstractions 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 – which trivially satisfies the safety requirement). Then, for each token in the output, it checks to see (using the theorem prover PVS [24]) 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.

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. BASL's array format allow specification of an integer abstraction for the array index and an abstraction for the component type [10].

**A Library of Abstractions** Since they are so widely applied, abstractions for integers are organized into several different families including the *concrete* (or *identity*), *range*, *set*, *modulo* and *point* families which we discuss below.

A *concrete* AI (also known as an *identity* AI) performs no abstraction at all, but rather preserves all concrete values and uses the original concrete operations on these. 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 $\{belowl, l, ..., u, aboveu\}$; an abstraction that preserves the *sign* of values is a range-$(0, 0)$ abstraction. 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 EvenOdd 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 has the effect of throwing away all information about the data domain.

**Defining Field Abstractions** Bandera includes tool support to ease the process of binding class fields to abstractions. Abstractions are indexed by type, thus when the user considers a field, such as `BoundedBuffer.bound`, the type, `int`, can be used to present the candidate abstractions from the library as illustrated in Figure 3. The user selects from these abstractions and repeats the process for other variables that have been determined to require abstraction. Once all such bindings have been made the tools calculate abstractions for all other fields in the program. The resulting inferred abstract types are displayed for the user to view as illustrated at the bottom of Figure 3. Conflicts in the inferred type for a given field are presented to the user for resolution. Fields which are unconstrained by the type inference can be set to a default type which is usually either the concrete type or the *point* abstraction.

### 2.2 Generating an Abstracted System Model and Property

Generating an abstract program involves three 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. The resulting abstract program yields an *over-approximation* of the concrete program's behavior. An over-approximation ensures that every behavior in the concrete program that violates a given property is also present in the abstract program. To ensure the soundness of verification results, the third step abstracts the property to be checked so as to *under-approximate* the set of behaviors described by the original property. An under-approximation ensures that any program behavior that

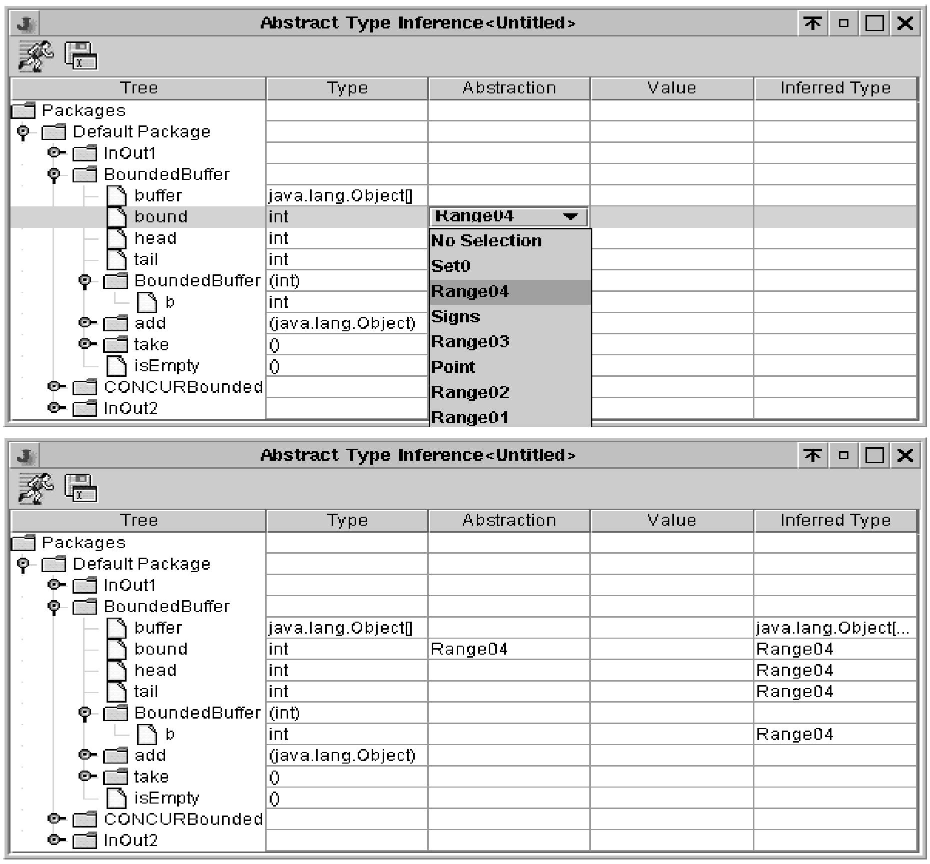


**Fig. 3.** Abstraction selection and abstract type inference

is contained in the set of behaviors described by the abstract property will also be contained in the set of behaviors described by the original property (these issues are discussed in greater detail below and in Section 7).

**Compiling Abstractions to Java** Figure 4 shows excerpts of the Java representation of the BASL even-odd specification in Figure 2. Abstract tokens are implemented as integer values (constructed by shifting `1` into the position indicated by the bit mask declarations), and the abstraction function and operations have straightforward implementations as Java methods. The most noteworthy aspect of the implementation is the modeling of the approximation that arises due to abstraction. The approximate nature of the even-odd abstraction means that a "greater than" comparison of any pair of abstract values could be `true` or `false`. Instead of representing such sets directly (e.g., as a bit vector), a single value is chosen non-deterministically from the set of possible values. This is valid when the meaning of a particular program is taken to be the collection of all

```
public class EvenOdd {
  public static final int EVEN  = 0; // bit mask
  public static final int ODD = 1; // bit mask

  public static int abs(int n) {
    if (n % 2 == 0) return (1 << EVEN);
    if (n % 2 == 1 || n % 2 == -1) return (1 << ODD);
    throw new RuntimeException();
  }

  public static int mul(int arg1, int arg2) {
    if (arg1==(1 << ODD)  && arg2==(1 << ODD))  return (1 << ODD);
    return (1 << EVEN);
  }

  public static boolean gt(int arg1, int arg2) {
    return Bandera.choose();
  }
}
```

**Fig. 4.** Compilation of BASL EvenOdd AI (excerpts)

possible traces or executions of the program. In Figure 4, the `Bandera.choose()` method denotes a non-deterministic choice between `true` and `false` values. This method has no Jimple implementation; instead, when Bandera compiles the abstracted program down to the input of given a model-checker, the method is implemented in terms of the model-checker's built-in constructs for expressing non-deterministic choice. Since the model-checker will generate all paths leading out of a non-deterministic choice, this ensures that all appropriate behaviors are represented in the model. This approach has some interesting implications compared to more tradition presentations of abstract interpretation. Using non-determinism to model imprecision in this manner (in essence, by transforming data imprecision into extra control paths), means that the abstract interpretation is maximally polyvariant, and there is never any merging of information using, for example, least-upper-bound operators. This approach can be effective since abstract domains in Bandera are finite, of finite height, and typically quite small. An alternative approach would be to use a set of abstract tokens to represent imprecision and to represent the set as a bit-vector. However, splitting sets into single tokens using non-determinism as described above yields a much simpler implementation.

**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 phase outlined 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 even-odd and `y` was not selected for abstraction. This code fragment will be transformed into:

```
EvenOdd.add(EvenOdd.add(x, Coerce.IntToEvenOdd(y)),
            EvenOdd.Even);
```

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

**Property Abstraction** Bandera's program abstraction approach yields a model in which execution states safely over-approximate the values of program variables. For example, a concrete state where variable `x` has the value 2 may be approximated by a modulo-2 abstracted value of *even*. When abstracting properties, this can be problematic if the abstractions do not preserve the ability to exactly decide the predicates in the property. Consider a predicate `x==4` evaluated in the example state described above. This predicate would appear to be true in the abstract state, since 4 is clearly abstracted by *even*, but the predicate evaluates to false in the concrete state.

Bandera abstracts the predicates appearing in the property being checked using an approach that is similar to [20]. Consider an AI for a variable `x` (*e.g.*, signs) that appears in a predicate (*e.g.*, `(x<1)`). Bandera converts this to a disjunction of predicates of the form `x==`*a*, where *a* are the abstract values that correspond to values that imply the truth of the original predicate (*e.g.*, `x==`*neg* implies `x<1` as does `x==`*zero*, but `x==`*pos* does not). Thus, this abstract disjunction, `x==`*zero* `&& x==`*neg*, under-approximates the concrete predicate insuring that the property holds on the original program if it does on the abstract program.

### 2.3 Abstract Model Checking

The resulting abstracted program and property are converted to BIR from their Jimple form and then to the input language of the selected model checker. Bandera runs the model checker and displays the results to the user. Counter-examples are mapped back to the unabstracted source program.

In addition to supporting exhaustive and sound verification of properties, Bandera provides a number of useful bounded state-space exploration strategies. Bounds can be placed on resources such as the size of integers and arrays, on the number of threads allocated, and on the number of object instances allocated at particular allocator sites. Bandera can construct models for existing model checkers, such as Spin, that perform *resource-bounded searches* [18] that can often yield effective bug-finding without performing any abstraction. These searches can be thought of as depth-bounded searches where the depth of the search is controlled by the bounds placed on different resources. When the bound is exceeded for a particular resource along a particular execution path, the model-checker simply aborts the search along that path and continues searching other unexplored paths.

### 2.4 Counter-example Feasibility

Model checking an abstracted program may produce a counter-example that is infeasible with respect to the program's concrete semantics. Since counter-examples may be very long and complex, user's require tool support to assist in the determination of feasibility. Bandera includes both an on-line technique for biasing the state-space search to find guaranteed feasible counter-examples and an off-line for simulating a counter-example on the abstract and concrete programs and checking their correspondence. The former, while unsound, has the advantage of being fast and surprisingly effective. A detailed presentation of these techniques is given in [27].

## 3 Program Syntax and Semantics

We noted in the previous section that Bandera translates Java programs into the Jimple intermediate form. To capture the essence of the Jimple structure for our formal overview of Bandera abstraction, we use the simple flowchart language FCL of Gomard and Jones [13, 14, 19].

