A Language Framework For Expressing Checkable Properties of Dynamic Software*†

James C. Corbett
University of Hawai'i
Honolulu HI, 96822, USA
corbett@hawaii.edu

Matthew B. Dwyer, John Hatcliff, Robby
Kansas State University
Manhattan KS, 66506, USA
{dwyer,hatcliff,robby}@cis.ksu.edu

May 16, 2000

Abstract

Research on how to reason about correctness properties of software systems using model checking is advancing rapidly. Work on extracting finite-state models from program source code and on abstracting the those models is focused on enabling the tractable checking of program properties such as freedom from deadlock and assertion violations. For the most part, the problem of specifying more general program properties has not been considered.

In this paper, we report on the support for specifying properties of dynamic multi-threaded Java programs that we have built into the Bandera system. Bandera extracts finite-state models, in the input format of several existing model checkers, from Java code based on the property to be checked. The Bandera Specification Language (BSL) provides a language for defining general assertions and pre/post conditions on methods. It also supports the definition of observations that can be made of the state of program objects and the incorporation of those observations as predicates that can be instantiated in the scope of object quantifiers and used in describing common forms of state/event sequencing properties. We describe BSL and illustrate it on an example analyzed with Bandera and the Spin model checker.

1 Introduction

Several current projects [18, 4, 16, 1, 7] are aiming to demonstrate the effectiveness of model-checking as a quality assurance mechanism for software source code. Tools developed in these projects typically use one of the following two strategies: (1) they take program source code and compile it to the model description language of an existing model-checker (*e.g.*, Promela – the description language of the Spin model-checker [17]), or (2) they use a dedicated model-checking engine to process a model that is derived on-the-fly from source code. Both of these strategies have relative strengths and weakness when it comes to attacking what we call the *model construction problem*: the challenging problem of taking real-world software components built with sophisticated language constructs that give rise to gigantic state-spaces and, from these, forming correct compact representations suitable for tractable model-checking.

Our software model-checking project, called Bandera, adopts a generalization of the first strategy: it provides an environment for compiling finite-state models from Java source code into several existing model-checking engines including Spin, dSpin [8], and NuSMV [3]. The environment includes numerous optimization phases including (a) a program slicing phase to remove program components that are irrelevant for the property being model-checked and (b) an abstraction phase to reduce the cardinality of data sets being

*This work supported in part by NSF under grants CCR-9633388, CCR-9703094, CCR-9708184, and CCR-9701418 and DARPA/NASA under grant NAG 21209.

†KSU CIS Technical Report 2000-2

manipulated. Just like in a conventional compiler, the translation process is carefully engineered through various intermediate languages so that adding a new back-end model-checker requires relatively little work.

Besides attacking the model-construction problem, we are also devoting significant resources to what we call the *requirement specification problem*: the problem of providing powerful yet easy to use mechanisms for expressing temporal requirements of software source code. Within the context of the Bandera project, one option would be to have developers to express requirements in the specification language of one of the underlying model-checkers (e.g., in Spin's Linear Temporal Logic (LTL) specification language, or Computational Tree Logic (CTL) of NuSMV). This has several disadvantages.

1. Although temporal logics such as LTL and CTL are theoretically elegant, practitioners and even researchers sometimes find it difficult to use them to accurately express the complex state and event sequencing properties often required of software. Once written, these specifications are often hard to reason about, debug, and modify.
2. In the Bandera project, following this option forces the analyst to commit to the notation of a particular model-checker back-end. If more than one checker is to be used, the specifications may have to be recoded in a different language (*e.g.*, switching from LTL to CTL), and then perhaps simultaneously modified if the requirement is modified.
3. This option is also problematic because it forces the specification to be stated in terms of the *model's representation of program features* such as control-points, local and instance variables, array access, nested object dereferences as rendered, *e.g.*, in Promela, instead of in terms of the source code itself. Thus, the user must understand these typically highly optimized representations to accurately render the specifications. This is somewhat analogous to asking a programmer to state assertions in terms of the compiler's intermediate representation. Moreover, the representations may change depending on which optimizations were used when generating the model. Even greater challenges arise when modeling the *dynamism* found in typical object-oriented software: components corresponding to dynamically created objects/threads are dynamically added to the state-space during execution. These components are *anonymous* in the sense that they are often not bound directly to variables appearing in the source program. The lack of fixed source-level component names makes it difficult to write specifications describing dynamic component properties: such properties have to be expressed in terms of the model's representation of the heap.
4. Finally, depending on what form of abstraction is used, the specifications would have to be modified to reference *abstractions of source code features* instead of the source code features themselves.