Since our abstraction framework involves presenting abstraction definitions as types, we present a formalization of the framework using multi-sorted algebras. This allows new abstractions to be introduced as new sorts/types.

### 3.1 Signatures and Algebras

A signature $\Sigma$ is a structure containing a set $\mathsf{Types}[\Sigma]$ of types (which must include the distinguished type $\mathsf{Bool}$), a non-reflexive subtyping relation $\ll_\Sigma$ between the types of $\mathsf{Types}[\Sigma]$ that forms an upper semi-lattice and for each $(\tau_1, \tau_2) \in \ll_\Sigma$ a coercion symbol $[\tau_1 \ll_\Sigma \tau_2]$, a set $\mathsf{Ops}[\Sigma]$ of typed operation (*e.g.*, $+$), a set $\mathsf{Tests}[\Sigma]$ of typed test symbols (*e.g.*, $=$, $>$), and a set $\mathsf{Cons}[\Sigma]$ of typed constant symbols (*e.g.*, 2, 3). For notational simplicity, we will only consider binary operations and tests. The type of an operation $o \in \mathsf{Ops}[\Sigma]$ is denoted $[o]_\Sigma = \tau_1 \times \tau_2 \to \tau$ (similarly for tests and constants). For simplicity, for operation types $[o]_\Sigma = \tau_1 \times \tau_2 \to \tau$ we assume $\tau_1 = \tau_2 = \tau$ and for test types $[o]_\Sigma = \tau_1 \times \tau_2 \to \mathsf{Bool}$ we assume $\tau_1 = \tau_2$. This corresponds to the type structure of most of the built in base type operations in Java.

A $\Sigma$-algebra is a structure containing for each $\tau \in \mathsf{Types}[\Sigma]$ a semantic domain $[\![\tau]\!]^A_\Sigma$, for each pair $(\tau_1, \tau_2) \in \ll_\Sigma$ a total coercion relation $[\![[\tau_1 \ll_\Sigma \tau_2]]\!]^A_\Sigma \subseteq [\![\tau_1]\!]^A_\Sigma \times [\![\tau_2]\!]^A_\Sigma$, for each operation symbol $o \in \mathsf{Ops}[\Sigma]$ with type $\tau \times \tau \to \tau$ a relation $[\![o]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma$, for each test symbol $t \in \mathsf{Tests}[\Sigma]$ with type $\tau \times \tau \to \mathsf{Bool}$ a total relation $[\![t]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma \times [\![\tau]\!]^A_\Sigma \times [\![\mathsf{Bool}]\!]$ where $[\![\mathsf{Bool}]\!] = \{\mathit{true}, \mathit{false}\}$, and for each constant symbol $c \in \mathsf{Cons}[\Sigma]$ with type $\tau$ a set $[\![c]\!]^A_\Sigma \subseteq [\![\tau]\!]^A_\Sigma$ (we will drop the super/sub-scripts $\Sigma$ and $A$ when these are clear from the context). Using relations instead of functions to model the semantics of operations and tests (and sets instead of a single value for constants) provides a convenient way to capture the imprecision of abstractions.

In Bandera, the abstraction process begins by considering the concrete semantics of a program which we will model using a *basis configuration* – a basis signature $\Sigma_{basis}$ with

$$\begin{aligned}
\mathsf{Types}[\Sigma_{basis}] &= \{\mathsf{Int}, \mathsf{Bool}\}, \\
\ll_{\Sigma_{basis}} &= \emptyset, \\
\mathsf{Ops}[\Sigma_{basis}] &= \{+, -, *, \ldots\}, \\
\mathsf{Tests}[\Sigma_{basis}] &= \{>, =, \&\&, ||, \ldots\}, \\
\mathsf{Cons}[\Sigma_{basis}] &= \{\ldots, -1, 0, 1, \ldots, \mathsf{true}, \mathsf{false}\}
\end{aligned}$$

and a basis algebra $A_{basis}$ with the usual carrier sets for $[\![\mathsf{Int}]\!]$ and $[\![\mathsf{Bool}]\!]$, the usual functional interpretation for operations and tests $[\![+]\!]$, $[\![-]\!]$, $[\![>]\!]$, $[\![=]\!]$, *etc.*, and singleton set interpretations for integer and boolean constants, *e.g.*., $[\![1]\!] = \{1\}$, $[\![\mathsf{true}]\!] = \{\mathit{true}\}$. The subtyping relation is empty in the basis signature, because we use subtyping to express refinement relationships between abstractions, and no abstractions appear in as yet unabstracted concrete programs.

### 3.2 Program and Property Syntax

**Program syntax** Figure 5 presents the definition of FCL syntax. An FCL program consists of a list of parameters $x^*$, a label $l$ of the initial block to be executed, and a non-empty list $b^+$ of *basic blocks*. Each basic block consists of a *label* followed by a (possibly empty) list of *assignments*. Each block concludes with a *jump* that transfers control from that block to another one. Even though the syntax specifies prefix notation for operators, we will often use infix notation in examples with deeply nested expressions to improve readability. As noted earlier, the basis signature contains an empty subtyping relation, so coercion expressions $[\tau \ll_{\Sigma} \tau']\ e$ will not appear in concrete programs to be abstracted.

Figure 6 presents an FCL program that computes the power function. The input parameters to the program are `m` and `n`, and the initial block is specified by the line `(init)`. The parameters can be referenced and assigned to throughout the program. Other variables such as `result` can be introduced without explicit declaration. The initial value of a non-parameter variable is 0. The output of program execution is the state of memory when the **return** construct is executed.

In the presentation of FCL semantics, we need to reason about nodes in a *statement-level control-flow graph* (CFG),*i.e.*, a graph where there is a separate node $n$ for each assignment and jump in a given program $p$. We will assume that each statement (CFG node) has an identifier that is unique within the program, and we will annotate each statement in the source code with its unique identifier. For example, the first assignment in the `loop` block has the unique identifier (or node number) [`loop`.1].

To access code at particular program points within a given FCL-program, $p$, we use the functions $\mathit{code}[p]$, $\mathit{first}[p]$, $\mathit{succ}[p]$, defined below. We will drop the $[p]$ argument from the functions when it is clear from the context.

*Syntax Domains*

$$
\begin{array}{llll}
p & \in \text{Programs}[\Sigma] & x & \in \text{Variables}[\Sigma] \\
b & \in \text{Blocks}[\Sigma] & T & \in \text{Type-Identifiers}[\Sigma] \\
l & \in \text{Block-Labels}[\Sigma] & e & \in \text{Expressions}[\Sigma] \\
a & \in \text{Assignments}[\Sigma] & j & \in \text{Jumps}[\Sigma] \\
al & \in \text{Assignment-Lists}[\Sigma] & o & \in \mathsf{Ops}[\Sigma] \\
 & & t & \in \mathsf{Tests}[\Sigma] \\
 & & c & \in \mathsf{Cons}[\Sigma]
\end{array}
$$

*Grammar*

$$
\begin{array}{ll}
p \ ::= (x^*)\ (l)\ b^+ & a ::= x \mathbin{\mathbf{:=}} e; \ \mid \ \mathbf{skip}; \\
b \ ::= l :\ al\ j & e ::= c \ \mid \ x \ \mid \ T\ e \ \mid \ o(e_1, e_2) \ \mid \ t(e_1, e_2) \\
al ::= a^* & j ::= \mathbf{goto}\ l; \ \mid \ \mathbf{return}; \ \mid \\
 & \quad\ \ \mathbf{if}\ e\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2;
\end{array}
$$

*Typing Rules (expressions)*

$$
\frac{}{\Gamma \vdash_\Sigma c : [c]_\Sigma} \qquad \frac{}{\Gamma \vdash_\Sigma x : \Gamma(x)}
$$

$$
\frac{\Gamma \vdash_\Sigma e_i : \tau \qquad [o]_\Sigma = \tau \times \tau \to \tau}{\Gamma \vdash_\Sigma o(e, e) : \tau}
$$

$$
\frac{\Gamma \vdash_\Sigma e_i : \tau \qquad [t]_\Sigma = \tau \times \tau \to \mathsf{Bool}}{\Gamma \vdash_\Sigma t(e_1, e_2) : \mathsf{Bool}}
$$

$$
\frac{\Gamma \vdash_\Sigma e : \tau \qquad (\tau, \tau') \in \ll_\Sigma}{\Gamma \vdash_\Sigma [\tau \ll_\Sigma \tau']\ e : \tau'}
$$

**Fig. 5.** Syntax of the Flowchart Language FCL for a signature $\Sigma$

- The code map function $code[p]$ maps a CFG node $n$ to the code for the statement that it labels. Taking the program of Figure 6 as an example, $code$([`loop`.2]) yields the assignment `n := -(n 1);`.
- The function $first[p]$ maps a label $l$ of $p$ to the first CFG node occurring in the block labeled $l$. For example, $first$(`loop`) = [`loop`.1].
- The function $succ[p]$ maps each node to the set of nodes which are immediate successors of that node. For examples, $succ$([`test`.1]) = {[`loop`.1], [`end`.1]}.

**Property Syntax** LTL [22] is a rich formalism for specifying state and action sequencing properties of systems. An LTL specification describes the intended behavior of a system on all possible executions.

The syntax of LTL in Figure 7 includes primitive propositions $P$ with the usual propositional connectives, and three temporal operators. Bandera distinguishes logical connectives (e.g., $\wedge$, $\vee$) in the specification logic from logical

```
(m n)
(init)

  init: result := 1;                          [init.1]
        goto test;                            [init.2]

  test: if <(n, 1) then end else loop;        [test.1]

  loop: result := *(result, m);               [loop.1]
        n := -(n, 1);                         [loop.2]
        goto test;                            [loop.3]
  end:  return;                               [end.1]
```

**Fig. 6.** Power FCL program

*Syntax Domains*

$\psi \in \text{Formulas}[\Sigma]$ $\qquad$ $e \in \text{Expressions}[\Sigma]$
$P \in \text{Propositions}[\Sigma]$

*Grammar*

$$\begin{aligned} \psi ::= {} & P \mid \neg P \mid \\ & \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \\ & \Box\psi \mid \Diamond\psi \mid \psi_1 \ \mathcal{U}\ \psi_2 \end{aligned} \qquad P ::= [n] \mid e$$

**Fig. 7.** Syntax of the FCL property language for a signature $\Sigma$

program operations, and it automatically transforms property specifications to Negation Normal Form (NNF) [17] to simplify the property abstraction process. Accordingly, the syntax of Figure 7 only generates formulas in NNF, and we will assume but not encode explicitly the fact that expressions $e$ in propositions do not contain logical operators.

When specifying properties of software systems, one typically uses LTL formulas to reason about execution of particular program points (*e.g.*, entering or exiting a procedure) as well as values of particular program variables. To capture the essence of this for FCL, we use the following primitive propositions.

- Intuitively, $[n]$ holds when execution reaches the statement with unique identifier $n$ (*i.e.*, the statement at node $n$ will be executed next). We call propositions of this form *node propositions*.
- Intuitively, $e$ holds when the evaluation of $e$ at the current node is not false. We call propositions of this form *variable propositions*.

For example, a program requirement that says if `m` is odd initially, then when the program terminates `result` is odd, can be written as

$$\Diamond[\mathit{end}] \Rightarrow \Box([\mathit{init}] \wedge =(\%(\mathtt{m}, 2), 1) \Rightarrow \Diamond([\mathit{end}] \wedge =(\%(\mathtt{result}, 2), 1))).$$

Converting the property to NNF yields

$$\Box\neg[end] \lor \Box(\neg[init] \lor \neg{=}(\%(\mathtt{m},2),1) \lor \Diamond([end] \land {=}(\%(\mathtt{result},2),1))).$$

This particular property is an instance of a global response property [11] under the assumption[26] that the program eventually terminates (i.e., reach `end`). We need the assumption to work with since our abstractions cannot preserve liveness property. That is, an abstracted program may violate some liveness properties even though the the original program does not. This is due to the imprecision introduced in our abstraction process, for example, when an abstracted loop condition cannot be decided this gives rise to infinite traces in the program that may violate the liveness property.

Given an LTL formula $\psi$ where $\mathcal{P}$ is the of set of primitive propositions appearing in $\psi$, we will write Nodes[$\mathcal{P}$] for the set of CFG nodes mentioned in node propositions in $\mathcal{P}$, and Vars[$\mathcal{P}$] for the set of variables mentioned in variable propositions in $\mathcal{P}$.

### 3.3 Program and Property Semantics

**Program semantics** Figure 8 presents the semantics of FCL programs. The interpretation of a $\Sigma$-program $p$ is parameterized on a $\Sigma$-algebra $A$. Given a $\Sigma$, $A$, and a type-assignment $\Gamma$ mapping $\Sigma$-program variables to $\Sigma$-types, a store $\sigma$ is $(\Sigma, A, \Gamma)$-compatible when $domain(\sigma) = domain(\Gamma)$ and for all $x \in domain(\Gamma)$ . $\sigma(x) \in [\![\Gamma(x)]\!]$. The set of $(\Sigma, A, \Gamma)$-compatible stores is denoted $[\![\Gamma]\!]^A_\Sigma$. The semantics of a program is expressed via transitions on program states $(n, \sigma)$ where $n$ is a CFG node identifier from $p$ and $\sigma$ is a $(\Sigma, A, \Gamma)$-compatible store. A series of transitions gives an *execution trace* through $p$'s statement-level control flow graph. It is important to note that when execution is in state $(n_i, \sigma_i)$, the code at node $n_i$ has not yet been executed. Intuitively, the code at $n_i$ is executed on the transition from $(n_i, \sigma_i)$ to successor state $(n_{i+1}, \sigma_{i+1})$.

Figure 8 gives a simple operational semantics that formalizes the transition relation on states. In contrast to the small-step formalization of transition relation, a big-step semantics is used to formalize expression evaluation since we consider expression evaluation to be atomic. The top of the figure gives the definition of expression, assignment, and jump evaluation. The intuition behind the rules for these constructs is as follows.

- $\sigma \vdash_{expr} e \Rightarrow v$ means that under store $\sigma$, expression $e$ evaluates to value $v$.
- $\sigma \vdash_{assign} a \Rightarrow \sigma'$ means that under store $\sigma$, the assignment $a$ yields the updated store $\sigma'$.
- $\sigma \vdash_{jump} j \Rightarrow l$ means that under the store $\sigma$, jump $j$ will cause a transition to the block labeled $l$.

The three transition rules describe small-step transitions caused by assignment evaluation, jump evaluation leading to a new block, and jump evaluation leading to termination. We assume that the set of node labels Nodes[FCL] used in the semantics contains a distinguished node halt which we use to indicate a terminal state.

*Expressions and Assignments*

$$\frac{v \in [\![c]\!]^A_\Sigma}{\sigma \vdash_{expr} c \Rightarrow v} \qquad \frac{}{\sigma \vdash_{expr} x \Rightarrow \sigma(x)} \qquad \frac{\sigma \vdash_{expr} e \Rightarrow v \qquad (v, v') \in [\![[\tau \ll_\Sigma \tau']]\!]^A_\Sigma}{\sigma \vdash_{expr} [\tau \ll_\Sigma \tau']\, e \Rightarrow v'}$$

$$\frac{\sigma \vdash_{expr} e_i \Rightarrow v_i \qquad [\![o]\!]^A_\Sigma(v_1, v_2, v)}{\sigma \vdash_{expr} o(e_1, e_2) \Rightarrow v} \qquad \frac{\sigma \vdash_{expr} e_i \Rightarrow v_i \qquad [\![t]\!]^A_\Sigma(v_1, v_2, v)}{\sigma \vdash_{expr} t(e_1, e_2) \Rightarrow v}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{assign} x := e \Rightarrow \sigma[x \mapsto v]} \qquad \frac{}{\sigma \vdash_{assign} \mathbf{skip} \Rightarrow \sigma}$$

*Jumps*

$$\frac{}{\sigma \vdash_{jump} \mathbf{goto}\ l \Rightarrow l} \qquad \frac{}{\sigma \vdash_{jump} \mathbf{return} \Rightarrow \mathsf{halt}}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \mathit{true}}{\sigma \vdash_{jump} \mathbf{if}\ e\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2 \Rightarrow l_1}$$

$$\frac{\sigma \vdash_{expr} e \Rightarrow \mathit{false}}{\sigma \vdash_{jump} \mathbf{if}\ e\ \mathbf{then}\ l_1\ \mathbf{else}\ l_2 \Rightarrow l_2}$$

*Transitions*

$$\frac{\sigma \vdash_{assign} a \Rightarrow \sigma'}{(n, \sigma) \longmapsto (n', \sigma')} \qquad \begin{array}{l} \text{if} \quad \mathit{code}(n) = a \\ \text{where } n' = \mathit{succ}(n) \end{array}$$

$$\frac{\sigma \vdash_{jump} j \Rightarrow l}{(n, \sigma) \longmapsto (n', \sigma)} \qquad \begin{array}{l} \text{if} \quad \mathit{code}(n) = j \\ \text{where } n' = \mathit{first}(l) \end{array}$$

$$\frac{\sigma \vdash_{jump} j \Rightarrow \mathsf{halt}}{(n, \sigma) \longmapsto (\mathsf{halt}, \sigma)} \qquad \text{if } \mathit{code}(n) = j$$

*Semantic Values*

$$\begin{array}{l} n \in \text{Nodes[FCL]} \\ s \in \text{States[FCL]} = \text{Nodes[FCL]} \times \text{Stores[FCL]} \end{array}$$

**Fig. 8.** Operational semantics of a $\Sigma$-FCL program with respect to $\Sigma$-algebra $A$

**Property semantics** The semantics of a primitive proposition is defined with respect to states.

$$[\![[m]]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{\mathit{true}\} & \text{if } m = n \\ \{\mathit{false}\} & \text{otherwise} \end{cases}$$

$$[\![\neg[m]]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{\mathit{true}\} & \text{if } m \neq n \\ \{\mathit{false}\} & \text{otherwise} \end{cases}$$

$$[\![e]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{\mathit{true}\} & \text{if } \sigma \vdash_{expr} e \not\Rightarrow \mathit{false} \\ \{\mathit{false}\} & \text{otherwise} \end{cases}$$

$$[\![\neg e]\!]^A_\Sigma(n, \sigma) = \begin{cases} \{\mathit{true}\} & \text{if } \sigma \vdash_{expr} e \not\Rightarrow \mathit{true} \\ \{\mathit{false}\} & \text{otherwise} \end{cases}$$

Note that the semantics of expression propositions defines an under-approximation, i.e., the proposition expression is not considered true if the expression evaluates to $\{true, false\}$.

The semantics of an LTL formula is defined with respect to traces, where each trace is a (possibly infinite) non-empty sequence of states written $\Pi = s_1, s_2, \ldots$. We write $\Pi^i$ for the suffix starting at $s_i$, *i.e.*, $\Pi^i = s_i, s_{i+1}, \ldots$. Thus, an execution trace of $p$ is a state sequence $\Pi = s_1, s_2, \ldots$ with the following constraints: $s_1$ is an initial state for $p$ and $s_i \longmapsto s_{i+1}$.

The temporal operator $\Box$ requires that its argument be true from the current state onward, the $\Diamond$ operator requires that its argument become true at some point in the future, and the $\mathcal{U}$ operator requires that its first argument is true up to the point where the second argument becomes true. We refer the reader to, e.g., [17], for a formal definition of the semantics of LTL.