In this paper, we describe the design and implementation of the Bandera Specification Language (BSL) — a source-level, model-checker independent language for expressing temporal properties of Java program actions and data. BSL addresses the problems outlined above and provides support for overcoming the hurdles one faces when specifying properties of dynamically evolving software. For example, consider a property of a bounded buffer implementation stating that *no buffer stays full forever*. There are several challenges in rendering this specification in a form that can be model-checked including

- defining the meaning of *full* in the implementation,
- quantifying over time to insure that full buffers eventually become non-full, and
- quantifying over all dynamically created bounded buffers instances in the program.

BSL separates these issues and treats them with special purpose sub-languages as indicated in Figure 1.

- An *assertion* sublanguage allows developers to define constraints on program contexts in familiar assertion-style notation. Assertions can be selectively enabled/disabled so that one can easily identify only a subset of assertions for checking. Bandera exploits this capability by optimizing the generated models (using slicing and abstraction) specifically for the selected assertions.
- A temporal property sublanguage provides support for defining *predicates* on common Java control points (*e.g.*, method invocation and return) and Java data (including dynamically created threads and

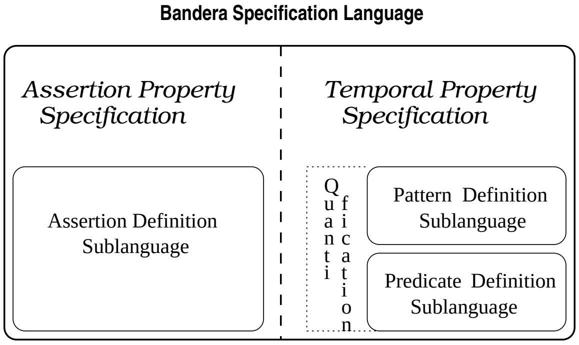

Figure 1: BSL organization

objects). These predicates become the basic *propositions* in temporal specifications. The temporal specification language is based not on a particular temporal logic, but on a collection of field-tested temporal specification patterns developed in our earlier work [13]. This pattern language is extensible and allows for libraries of domain-specific patterns to be created.

Interacting with both the predicate and pattern support in BSL is a powerful quantification facility allows that temporal specifications to be quantified over all objects/threads from particular classes. Quantification provides a mechanism for *naming* potentially anonymous data, and we have found this type of support to be crucial for expressive reasoning about dynamically created objects.

Assertions and predicates are included in the source code as Javadoc comments. This allows for HTML-based documentation to easily be extracted and browsed by developers, and for special purpose doclet-processing to provide a rich tool-based infra-structure for for writing specifications.

Even though BSL is based on Java, and, to some extent is driven by the desire to abstract away from the collection of model-checker back-ends found in Bandera, we believe that the general ideas embodied in this language will be useful in any environment for model-checking software source code.

The rest of the paper is organized as follows. Section 2 discusses some of the salient issues that influenced our design of the language. Section 3 presents a simple Java program that we use to illustrate the features of the language, and gives a brief overview of how Bandera models concurrent Java programs. Section 4 describes the design and implementation of the assertion sublanguage. Section 5 describes the design and implementation of the temporal specification sublanguage. A brief discussion of the implementation of BSL in Bandera and its application to check properties of an example is given in Section 6. Section 7 discusses related work and Section 8 describes a variety of extensions to the current implementation of BSL that are under development and concludes.

2 Design Rationale

BSL is the latest in a series of languages and systems that we have created to support specification of temporal properties of software systems. Experience with an earlier simple property language for Ada software [11] and with a system of temporal specification patterns [13] has yielded the following design criteria that we have tried to follow as we address the challenging issues surrounding specifying properties of Java source code.

1. The specification language must hide the intricacies of temporal logic by emphasizing common specification coding patterns.
2. Even though details of temporal logics are to be hidden, expert users should be allowed "back doors" to write specifications (or fragments of specifications) directly in, e.g., LTL.