## 4 Defining Abstractions

Section 2.1 noted that each Bandera abstraction is associated with a concrete type $\tau$ and that each abstraction definition has four components: an abstraction name, an abstract domain, an abstraction function/relation, and abstract versions of each concrete operation and test. Accordingly, if $\tau$ is a type from $\Sigma$, an $[\Sigma, A]$-compatible *$\tau$-abstraction* $\alpha$ is a structure containing an abstraction type identifier $\tau_\alpha$, a finite abstraction domain $[\![\tau_\alpha]\!]$, an abstraction relation $\rightsquigarrow_\alpha \subseteq [\![\tau]\!]_\Sigma^A \times [\![\tau_\alpha]\!]$, for each $\tau$ operation symbol $o$ in $\Sigma$ an operation $[\![o_\alpha]\!] \subseteq [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!]$, and for each $\tau$ test symbol $t$ in $\Sigma$ a test $[\![o_\alpha]\!] \subseteq [\![\tau_\alpha]\!] \times [\![\tau_\alpha]\!] \times [\![\mathsf{Bool}]\!]$.

To ensure that properties that hold true for the abstracted system also hold true in the original concrete system, one needs the standard notion of safety (denoted $\lhd$) as a simulation between operation/test relations.

**Definition 1.** *(Safe abstract operations and abstract tests)*
*Let $\tau_\alpha$ be a $\tau$-abstraction.*

- $[\![o]\!] \lhd [\![o_\alpha]\!]$ *iff for every $c_1, c_2, c \in [\![\tau]\!]$, and $a_1, a_2 \in [\![\tau_\alpha]\!]$, if $c_1 \rightsquigarrow_\alpha a_1$, $c_2 \rightsquigarrow_\alpha a_2$, and $[\![o]\!](c_1, c_2, c)$ then there exists $a \in [\![\tau_\alpha]\!]$ such that $[\![o_\alpha]\!](a_1, a_2, a)$ and $c \rightsquigarrow_\alpha a$,*
- $[\![t]\!] \lhd [\![t_\alpha]\!]$ *iff for every $c_1, c_2 \in [\![\tau]\!]$, $a_1, a_2 \in [\![\tau_\alpha]\!]$, and $b \in [\![\mathsf{Bool}]\!]$, if $c_1 \rightsquigarrow_\alpha a_1$, $c_2 \rightsquigarrow_\alpha a_2$, and $[\![t]\!](c_1, c_2, b)$, then $[\![t_\alpha]\!](a_1, a_2, b)$.*

As noted in Section 2.1, when defining an abstract type $\tau_\alpha$ for integers in Bandera, the user only needs to use BASL to specify its abstraction domain $[\![\tau_\alpha]\!]$ and its abstraction relation $\rightsquigarrow_\alpha$. Safe operations and tests involving the new abstract type are generated by Bandera automatically using a calculation similar in style to those used to calculate weakest-preconditions in predicate abstraction.

For example, suppose that the user wants to define a new integer abstraction $\tau_{signs}$. The user then can define the abstraction domain as $[\![\tau_{signs}]\!] =$

$\{neg, zero, pos\}$. In writing the abstraction function, the user would write simple predicates to create a covering of the integer domain (e.g., as shown for the EvenOdd BASL definition in Figure 2). In our formal notation, we capture this using the following predicates for each of the abstract tokens in the $\tau_{signs}$ domain: $neg? = \lambda x.\, x < 0, zero? = \lambda x.\, x = 0$, and $pos? = \lambda x.\, x > 0$. Given these predicates, we can define the associated abstraction relation $\rightsquigarrow_{signs}$ as

$$\forall x \in [\![\mathsf{Int}]\!]\,.\, \forall a \in [\![\tau_{signs}]\!]\,.\, x \rightsquigarrow_{signs} a \text{ iff } a?(x).$$

Given these definitions, Bandera automatically constructs a *safe* definition for each abstract operation and test essentially by (a) beginning with a worst case assumption that the relation defining the abstract operation is total (note that this is a safe definition since a total relation covers all possible behaviors of the concrete system), and then (b) calling the decision procedures of PVS to see if individual tuples in the relation can be eliminated.

For example, consider how the definition of $+_{signs}$ would be derived. Since the abstract domain $[\![\tau_{signs}]\!]$ contains 3 abstract tokens, we would initially have 27 tuples in the total relation associated with $[\![+_{signs}]\!]$. Now, for each tuple $(a_1, a_2, a) \in [\![+_{signs}]\!]$, Bandera would construct a purported theorem in the input syntax of PVS which we represent as follows

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!]\,.\, a_1?(n_1) \wedge a_2?(n_2) \implies \neg a?([\![+]\!](n_1, n_2)).$$

If PVS can prove the fact above, then the tuple $(a_1, a_2, a)$ can safely be eliminated relation defining $[\![+_{signs}]\!]$ because, since the theorem is true, $a$ is never needed to simulate the result of adding $n_1$ and $n_2$.

Specifically, consider the three tuples $(pos, pos, neg)$, $(pos, pos, zero)$, and $(pos, pos, pos)$. The decision procedure is able prove the two theorems

$$\begin{array}{l}\forall n_1, n_2 \in [\![\mathsf{Int}]\!]\,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg neg?([\![+]\!](n_1, n_2)),\\ \forall n_1, n_2 \in [\![\mathsf{Int}]\!]\,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg zero?([\![+]\!](n_1, n_2)),\end{array}$$

but it fails to prove

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!]\,.\, pos?(n_1) \wedge pos?(n_2) \implies \neg pos?([\![+]\!](n_1, n_2)),$$

so Bandera would remove the first two tuples from the relation and have $[\![+_{signs}]\!](pos, pos) = \{pos\}$ (depicting the relation as a set-valued function).

In summary, the definitions for abstract operations and tests for integer abstractions is as follows.

**Definition 2.** *Let $\tau_\alpha$ be an integer abstraction.*

- *For all $a_1, a_2, a \in [\![\tau_\alpha]\!]$, $[\![o_\alpha]\!](a_1, a_2, a)$ iff the decision procedure fails to decide*

$$\forall n_1, n_2 \in [\![\mathsf{Int}]\!]\,.\, a_1?(n_1) \wedge a_2?(n_2) \implies \neg a?([\![o]\!](n_1, n_2)).$$

- *For all $a_1, a_2 \in [\![\tau_\alpha]\!]$ and $b \in [\![\mathsf{Bool}]\!]$, $[\![t_\alpha]\!](a_1, a_2, b)$, iff the decision procedure fails to decide*

$$\forall n_1, n_2.\, a_1?(n_1) \wedge a_2?(n_2) \implies b \neq [\![t]\!](n_1, n_2).$$

This technique can also be used to infer coercions between two integer abstractions $\alpha$ and $\alpha'$. Specifically, $(a, a') \in [\![[\tau_\alpha \ll \tau_{\alpha'}]]\!]$ if the decision procedure fails to decide $\forall n.a?(n) \Longrightarrow \neg a'?(n)$.

## 5 Attaching Abstractions

Bandera's process for transforming concrete programs to abstract programs requires that each variable and each occurrence of a constant, operation, and test in the concrete program be bound to an abstraction type; these bindings indicate to the program transformer which versions of abstract program elements should be used in place of a particular concrete element (e.g., which abstract version of the + operation should be used in place of a particular concrete instance of +). Requiring the user to specify all of this binding information directly would put a tremendous burden on the user.

To construct the desired binding information while at the same time keeping user effort to a minimum, Bandera provides an abstract type inference facility. A user begins the type inference phase by selecting abstractions from the abstraction library for a small number of program variables that the user deems relevant. Bandera provides a default set of coercions between library abstractions for each concrete type that the user can override if desired. For example, for any $\mathsf{Int}$ abstraction $\tau_\alpha$ a coercion relation $[\![[\mathsf{Int} \ll \tau_\alpha]]\!]$ is automatically introduced by taking $[\![[\mathsf{Int} \ll \tau_\alpha]]\!] = \ \leadsto_\alpha$ (i.e., the coercion is just the abstraction function) and a coercion relation $[\![[\tau_\alpha \ll \mathsf{Point}]]\!]$ is automatically introduced where $[\![[\tau_\alpha \ll \mathsf{Point}]]\!]$ simply maps all elements of $[\![\tau_\alpha]\!]$ to the single element of the $\mathsf{Point}$ domain. The default coercion definitions plus any user-defined coercions give rise to a subtyping structure for each concrete type. This abstraction selection and subtyping/coercion information forms the input to the type inference component. In the Bandera methodology, boolean variables are never abstracted since they already have a small domain. We will model this in the definitions below by abstracting all boolean variables with an identity abstraction which has the effect of leaving boolean variables and values unchanged by the abstraction process.

Given the program, initial abstraction selection, and subtyping information, type inference proceeds in two steps.

1. Abstract types are propagated along value flows to determine abstraction bindings for as many constructs as possible. If there are any abstract type conflicts during this process, type inference is halted and the user is presented with an error message.
2. Some variables and constructs may not be assigned abstract types in the first step because they are *independent* of the variables in the initial abstraction selection. Abstractions for independent variables/constructs are determined according to default abstractions specified by the user. The most commonly used defaults are (a) to abstract all independent variables/constructs with the *point* abstraction which has the effect of discarding all information about values manipulated by these constructs, or (b) to abstract these constructs

*Annotated Program*

```
(m n)
(init)

  init: result := 1^1;
        goto test;

  test: if (< (n^2, 1^3))^4
        then end else loop;

  loop: result := (*(result^5, m^6))^7;
        n := (-(n^8, 1^9))^10;
        goto test;

  end:  return;
```

*Abstracted Program*

```
(m n)
(init)

  init: result := odd;
        goto test;

  test  if <(n, 1)
        then end else loop;

  loop: result := *_eo(result, m);
        n := -(n, 1);
        goto test;

  end:  return;
```

**Fig. 9.** Annotated and abstracted versions of the FCL power program

with the identity abstraction which has the effect of preserving all information about the values manipulated by these constructs.