3. The language must support specification styles that are already used by developers such as assertions, pre/post-conditions, etc.
4. The specification artifacts themselves should be strongly connected to the code. The specifications will reference features from the code (e.g., variable names, method names, control points), so the principal specification elements should be located with the code so that developers can easily maintain them, read them for documentation, and browse them.
5. Finally, and perhaps must importantly, the language must include support for reasoning about language features used in real software such as dynamic object/thread creation, interface definitions, exceptions, etc. These often give rise to heap-allocated structures that are reachable by following chains of object references, but are effectively anonymous when considering a Java program's static fields and local variables.

Criterion (1) derives from experience with the Specification Patterns Project. BSL provides a macro facility that implements all the patterns from [13]. Conforming to criterion (2), it also allows expert users to write their own patterns or code their temporal logic specifications directly.

Following criterion (3), the design and presentation of BSL has been influenced by various Java assertion languages such as iContract [20]. In each of these languages above, developers write their specifications in Java comments using the Javadoc facility. We have also followed this approach because it is an effective way to address criterion (4): browse-able HTML documentation for BSL specifications can be created, and the close physical proximity of comments and code encourages regular maintenance.

BSL currently does not address all the program features mentioned in criterion (5) (*e.g.*, we don't discuss specifying properties of exceptions in this work), but it does make significant steps in handling heap-allocated data/threads.

3 Example

Figure 2 gives the implementation of a simple bounded buffer implementation in Java that is amenable to simultaneous use by multiple threads. This code illustrates several of the challenges encountered in specifying the behavior of Java programs. Each instance of the `BoundedBuffer` class maintains an array of objects and two indices into that array representing the `head` and `tail` of the active segment of the array. Calls to `add` (`take`) objects to (from) the buffer are guarded by a check for a full (empty) buffer using the Java condition-`wait` loop idiom. Comments in the code contain various BSL declarations, and we will discuss these in the following sections.

Bandera models a concurrent Java program as a finite-state transition system. Each state of the transition system is an abstraction of the state the Java program and each transition represents the execution of one or more statements transforming this abstract state. The state of the transition system records the current control location of each thread, the values of program variables relevant to the property being checked, and run-time information necessary to implement the concurrent semantics: the run state of each thread, the lock states, and the lock and wait queues of each object. We bound the number of states in the model by limiting the number of objects each allocator (e.g., `new ClassName`) can create to k (a parameter of the analysis).

4 The Assertion Sublanguage

4.1 Rationale

An assertion facility provides a convenient way for a programmer to specify a constraint on a program's data space that should hold when control reaches a particular control location. In C and C++ programming, assertions are typically embedded directly in source code using an `assert` macro, where the location of the assertion is given by the position of the macro invocation in the source program.

```
/**
 * @observable
 *  EXP Full:  (head == tail);
 *  EXP TailRange:  (tail >= 0 &&
 *                   tail < bound);
 *  EXP HeadRange:  (head >= 0 &&
 *                   head < bound);
 */
class BoundedBuffer {
 Object [] buffer;
 int bound, head, tail;

 /**
  * @assert
  *  PRE PositiveBound:  (b > 0);
  */
 public BoundedBuffer(int b) {
  bound = b;
  buffer = new Object[bound];
  head = 0;
  tail = bound - 1;
 }

 /**
  * @observable
  *  RETURN ReturnTrue:
  *            ($ret == true);
  */
 public synchronized
        boolean isEmpty() {
  return head ==
         ((tail + 1) % bound);
 }
```

```
 /**
  * @assert
  *  POST AddToEnd:
  *         (head == 0) ?
  *         buffer[bound-1] == o :
  *         buffer[head-1] == o;
  * @observable
  *  INVOKE Call;
  */
 public synchronized
        void add(Object o) {
  while ( tail == head )
   try { wait(); }
   catch (InterruptedException ex) {}
  buffer[head] = o;
  head = (head+1) % bound;
  notifyAll();
 }

 /**
  * @observable
  *  RETURN Return;
  */
 public synchronized Object take() {
  while ( isEmpty() )
   try { wait(); }
   catch ( InterruptedException ex) {}
  tail = (tail+1) % bound;
  notifyAll();
  return buffer[tail];
 }
}
```