As an example, consider the power program and the LTL formula from Section 3.2. Following the methodology for selecting abstractions in Section 2.1, the EvenOdd abstraction would be *appropriate* for the variable `m` of the power program in Figure 6. Intuitively, an abstraction is appropriate for a property when it is precise enough to decide all propositions appearing in the property.

Suppose that the abstraction library that is used contains the concrete (identity) Int, EvenOdd, Signs, and Point abstractions for integer types. The subtyping relation between abstractions must always form a lattice; the default subtyping relation has Int as the least element, Point as the greatest element, with the remaining abstractions being unrelated to each other. This lattice is augmented with an additional element $\perp$ which ends up being bound to variables/constructs whose type is unconstrained due to the fact that they are independent of the initial abstraction selection. The second phase of the type inference process described above involves replacing $\perp$ with one of the default options for unconstrained types. We write $\sqsubseteq$ to denoted the augmented subtyping ordering used in the type inference process. Intuitively, if $\tau_1 \sqsubseteq \tau_2$, then $\tau_1$ is at least as precise as $\tau_2$.

To represent the binding between program constructs and abstractions, we will assume that each expression abstract syntax tree node is annotated with an unique label as in the left side of Figure 9. Bindings are then captured by a *cache* structure $\hat{\mathcal{C}}$ which maps variables, labeled AST nodes, and operator/test instances to types in the augmented subtyping lattice. Following convention, type inference is phrased as a constraint-solving problem in which constraints

*Syntax Domain Extension*

$$\bar{l} \in \text{Exp-Labels[FCL]}$$
$$\bar{t} \in \text{Terms[FCL]}$$

*Grammar with Labeled Expression Extension*

$$e ::= \bar{t}^{\bar{l}} \qquad \bar{t} ::= c \mid x \mid o(e_1, e_2) \mid t(e_1, e_2)$$

*Constraints (Excerpts)*

$$\begin{array}{lll}
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models c^{\bar{l}} & \text{iff} & \perp \sqsubseteq \hat{\mathcal{C}}(\bar{l}) \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models x^{\bar{l}} & \text{iff} & \hat{\mathcal{C}}(x) = \hat{\mathcal{C}}(\bar{l}) \text{ and } x\mathcal{R}\bar{l} \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models (o(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2}))^{\bar{l}} & \text{iff} & \langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models \bar{t}_1^{\bar{l}_1} \text{ and } \langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models \bar{t}_2^{\bar{l}_2} \\
& & \quad \text{and } \hat{\mathcal{C}}(\bar{l}_1) \sqsubseteq \hat{\mathcal{C}}(o, \bar{l}) \text{ and } \hat{\mathcal{C}}(\bar{l}_2) \sqsubseteq \hat{\mathcal{C}}(o, \bar{l}) \\
& & \quad \text{and } \hat{\mathcal{C}}(o, \bar{l}) = \hat{\mathcal{C}}(\bar{l}) \\
& & \quad \text{and } \bar{l}_1\mathcal{R}(o, \bar{l}) \text{ and } \bar{l}_2\mathcal{R}(o, \bar{l}) \\
& & \quad \text{and } (o, \bar{l})\mathcal{R}\bar{l} \\
\langle \hat{\mathcal{C}}, \mathcal{R} \rangle \models x := \bar{t}^{\bar{l}} & \text{iff} & \hat{\mathcal{C}}(\bar{l}) \sqsubseteq \hat{\mathcal{C}}(x) \text{ and } \bar{l}\mathcal{R}x
\end{array}$$

*Data Structures*

$$\begin{array}{ll}
\text{(graph nodes)} & \mathcal{N} = \text{Variables[FCL]} \cup \text{Exp-Labels[FCL]} \cup \\
& \qquad ((\text{Operations[FCL]} \cup \text{Tests[FCL]}) \times \text{Exp-Labels[FCL]}) \\
\text{(cache)} & \hat{\mathcal{C}} = \mathcal{N} \rightharpoonup \mathsf{Types}[\Sigma_\alpha]_\perp \\
\text{(dependencies)} & \mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}
\end{array}$$

**Fig. 10.** Type Inference

on cache entries are generated in a syntax-directed traversal of the program and then solved using a union-find data structure.

### 5.1 Type Inference: Generating Constraints

Figure 10 presents the specification of constraint generation in the style of [23]. The data structures used include a type dependency graph with nodes $\mathcal{N}$ that are either variables, labels of AST nodes, or operation/test occurrences which are identified by a pair consisting of the operation/test symbol and a label for the node in which the operation/test instance occurs. As described above, a cache $\hat{\mathcal{C}}$ maps each graph node to a type from the augmented lattice $\mathsf{Types}[\Sigma_\alpha]_\perp$. The relation $\mathcal{R}$ maintains dependency information associated with value flows. This information is used in the second phase of type inference (described above) where constructs that are independent of initial abstraction bindings are assigned types. Due to the manner in which constraints are constructed, $x\mathcal{R}y$ implies $\hat{\mathcal{C}}(x) \sqsubseteq \hat{\mathcal{C}}(y)$.

Constraints on $\hat{\mathcal{C}}$ and $\mathcal{R}$ are generated in a syntax-directed manner according to the following intuition.

– There are no constraints on constants except those imposed by the context (which are captured by other rules). Thus, the type assigned to a constant can be any value in the lattice at or above $\perp$.
– A variable reference expression (which yields the value of variable) should have the same type as the variable itself, and $x\mathcal{R}\bar{l}$ captures the fact that the type of $x$ must be at least as precise as that of $\bar{l}$.
– The abstraction associated with the arguments of an operator application must lie at or below the abstraction associated with the operator itself. Note that this will allow the type of an argument to be coerced to the type of the operator.
– The abstraction associated with the right-hand side of an assignment must lie at or below the abstraction associated with the variable being assigned. Note that this will allow the type of the right-hand side to be coerced to the type of the left-hand side.

In addition to the constraints generated from the rules above, for every variable $x$ appearing in the user's initial abstraction selection, a constraint $\hat{\mathcal{C}}(x) = \tau_x$ is added where $\tau_x$ is the abstraction type chosen for $x$. For each remaining variable $y$, a constraint $\perp \sqsubseteq \hat{\mathcal{C}}(y)$ is added.

Generating the least solution for a system of constraints yields abstraction bindings that are as precise as possible (with respect to the subtyping rules). In particular, the ability to use coercions at operation/test arguments, etc. avoids having argument abstractions determined by the context (i.e., having to receive the same type as the operation), and thus allows abstraction assignments at such positions to be as precise as possible.

For the power program example on the left in Figure 9, the following constraints are generated (ignoring $\mathcal{R}$ for now).

| | | | |
|---|---|---|---|
| $\hat{\mathcal{C}}(1) = \perp$ | $\hat{\mathcal{C}}(1) \sqsubseteq \hat{\mathcal{C}}(\texttt{result})$ | $\hat{\mathcal{C}}(\texttt{n}) = \hat{\mathcal{C}}(2)$ | $\hat{\mathcal{C}}(2) \sqsubseteq \hat{\mathcal{C}}(<, 4)$ |
| $\hat{\mathcal{C}}(3) = \perp$ | $\hat{\mathcal{C}}(3) \sqsubseteq \hat{\mathcal{C}}(<, 4)$ | $\hat{\mathcal{C}}(<, 4) = \hat{\mathcal{C}}(4)$ | $\hat{\mathcal{C}}(\texttt{result}) = \hat{\mathcal{C}}(5)$ |
| $\hat{\mathcal{C}}(5) \sqsubseteq \hat{\mathcal{C}}(*, 7)$ | $\hat{\mathcal{C}}(\texttt{m}) = \hat{\mathcal{C}}(6)$ | $\hat{\mathcal{C}}(6) \sqsubseteq \hat{\mathcal{C}}(*, 7)$ | $\hat{\mathcal{C}}(*, 7) = \hat{\mathcal{C}}(7)$ |
| $\hat{\mathcal{C}}(7) \sqsubseteq \hat{\mathcal{C}}(\texttt{result})$ | $\hat{\mathcal{C}}(\texttt{n}) = \hat{\mathcal{C}}(8)$ | $\hat{\mathcal{C}}(8) \sqsubseteq \hat{\mathcal{C}}(-, 10)$ | $\hat{\mathcal{C}}(9) = \perp$ |
| $\hat{\mathcal{C}}(9) \sqsubseteq \hat{\mathcal{C}}(-, 10)$ | $\hat{\mathcal{C}}(-, 10) = \hat{\mathcal{C}}(10)$ | $\hat{\mathcal{C}}(10) \sqsubseteq \hat{\mathcal{C}}(\texttt{n})$ | |

Given a user selection of $\mathsf{EvenOdd}$ for m, the following additional constraints are generated.

$$\hat{\mathcal{C}}(\texttt{m}) = \mathsf{EvenOdd} \text{ and } \perp \sqsubseteq \hat{\mathcal{C}}(\texttt{n}) \text{ and } \perp \sqsubseteq \hat{\mathcal{C}}(\texttt{result})$$

### 5.2 Type Inference: Solving Constraints

Once constraints are generated as described above, Bandera finds the least solution with respect to $\sqsubseteq$. For example, the least solution for the constraints from the power example is as follows.

| | | | |
|---|---|---|---|
| $\hat{\mathcal{C}}(\texttt{m}) = \mathsf{EvenOdd}$ | $\hat{\mathcal{C}}(\texttt{n}) = \perp$ | $\hat{\mathcal{C}}(\texttt{result}) = \mathsf{EvenOdd}$ | $\hat{\mathcal{C}}(1) = \perp$ |
| $\hat{\mathcal{C}}(2) = \perp$ | $\hat{\mathcal{C}}(3) = \perp$ | $\hat{\mathcal{C}}(4) = \perp$ | $\hat{\mathcal{C}}(5) = \mathsf{EvenOdd}$ |
| $\hat{\mathcal{C}}(6) = \mathsf{EvenOdd}$ | $\hat{\mathcal{C}}(7) = \mathsf{EvenOdd}$ | $\hat{\mathcal{C}}(8) = \perp$ | $\hat{\mathcal{C}}(9) = \perp$ |
| $\hat{\mathcal{C}}(10) = \perp$ | $\hat{\mathcal{C}}(<, 4) = \perp$ | $\hat{\mathcal{C}}(*, 7) = \mathsf{EvenOdd}$ | $\hat{\mathcal{C}}(-, 10) = \perp$ |

As illustrated by the presence of $\perp$ in some of the bindings above, this step may leave the type of some variables/AST-nodes unconstrained. In general, $\perp$-bindings such as those shown above actually fall into two categories.