Figure 2: Bounded Buffer Implementation with Predicate Definitions

Due to Java's support for extracting HTML documentation from Java source code comments via Javadoc technologies, several popular Java assertion facilities, such as iContract [20], support definition of assertions in Java method header comments. BSL also adopts this approach. For example, Figure 2 shows the declaration of the BSL assertion `PRE PositiveBound: (b > 0)`. In this assertion, the data constraint is `(b > 0)` and the control location is specified by the occurrence of the tag `@assert PRE` in the method header documentation for `BoundedBuffer` constructor: the constraint must hold whenever control is at the first executable statement in the constructor.

4.2 Syntax and informal semantics

Figure 4 gives the syntax of BSL assertions. Sets of assertions are defined using the Javadoc tag `@assert` in the header documentation for methods or constructors. Each assertion set defined by an `@assert` tag can be given an optional name, and this name along with the name for each assertion in the set, is used to uniquely identify the assertion so that it be can be selectively enabled/disabled. If the set name is omitted, the fully qualified name of the corresponding method is used as the name for the assertion set. The optional name is followed by zero or more Java newline comments.

Besides precondition assertions as illustrated above, BSL supports location assertions and postcondition

```
import assertion BoundedBuffer.*;
import predicate BoundedBuffer.*;

// Enable PositiveBound pre-cond.  assertion of BoundedBuffer
BufferAssertions:  enable assertions { PositiveBound, AddToEnd };

// Indices always stay in range
IndexRangeInvariant:  forall[b:BoundedBuffer].
                        {HeadRange(b) && TailRange(b)} is universal globally;

// Full-buffers eventually get emptied
FullToNonFull:  forall[b:BoundedBuffer].
                  {!Full(b)} responds to {Full(b)} globally;

// Empty-buffers must be added to before being taken from
NoTakeWhileEmpty:  forall[b:BoundedBuffer].
                     {take.Return(b)} is absent
                     after {isEmpty.ReturnTrue(b)} until {add.Call(b)}
```

Figure 3: Bounded Buffer Properties rendered in BSL

```
<assertions> ::= @assert <assertion-set-name>? <comment>* <assertion>*

<assertion-set-name> ::= <java-id> | <assertion-set-name> . <java-id>
<label>              ::= <java-id>
<assertion-name>     ::= <java-id>

<assertion> ::= PRE <assertion-name> : <exp> ; <comment>*
              | LOCATION '[' <label> ']' <assertion-name> : <exp> ; <comment>*
              | POST <assertion-name> : <exp> ; <comment>*
```

Figure 4: BSL assertion syntax

assertions. `LOCATION[<label>] <assertion-name>: <exp>` is satisfied if `<exp>` is true when control is at the Java statement labeled by `<label>` in the corresponding method).[1] `POST <assertion-name>: <exp>` is satisfied if `<exp>` is true immediately *after* the execution of any of the corresponding method's `return` statements or after the execution of the last statement in the method if it has no return statement. The expression `<exp>` can refer to the return value of the method using the Bandera reserved identifier `$ret`.

There are various other well-formedness conditions associated with variable scoping that we will not discuss in detail here. For example, a precondition assertion cannot reference local variables for the method since these have not be initialized when the byte-code for the method body begins executing, and a label assertion can only reference variables are in scope at the point where the given label appears in the source code.

Once assertions have declared, a selection of the assertions can be presented to Bandera as an *assertion specification* to be model-checked. Figure 3 presents a BSL file where the first specification `BufferAssertions` enables checking of the `PositiveBound` and `AddtoEnd` assertions of Figure 2. Violated assertions are reported back to the user by presenting a trace through the source program that terminates at the location in which the data condition is violated.

[1]Even though Java does not include `goto`'s, it includes labels to indicate the targets of `break` and `continue` statements.

4.3 Implementation issues

As with other Java assertion tools, the BSL assertion implementation acts a preprocessor: it transforms the source code and embeds each enabled assertion using a Bandera library method `Bandera.assert(boolean)`. A little bit of extra work is needed to maintain proper label correspondence in `LOCATION` assertions and to calculate the value of the variable `$ret` in `POST` assertions, but the transformation is otherwise straightforward. One can also hardcode assertions directly with `Bandera.assert`, but this is discouraged. When Bandera generates models for Spin, each `Bandera.assert` call is translated to a Promela `ASSERT` statement.

5 The Temporal Specification Sublanguage

While assertions provide a convenient way to write constraints on data at particular program points, they are not powerful enough to directly specify interesting temporal relationships between system actions. Since such temporal properties are often required of concurrent systems, model-checkers usually support a temporal property specification language based on, e.g., LTL or CTL. These temporal specification languages subsume assertions in the sense that any assertion (l, c), where l is a location and c a condition on the data at that location, can be encoded in a temporal property: along all paths, it must be true in every state s that if s's control point is l then c holds. However, model-checkers, such as Spin, provide support for assertions because they can be checked much more efficiently than general temporal properties. To take advantage of the potential for faster checks, BSL separates assertion and general temporal properties.

In this section, we describe BSL's temporal specification sublanguage. First, we introduce a system of predicates on program features including common control points such as method entry and exit, and both class and instance data. These predicates become the primitive propositions that one reasons about when writing temporal property specifications. We describe our extensible pattern-based system for constructing temporal specifications. Woven throughout both the predicates and the temporal patterns is a notion of object quantification that provides a mechanism for generating names of dynamically created objects.

5.1 Predicate Definition Sublanguage

BSL provides two kinds of predicates : location insensitive predicates—predicates that are used for defining observables regardless of program points, and location sensitive predicates—predicates that are used for defining observables at specific program points. For example, Figure 2 shows a declaration of a location insensitive predicate `EXP Full: (head == tail)` in the class `BoundedBuffer` header documentation. This form of predicate, called an *expression predicate*, is often used to define class invariants or to indicate distinguished states (*e.g.*, a full buffer) in class or instance data. Since expression predicates do not refer to particular control points in methods, they can only be defined in class header documentation.

In addition to categorizing predicates based on location sensitivity, we also categorize predicates based on the kinds of fields or code to which they refer. *Static predicates* are used to reason about `static` fields (class variables) or program points of `static` Java methods. *Instance predicates* are used to reason about instance fields (memory cells that appear in each object of a given class) or program points in Java virtual methods. For example, the `Full` predicate is an instance predicate, because it refers to instance data members of class `BoundedBuffer`. The `static` modifier is used in an expression predicate declaration to indicate that it is a static predicate. When an instance predicate is used in a specification, it must be passed a parameter to indicate the instance to which the predicate applies. For example, the `FullToNonFull` property of Figure 3 shows the `Full` predicate being parameterized by the quantified variable `b`.

5.1.1 Syntax and informal semantics

Figure 5 gives the syntax of BSL predicate definitions. Sets of predicates are defined using the Javadoc tag `@observable` in the header documentation for classes or methods. Each predicate set defined by an `@observable` tag can be given an optional name, and this name along with the name for each predicate

```
<predicates> ::= @observable <predicate-set-name>? <comment>* <predicate>*

<predicate-set-name> ::= <java-id> | <predicate-set-name> . <java-id>
<label>              ::= <java-id>
<predicate-name>     ::= <java-id>

<predicate> ::= static? EXP <predicate-name> : <exp> ; <comment>*
              | INVOKE <predicate-name> [: <exp>]? ; <comment>*
              | LOCATION '[' <label> '] ' <predicate-name> [: <exp>]? ; <comment>*
              | RETURN <predicate-name> [: <exp>]? ; <comment>*
```

Figure 5: BSL predicate definition syntax

in the set, is used to uniquely identify the predicate so that it can be referred to in a temporal property specification. If the set name is omitted, the fully qualified name of the corresponding class or method is used as the name for the predicate set. The optional name is followed by zero or more Java newline comments.

Besides expression predicates as illustrated above, BSL supports invocation predicates, location predicates, and return predicates, which are all location sensitive predicates that are defined in method header documentation. These location sensitive predicates are static if they are defined for static methods.