The first category contains variables/AST-nodes that produce values that can flow into a context that *is* constrained. For example, this is the case with the AST node labeled 1: the constant `1` flows into the variable `result` (which is bound to $\mathsf{EvenOdd}$) as a consequence of the assignment. Thus, to obtain an abstraction assignment that is precise as possible (i.e., one that does not "bump up" the abstraction assigned to `result` to a higher value in the lattice of abstractions), the abstraction chosen for such nodes should not be greater than that of any constrained context into which the values produced by such nodes can flow. The dependency information provided by the $\mathcal{R}$ structure is used to determine if a unconstrained node falls into this category (i.e., if a constrained context be reached by following the dependency arcs of $\mathcal{R}$).

The second category contains variables/AST-nodes that produce values that do *not* flow into constrained contexts. There are several reasonable views as to what the abstraction bindings should be for items in this category. One view is that one should generate models that are as abstract as possible in order to reduce the size of the state space as much as possible. Following this view, one might bind the $\mathsf{Point}$ abstraction to each item in this category. On the other hand, this could result in such an over-approximation that infeasible counter-examples are introduced. Thus, one might want to generate models that are as precise as possible. Following this view, one might bind the $\mathsf{Int}$ abstraction to each item in this category. Note that although such a choice might lead to an unbounded state-space (since integers are left unabstracted in the program), this is still quite useful in practice since model-checkers such as Spin allow arbitrary integer values with bounds imposed only by the number of bits used in the storage class (e.g., `byte`, `int`, etc.). Bandera actually provides a flexible mechanism for declaring upper and lower-bounds on individual integer variables.

In any case, the two categories above are currently treated as follows in Bandera. For the first category, Bandera binds items to the concrete $\mathsf{Int}$ abstraction. This always satisfies the constraints since $\mathsf{Int}$ is the least element in the non-augmented abstraction lattice, and it follows the heuristic of keeping abstractions as precise as possible. At the point where concrete integers flow into abstracted contexts, an appropriate coercion will be introduced in the model. Since items in the second category are completely unconstrained, Bandera allows the user to select a default abstraction $\tau_{\mathsf{def}}$ (typically, $\mathsf{Int}$ or $\mathsf{Point}$) for these items.

Capturing this in our formal notation, Bandera proceeds by building a new $\hat{\mathcal{C}}'$ from $\hat{\mathcal{C}}$ as follows.

$$\hat{\mathcal{C}}'(x) = \begin{cases} \hat{\mathcal{C}}(x), & \text{if } \hat{\mathcal{C}}(x) \neq \perp \\ \mathsf{Int}, & \text{if } \hat{\mathcal{C}}(\mathcal{R}^*(x)) \neq \{\perp\} \\ \tau_{\mathsf{def}}, & \text{if } \hat{\mathcal{C}}(\mathcal{R}^*(x)) = \{\perp\} \end{cases}$$

That is, already assigned an abstraction keep the same abstraction in $\hat{\mathcal{C}}'$, items from the first category above get assigned Int, and items from the second category get assigned the chosen default abstraction. In the example program, this results in the following final bindings.

$$
\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(\mathtt{n}) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(\mathtt{result}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(1) = \mathsf{Int} \\
\hat{\mathcal{C}}(2) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(3) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(4) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(5) = \mathsf{EvenOdd} \\
\hat{\mathcal{C}}(6) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(8) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(9) = \tau_{\mathsf{def}} \\
\hat{\mathcal{C}}(10) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(<,4) = \tau_{\mathsf{def}} & \hat{\mathcal{C}}(*,7) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(-,10) = \tau_{\mathsf{def}}
\end{array}
$$

As a future improvement to the treatment of items from the first category (where $\hat{\mathcal{C}}(\mathcal{R}^*(x)) \neq \{\bot\}$), it may be desirable to give user the flexibility to replace Int with any less precise abstraction $\tau$ that still lies at or below the abstraction of any context that an item's value may flow into, i.e.,

$$\tau \sqsubseteq \sqcap\{\hat{\mathcal{C}}'(y) \mid y \in \mathcal{R}^*(x) \text{and } \hat{\mathcal{C}}'(y) \neq \bot\}.$$

From a usability standpoint, it is important to note that the type inference algorithm outlined above is efficient and scales well, and that the process of selecting abstractions and visualizing type inference results is interactive. Thus, the user can experiment with the abstraction selection with ease, e.g., by incrementally adding the abstraction selections and visualizing the effects of each selection.

Bandera provides feedback to the user if the abstraction selection is inconsistent. For example, suppose that the user selects `m` as EvenOdd abstraction and `result` as Signs abstraction. A conflict arises because of the following constraints cannot be satisfied.

$$
\begin{array}{llll}
\hat{\mathcal{C}}(\mathtt{m}) = \mathsf{EvenOdd} & \hat{\mathcal{C}}(\mathtt{result}) = \mathsf{Signs} & \hat{\mathcal{C}}(\mathtt{result}) = \hat{\mathcal{C}}(5) & \hat{\mathcal{C}}(5) \sqsubseteq \hat{\mathcal{C}}(*,7) \\
\hat{\mathcal{C}}(\mathtt{m}) = \hat{\mathcal{C}}(6) & \hat{\mathcal{C}}(6) \sqsubseteq \hat{\mathcal{C}}(*,7) & \hat{\mathcal{C}}(*,7) = \hat{\mathcal{C}}(7) & \hat{\mathcal{C}}(7) \sqsubseteq \hat{\mathcal{C}}(\mathtt{result})
\end{array}
$$

## 6 Generating Abstract Programs

Once abstract type inference has been carried on a $\Sigma$-program interpreted with $A$, the set of $[\Sigma, A]$-compatible abstractions $\{\alpha_1, ..., \alpha_n\}$ chosen by the user and the final abstraction bindings from the type inference process are used to induce an abstract program based on a new signature and algebra $[\Sigma_\alpha, A_\alpha]$ that combines the selected abstractions. Section 2.2 noted that this process is implemented in Bandera by replacing primitive concrete Java operations in the program to be abstracted with calls to Java methods in abstraction library classes that implement semantics associated with abstract versions of operations.

We first formalize the notion of these library classes/methods by reifying the abstraction semantics from Section 4 into constants and symbols to be used in the signature for the abstract program. Specifically, given a *$\tau$-abstraction* $\alpha$, we form a new type named by $\alpha$'s abstraction type identifier $\tau_\alpha$. For this type, constants, operation symbols, and test symbols, are constructed as follows.

- $\mathsf{Cons}[\tau_\alpha] = \{\underline{a} \mid a \in [\![\tau_\alpha]\!]\}$. That is, $\tau_\alpha$ constants are formed by introducing a fresh symbol $\underline{a}$ for each element of the abstract domain. This corresponds to the use of constants such as `EvenOdd.Even` in abstracted Java programs (see Section 2.2).
- $\mathsf{Ops}[\tau_\alpha] = \{o_\alpha \mid o \in \mathsf{Ops}[\tau]\}$. That is, $\tau_\alpha$ operation symbols are formed by introducing a fresh symbol $o_\alpha$ for each operation symbol associated with the $\Sigma$-type $\tau$ being abstracted. This corresponds to the use of method calls such as `EvenOdd.add` (see Section 2.2).
- $\mathsf{Tests}[\tau_\alpha] = \{t_\alpha \mid t \in \mathsf{Ops}[\tau]\}$. That is, $\tau_\alpha$ test symbols are formed by introducing a fresh symbol $t_\alpha$ for each test symbol associated with the $\Sigma$-type $\tau$ being abstracted.

With these syntactic elements in hand, we now form a signature to be used for the abstracted program by combining the symbols introduced above. Given user-selected abstractions $\{\alpha_1, ..., \alpha_n\}$ along with default and user declared coercions, a new signature $\Sigma_\alpha$ representing this combination of abstractions is constructed as follows.

- $\mathsf{Types}[\Sigma_\alpha] = \{\tau_{\alpha_1}, ..., \tau_{\alpha_n}\}$ where $\tau_{\alpha_i}$ is the type identifier corresponding to each abstraction $\alpha_i$.
- $\mathsf{Ops}[\Sigma_\alpha] = \bigcup_{i \in \{1,\ldots,n\}} \mathsf{Ops}[\alpha_i]$.
- $\mathsf{Tests}[\Sigma_\alpha] = \bigcup_{i \in \{1,\ldots,n\}} \mathsf{Tests}[\alpha_i]$.
- $\ll_{\Sigma_\alpha} = \{(\tau_1, \tau_2) \mid \text{a coercion exists from } \tau_1 \text{ and } \tau_2\}$.

An appropriate abstract $\Sigma_\alpha$-algebra is now formed in a straightforward manner as follows.

- For all types $\tau_\alpha \in \Sigma_\alpha$, $[\![\tau_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![\tau_\alpha]\!]$ (i.e., the domain specified in the $\alpha$ abstraction).
- For all $o_\alpha \in \mathsf{Ops}[\Sigma_\alpha]$ where $o_\alpha$ is a $\tau_\alpha$ operation, $[\![o_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![o_\alpha]\!]$ (i.e., the operation interpretation specified in the $\alpha$-abstraction).
- For all $t_\alpha \in \mathsf{Tests}[\Sigma_\alpha]$ where $t_\alpha$ is a $\tau_\alpha$ test, $[\![t_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha} = [\![t_\alpha]\!]$ (i.e., the test interpretation specified in the $\alpha$-abstraction).
- For each coercion symbol $[\tau_1 \ll \tau_2]$, the corresponding coercion relation is defined for default coercions as explained earlier or defined by the user.