`INVOKE <predicate-name> [: <exp>]?` is true when control is at the first executable statement in the corresponding method and `<exp>` is true; absence of `<exp>` defaults to true. For instance `INVOKE` predicates, an additional constraint applies: the supplied object parameter must match the *receiver object* (*i.e.*, the object upon which the method was invoked). For example, in the `NoTakeWhileEmpty` property of Figure 3, the instance `INVOKE` predicate `add.Call` holds only when the `add` method is invoked on the object referenced by the quantified variable `b`.

The semantics of `LOCATION '[' <label> ']' <predicate-name> [:<exp>]?` is similar to that of the invoke predicate, except that the relevant control point is the Java statement labeled by `<label>` in the corresponding method. `RETURN <predicate-name> [:<exp>]?` is similar to an invoke predicate, except that the relevant control points are the points immediately *after* any of the corresponding method's `return` statements or after the last statement in the method if it has no return statement. The expression `<exp>` can refer to the return value of the method using the Bandera reserved identifier `$ret`. For example, in the `NoTakeWhileEmpty` property of Figure 3, the instance `RETURN` predicate `ReturnTrue` holds iff the `isEmpty` method was invoked on the object referenced by the quantified variable `b` and control is immediately after the `return` statement of `isEmpty` and the return value of the method is `true`.

The Java expressions that are supported in `<exp>` are Java expressions that are guaranteed to have no *side-effects*. Currently we restrict Java expressions to exclude assignments, array or object creations, and method invocations to assure side-effect freedom.

There are various other well-formedness conditions associated with variable scoping that we will not discuss in detail here. For example, a return predicate cannot refer to local variables of the method since there might be several return statements for the method and each having a different set of visible local variables.

5.1.2 Implicit Constraints

Since the execution of certain Java/BSL operators can throw run-time exceptions (e.g., NullPointerException), expressions containing such operators are conjoined with implicit constraints prohibiting such exceptions (which might otherwise interfere with the model checking). For example, the predicate expression `x.f` is interpretted as `(x != null) && x.f`—if `x == null`, the predicate is false. If static analysis can determine that such exceptions will not be thrown, then the constraints can be omitted.

5.2 Specifying Temporal Patterns

The automata and temporal-logic formalisms that are commonly used to describe state and event sequencing properties of concurrent systems can be subtle and difficult to use correctly. Even people with significant experience with temporal logics find it difficult to specify common software requirements in their formalism of choice. Consequently it is a significant challenge to make these formalisms amenable to effective use by practicing software developers.

To address this issue, in previous work [10] we identified a small number of commonly occurring classes of temporal requirements, e.g., invariance, response, and precedence properties. We refer to these classes as *specification patterns*; there are five basic patterns:

- *universal* properties require the argument to be true throughout the execution
- *absence* properties require that the argument is never true in the execution
- *existence* properties require that the argument is true at some point in the execution
- *response* properties require that the occurrence of a designated state/event is followed by another designated state/event in the execution
- *precedence* properties require that a designated state/event always occurs before the first occurrence of another designated state/event

In addition several *chain* patterns allow for the construction of sequences of dependent response and precedence relationships to be specified. A web-site [12] presents the current set of eight patterns and their variations as well as translations into five different common temporal specification formalisms, including LTL and CTL.

In developing this system of patterns we found that it was useful to distinguish the required pattern of states/events from the region of system execution in which this pattern was required. *Pattern scopes* define variations of the basic patterns in which checking of the pattern is disabled during specified regions of execution. There are five basic scopes; a pattern can hold

- *globally* throughout the system's execution,
- *after* the first occurrence of a state/event,
- *before* the first occurrence of a state/event,
- *between* a pair of designated states/events
- during the interval, or *after* one state/event *until* the next occurrence of another state/event or throughout the rest of the execution if there is no subsequent occurrence of that state/event

In subsequent work [13], we validated that these specification patterns were representative of a large majority of the temporal requirements that researchers and users of finite-state verification tools had written. We studied over 600 property specifications and found that over 94% were instances of the patterns; interestingly over 70% of the properties were either universal or response properties.

We believe that there are several advantages to a pattern based approach. It can shorten the learning curve by presenting a smaller collection of concepts to specification writers. These patterns are expressible in nearly all of the commonly used specification languages for existing finite-state verification tools, thus, patterns provide some degree of tool independence. Finally, techniques for optimizing the construction of finite-state models can exploit information about the structure of the patterns.