Figure 11 presents rules that formalize the translation of concrete programs to abstract programs. The rules are guided by bindings of labeled AST nodes to abstract types as captured by the cache $\hat{\mathcal{C}}$.

The first group of rules in Figure 11 have the form $\vdash c \uparrow^{\tau_\alpha}_{\tau} e_\alpha$ and describe how constants of type source $\tau$ may be transformed (or coerced) to abstract constants of target type $\tau_\alpha$. If there is no difference between the source and target types, then the transformation is the identity transformation. If there is a single abstract constant associated with a concrete integer constant then the transformation yields that abstract constant. Otherwise, a coercion expression is introduced to carry out the transformation during model-checking. Recall that boolean program elements are never abstracted, so the presented rules cover all possible cases for boolean constants.

*Constant Coercion*

$$\vdash c \uparrow^{\mathsf{Bool}}_{\mathsf{Bool}} c \qquad\qquad \vdash c \uparrow^{\mathsf{Int}}_{\mathsf{Int}} c$$

$$\vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} \underline{a} \quad \begin{array}{l}\text{if } \mathsf{Int} \neq \tau_\alpha\\ \text{where } \rightsquigarrow_\alpha ([\![c]\!]) = \{a\}\end{array} \qquad \vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} [\mathsf{Int} \ll \tau_\alpha]\, c \quad \begin{array}{l}\text{if } \mathsf{Int} \neq \tau_\alpha\\ \text{where } \rightsquigarrow_\alpha ([\![c]\!]) \neq \{a\}\end{array}$$

*Expression Coercion*

$$\frac{\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha}{\hat{\mathcal{C}} \vdash \bar{t}^{l} \Uparrow^{\tau_\alpha}_{\tau_\alpha} e_\alpha}$$

$$\frac{\vdash c \uparrow^{\tau_\alpha}_{\mathsf{Int}} e_\alpha}{\hat{\mathcal{C}} \vdash c^{l} \Uparrow^{\tau_\alpha}_{\mathsf{Int}} e_\alpha} \quad \text{where } \mathsf{Int} \neq \tau_\alpha \qquad \frac{\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha}{\hat{\mathcal{C}} \vdash \bar{t}^{l} \Uparrow^{\tau_{\alpha 2}}_{\tau_{\alpha 1}} [\tau_{\alpha 1} \ll \tau_{\alpha 2}]\, e_\alpha} \quad \begin{array}{l}\text{if } \tau_{\alpha_1} \neq \tau_{\alpha_2}\\ \text{and } \bar{t} \neq c\end{array}$$

*Expression Translation*

$$\frac{\vdash c \uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\tau} e_\alpha}{\hat{\mathcal{C}} \vdash c \Rightarrow e_\alpha} \quad \text{where } c \in \mathsf{Cons}[\tau] \qquad\quad \hat{\mathcal{C}} \vdash x \Rightarrow x$$

$$\frac{\hat{\mathcal{C}} \vdash \bar{t}_1^{\bar{l}_1} \Uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\hat{\mathcal{C}}(\bar{l}_1)} e_{\alpha 1} \quad \hat{\mathcal{C}} \vdash \bar{t}_2^{\bar{l}_2} \Uparrow^{\hat{\mathcal{C}}(\bar{l}_2)}_{\hat{\mathcal{C}}(\bar{l})} e_{\alpha 2}}{\hat{\mathcal{C}} \vdash o(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2})^{l} \Rightarrow o_{\hat{\mathcal{C}}(\bar{l})}(e_{\alpha 1}, e_{\alpha 2})} \qquad \frac{\hat{\mathcal{C}} \vdash \bar{t}_1^{\bar{l}_1} \Uparrow^{\hat{\mathcal{C}}(\bar{l})}_{\hat{\mathcal{C}}(\bar{l}_1)} e_{\alpha 1} \quad \hat{\mathcal{C}} \vdash \bar{t}_2^{\bar{l}_2} \Uparrow^{\hat{\mathcal{C}}(\bar{l}_2)}_{\hat{\mathcal{C}}(\bar{l})} e_{\alpha 2}}{\hat{\mathcal{C}} \vdash t(\bar{t}_1^{\bar{l}_1}, \bar{t}_2^{\bar{l}_2})^{l} \Rightarrow t_{\hat{\mathcal{C}}(\bar{l})}(e_{\alpha 1}, e_{\alpha 2})}$$

**Fig. 11.** Translating concrete programs to abstract programs (excerpts)

The second group of rules in Figure 11 have the form $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Uparrow^{\tau_{\alpha 2}}_{\tau_{\alpha 1}} e_\alpha$ and are similar in spirit to the rules above. If there is no difference between the source and target types, the result of the transformation is simply the result of recursive transforming the labeled term $\bar{t}^{\bar{l}}$. If a constant is being coerced, the constant coercion rules are used. On non-constant terms where the source type is different from the target type, a coercion is inserted after recursively transforming the argument of the translation.

The third group of rules in Figure 11 have the form $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha$. The constant coercion rules are used to transform a constant from its concrete type to a possibly abstract type. In the rules for operations and tests, the expression coercion rules are used to transform and possibly coerce the arguments. Then, the concrete operation is replaced by the abstract version indicated by the corresponding cache entry.

The remaining rules which are not displayed in Figure 11 are straightforward — remaining constructs such as conditions, returns, and gotos are preserved while transforming all subexpressions.

The rules of Figure 11 generate a syntactically correct abstract program (the proposition below captures this for expressions).

**Proposition 1 (Syntactically correct abstract expressions).**
*Let $\Gamma \vdash_\Sigma e : \tau$ and $\hat{\mathcal{C}} \vdash \bar{t}^{\bar{l}} \Rightarrow e_\alpha$ where $\hat{\mathcal{C}}$ is compatible with $\Gamma$, $\bar{t}^{\bar{l}}$ is the labeled version of $e$. Then $\Gamma_\alpha \vdash_{\Sigma_\alpha} e_\alpha : \hat{\mathcal{C}}(\bar{l})$ where $domain(\Gamma_\alpha) = domain(\Gamma)$ and $\forall x \in domain(\Gamma_\alpha) \,.\, \Gamma_\alpha(x) = \hat{\mathcal{C}}(x)$.*

Applying the translation rules of Figure 11 to the power program in Figure 6 with context $\hat{\mathcal{C}}'$ from Section 5 gives an abstracted power program shown on the right in Figure 9.

In the definitions that follow, when $\Gamma_\alpha$ arises from $\Gamma$ due to the program abstraction process captured in the proposition above, we say that $\Gamma_\alpha$ is a abstract version of $\Gamma$.

We now consider some basic safety properties that we need to express the correctness of abstraction. If $\Sigma_\alpha$ and $A_\alpha$ represent the abstract signature and algebra generated from a basis $\Sigma$ and $A$, and $\Gamma_\alpha$ is an abstract version of $\Gamma$ built using a set $\alpha$ of abstractions, the safety relation between a $\Gamma$-compatible store $\sigma$ and a $\Gamma_\alpha$-compatible store $\sigma_\alpha$ (denoted $\sigma \lhd \sigma'$) holds iff for all $x \in domain(\Gamma) \,.\, \sigma(x) \rightsquigarrow_{\Gamma_\alpha(x)} \sigma_\alpha(x)$, i.e., the store values for each $x$ are related by the abstraction relation associated with $x$'s abstract type.

**Lemma 1.** *(Safety for expressions) Let $\Gamma \vdash_\Sigma e : \tau$ and let $\Gamma_\alpha \vdash_{\Sigma_\alpha} e_\alpha : \tau_\alpha$ be the abstract expression constructed by the type inference and program abstraction process described above. Let $\sigma \in [\![\Gamma]\!]^{A}_{\Sigma}$, $\sigma_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$, $v \in [\![\tau]\!]$, and $b \in [\![\mathsf{Bool}]\!]$:*

- *$\sigma \lhd \sigma_\alpha$ and $\sigma \vdash e \Rightarrow v$ implies $\exists v_\alpha \in [\![\tau_\alpha]\!]$ such that $\sigma_\alpha \vdash e_\alpha \Rightarrow v_\alpha$ and $v \rightsquigarrow_{\tau_\alpha} v_\alpha$*
- *$\sigma \lhd \sigma_\alpha$ and $\sigma \vdash e \Rightarrow b$ implies $\sigma_\alpha \vdash e_\alpha \Rightarrow b$*

**Lemma 2.** *(Safety for transitions) Let $\Sigma_\alpha$ and $A_\alpha$ represent the abstract signature and algebra generated from a basis $\Sigma$ and $A$, and let $\Gamma_\alpha$ be an abstract version of $\Gamma$ built using a set $\alpha$ of abstractions, then for every $\sigma, \sigma' \in [\![\Gamma]\!]^{A}_{\Sigma}$, and for every $\sigma_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$, and $n, n' \in \mathrm{Nodes[FCL]}$, $\sigma \lhd \sigma_\alpha$ and $(n, \sigma) \longmapsto (n', \sigma')$ implies $\exists \sigma'_\alpha \in [\![\Gamma_\alpha]\!]^{A_\alpha}_{\Sigma_\alpha}$ such that $(n, \sigma_\alpha) \longmapsto (n', \sigma'_\alpha)$ and $\sigma' \lhd \sigma'_\alpha$.*

Given these basic properties, the fact that a concrete program is simulated by its abstracted counterpart is established in a straightforward manner.

## 7 Generating Abstract Properties

When abstracting properties, we want to ensure that if an abstracted property holds for an abstracted program, then the original property holds for the original program. In order to achieve this goal, properties have to be *under-approximated.* This is the dual of the process of abstracting a program. A program is abstracted by over-approximating its behaviors, i.e., the abstracted program may contain more behaviors that are not present in the original program due to the imprecision introduced in the abstraction process. Thus, if the abstracted program satisfies a particular requirement, then we can safely conclude that the original program satisfies the requirement. When abstracting a property, however, the abstraction may introduce imprecision such that the abstracted property may allow more behaviors of the program that satisfies it. Thus, we only consider the cases where the abstracted property can precisely decide the original property, i.e., under-approximating it.

Property abstraction begins in Bandera by performing type-inference on and abstracting each expression $e$ in the property where property expressions are constructing following the grammar in Figure 7. Let $e$ be a property expression such that $\Gamma \vdash e : \mathsf{Bool}$ and $domain(\Gamma) = \text{Variables}[e]$ where $\text{Variables}[e]$ denotes the set of variables occurring in $e$. Furthermore, assume that $\Gamma_\alpha$ is an abstract version of $\Gamma$ and that $e_\alpha$ is an abstract version of $e$ (i.e., as generated by the transformation process described in the previous section where we have $\Gamma_\alpha \vdash e_\alpha : \mathsf{Bool}$).

Section 3.3 defined the semantics of expression propositions as an under-approximation (i.e., an expression is only considered to be true when it does not evaluate to false). Bandera represents this semantics by constructing explicitly a disjunctive normal form that encodes the cases of stored values that cause an expression proposition to be interpreted as *true*.

For an abstract property expression $e_\alpha$ such that $\Gamma_\alpha \vdash e_\alpha : \mathsf{Bool}$, we denote the set of $\Gamma_\alpha$-compatible stores that make $e_\alpha$ true as

$$TrueStores[\Gamma_\alpha](e_\alpha) \stackrel{\text{def}}{=} \{\sigma_\alpha \mid \sigma_\alpha \in [\![\Gamma_\alpha]\!] \textit{ and } \exists n.[\![e_\alpha]\!](n, \sigma_\alpha)\}.$$

Note that the semantics of expression propositions is independent of control points $n$.

Next, we denote a conjunction that specifies the bindings of a store $\sigma_\alpha$ as

$$Bindings(\sigma_\alpha) \stackrel{\text{def}}{=} \bigwedge\{=(x, a) \mid (x, a) \in \sigma_\alpha\}.$$

The following function $\mathcal{T}$ specifies the transformation that Bandera uses to generate abstracted properties (the transformation is structure preserving except for the case of proposition expressions which we give below).

$$\mathcal{T}(e_\alpha) = \bigvee\{Bindings(\sigma_\alpha) \mid \sigma_\alpha \in TrueStores[\Gamma_\alpha](e_\alpha)\}$$
$$\mathcal{T}(\neg e_\alpha) = \bigvee\{Bindings(\sigma_\alpha) \mid \sigma_\alpha \in TrueStores[\Gamma_\alpha](\neg e_\alpha)\}$$

For example, suppose that we want to abstract the property

$$\Box\neg[end] \lor \Box(\neg[init] \lor \neg{=}(\%(\mathtt{m}, 2), 1) \lor \Diamond([end] \land {=}(\%(\mathtt{result}, 2), 1))).$$

with `m` and `result` abstracted using the evenodd abstraction. After applying $\mathcal{T}$, the property becomes

$$\Box\neg[end] \lor \Box(\neg[init] \lor {=}(\mathtt{m}, even) \lor \Diamond([end] \land {=}(\mathtt{result}, odd))).$$

This is the case where the abstraction is precise enough to decide the original property.

However, suppose that now `m` is abstracted using the evenodd abstraction, and `result` is abstracted using the point abstraction. After applying $\mathcal{T}$, the property becomes

$$\Box\neg[end] \lor \Box(\neg[init] \lor {=}(\mathtt{m}, even) \lor \Diamond([end] \land \mathit{false})).$$

This is the case where an abstraction is not precise enough to decide a proposition, i.e., $=(\%(\texttt{result}, 2), 1)$ is under-approximated to *false*, because *point* is not precise enough. When submitted to a model checker, infeasible counter-examples would be generated as evidence of the imprecision. Various proofs of property under-approximation can be found in [25].

## 8 Related Work

There is a wide body of literature on abstract interpretation. In our discussions of related work, we confine ourselves to work on automated abstraction facilities dedicated to constructing abstract models suitable for model-checking from program source code or closely related artifacts.

The closest work to ours is that of Gallardo, et. al. [12] on *alpha SPIN* – a tool for applying data abstraction to systems described in Promela (the input language of SPIN [15]). Alpha SPIN collects abstractions in libraries and transforms both Promela models and properties following a strategy that is similar to Bandera's. Alpha SPIN does not include automated facilities such as those found in Bandera for deriving sound abstractions, finding appropriate program components to abstract using dependency information, nor automated support for attaching abstractions via type-inference.

A closely related project that focuses on data abstraction of C program source code is the work on the abC tool by Dams, Hesse, and Holzmann [7]. Rather than providing a variety of abstractions in a library, abC focuses on *variable hiding* – a conceptually simple and practically very useful form of data abstraction in model checking which amounts to suppressing all information about a given set of variables. abC uses an integrated demand-driven pointer analysis to deal effectively with C pointers, and it has been implemented as an extension of GCC. Functionality that is similar to what abC provides can be achieved using Bandera's slicing facility (which detects and removes irrelevant variables) and Bandera's Point abstraction. However, since abC is dedicated to variable hiding, it provides a more precise form of abstraction attachment (e.g., compared to Bandera's type inference) for pointer types.

The Automated Software Engineering group at NASA Ames has developed a flexible explicit-state model-checker Java Pathfinder (JPF) that works directly on Java byte-code [3]. JPF includes a number of interesting search heuristics that are proving effective in software model-checking. The Ames group has also produced a simple predicate abstraction tool and a distributed version of the model-checking engine. Due to the difficulties associated with dynamically created data, the JPF predicate abstraction tool applies to integer variables only and does not include support for automated refinement. In collaboration with researchers at NASA Ames, JPF has been incorporated as a back-end checker for Bandera.

The Microsoft Research SLAM Project [1] focuses on checking sequential C code using well-engineered predicate abstraction and abstraction refinement tools. As discussed in Section 1, the strengths of the SLAM abstraction tool compared to Bandera are its automated refinement techniques which can significantly reduce the effort required by the user of the tool. The tradeoffs are that such techniques are computationally more expensive than the "compiled abstraction" approach taken by Bandera, and they have not been scaled up to work with computational patterns often used in Java where programs iterate over dynamically created data structure.

The BLAST Project [28], inspired by the SLAM work, combines the three-steps of abstract-check-refine into a single phase. Like SLAM, BLAST also works on sequential C code, and tradeoffs between the BLAST and Bandera abstraction approach are the same as those between SLAM and Bandera.

Gerard Holzmann's Feaver tool extracts Promela programs from annotated C programs for checking with SPIN [15]. Feaver performs abstraction by consulting a user built lookup-table that maps textual patterns appearing the the source code to textual patterns that form pieces of the abstract program. This tool has been used in several substantial production telecommunications applications.

Eran Yahav has developed a tool for checking safety properties of Java programs [30] built on top of Lev-Ami and Sagiv's three-valued logic analysis tool (TVLA) [21].

## 9 Conclusion

We have given an overview of some of the technical issues associated Bandera's tools for constructing abstract models of Java software. These tools are based on classical abstract interpretation techniques [6], and aim to provide users with simple but effective mechanisms for generating tractable models suitable for verification using widely-applied model-checking engines. Bandera's abstraction techniques have been used effectively in case studies with researchers at NASA Ames involving checking properties of avionics systems.

The strength of the Bandera abstraction tools include their simplicity, their ability to scale to large programs, and the ease with which they can be applied to systems with dynamic allocation of data and threads. We believe the main contribution of our work is the integration of 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.

Weaknesses of the tool include the lack of automated refinement techniques and the lack of sophisticated heap abstractions. As noted earlier, work on projects such as SLAM [1] and BLAST [28] have demonstrated the effectiveness of automated refinement techniques when applied to sequential programs that do not manipulate dynamically created data. Scaling these techniques up to a language like Java is an open problem that could a long way toward addressing the lack of automated refinement techniques in Bandera. Sophisticated heap abstraction capabilities have been developed in work on shape analysis (e.g., the TVLA project [21]), but automated abstraction selection and refinement techniques have not be developed yet. Combining and scaling up the automated predicate abstraction refinement techniques and heap abstractions with automated refinement is a research direction that we are pursuing.

## References


1. T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2000.
2. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10th International Conference on Computer Aided Verification*, June 1998.
3. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
4. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
5. James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002. To appear.
6. 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.
7. G.J. Holzmann D. Dams, W. Hesse. Abstracting C with abC. In *Proc. 14th International Conference on Computer Aided Verification*, July 2002.
8. Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.
9. C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, September 1999.
10. Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
11. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
12. M. M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. aSPIN: Extending SPIN with abstraction. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 254–258. Springer-Verlag, 2002.
13. Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

14. John Hatcliff. An introduction to partial evaluation using a simple flowchart language. In John Hatcliff, Peter Thiemann, and Torben Mogensen, editors, *Proceedings of the 1998 DIKU International Summer School on Partial Evaluation*, Tutorials in Computer Science, Copenhagen, Denmark, June 1998.
15. G. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer-Verlag, 2000.
16. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
17. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 1999.
18. Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating Java for multiple model checkers: the bandera back end. Technical Report 2002-1, SAnToS Laboratory Technical Report Series, Kansas State University, Department of Computing and Information Sciences, 2002.
19. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
20. Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
21. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene-based static analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, 2000.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
23. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
24. 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.
25. Corina S. Păsăreanu. *Abstraction and Modular Reasoning for the Verification of Software*. PhD thesis, Kansas State University, 2001.
26. Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software : A comparative case study. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 16 80)*, September 1999.
27. Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, April 2001.
28. Rupak Majumdar Thomas A. Henzinger, Ranjit Jhala and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002.
29. Raja Valle-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, November 1999.
30. Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 27–40, January 2001.