BSL builds off of this work by providing a structured-English language front-end for the patterns which is illustrated in the fourth and fifth rule sets of Figure 6. Common synonyms are supported. For example, *invariance* and *universal* patterns and *leads to* can be used to express the `FullToNonFull` property from Figure 3 as

```
<temporal>   ::= <id> : <quantifier>* <pattern>

<quantifier> ::= forall [ <ids> : <java-id> ] .

<ids>        ::= <id>
               | <ids> , <id>

<pattern>    ::= <pred-expr> is universal <scope>
               | <pred-expr> is invariant <scope>
               | <pred-expr> is absent <scope>
               | <pred-expr> exists <scope>
               | <pred-expr> precedes <pred-expr> <scope>
               | <pred-expr> leads to <pred-expr> <scope>
               | <pred-expr> responds to <pred-expr> <scope>

<scope>      ::= globally
               | before <pred-expr>
               | after <pred-expr>
               | between <pred-expr> and <pred-expr>
               | after <pred-expr> until <pred-expr>

<pred-expr>  ::= <predicate-name>
               | <predicate-name> ( <id> )
               | ( <pred-expr> )
               | ! <pred-expr>
               | <pred-expr> && <pred-expr>
               | <pred-expr> || <pred-expr>
               | <pred-expr> -> <pred-expr>
```

Figure 6: BSL Pattern and Quantifier Syntax

```
FullToNonFull: forall[b:BoundedBuffer].
  {Full(b)} leads to {!Full(b)} globally
```

These pattern specifications are then translated into the specification formalism for the chosen checker. For example, if using Spin the *leads to* part of this property would be translated to the following LTL

```
[]( {Full(b)} -> <> {!Full(b)} )
```

Bandera supports user defined extension of BSL patterns. Users can define their own patterns and mappings to low-level specification formalisms. In this way, an expert specifier can customize a collection of patterns for the use of the developers on a project.

5.3 Specifying Properties for Class Instances

One of the chief difficulties in specifying properties of Java programs lies in naming the objects that constitute the state of the system. The majority of program state is heap allocated and is referenced through local thread variables by navigating chains of object references. We believe that, in general, one is interested in stating properties about all instances of a class or by distinguishing instances of a class by observing their state. One way to achieve this is to provide the ability to state properties for all instances of a class created during a program run.

The syntax for universal *class instance quantification* is given in Figure 6. BSL provides this through a mechanism that is independent of the specific checker used to reason about the property defined in the scope of the quantifier. This is achieved by customizing the model based on the quantifiers used and by embedding the property to be checked in a specification pattern that assures it will be checked only over the objects in the quantifier's domain.

For clarity, we describe the customization of the model in terms of a source program transformation rather than as a transformation on Bandera's intermediate program representation. This transformation requires the use of non-determinism in the model, which is not available in Java, so we introduce a static method `Bandera.choose` that is translated by Bandera to non-deterministic choice among its arguments in the model checker input. We also describe the embedding of a quantified LTL property; the approach is similar for other formalisms.

5.3.1 Universal Class Instance Quantification

Universal quantification is achieved as follows, for a quantified formula `forall[var:QuantifiedClass].P(var)`

1. For the bound variable `var` in the quantification, introduce a static field `var` of type `QuantifiedClass` in the public class `BoundVariables`. This will introduce a *global* state variable for each such field in the finite-state model for the program.

2. For each field in `BoundVariables` define a predicate `var_selected` which is true when `var` is not null.

3. At the end of each constructor for `QuantifiedClass` introduce the following code fragment:

```
if (BoundVariables.var == null)
  if (Bandera.choose(true,false)) BoundVariables.var = this;
```

   where `Bandera.choose(true,false)` introduces a non-deterministic choice between `true` and `false` values in the finite-state model for the program.

4. The temporal formula, `P`, to be checked on the generated model is modified in two ways. First, the pattern, `P`, in the scope of the quantifier, is expanded using the name `var` as parameters to the referenced predicates; call this expanded formula `P_var`. Second, this expanded formula is embedded in a context which controls the sequences of states on which `P_var` is evaluated. Specifically, for LTL, the expanded formula has the form:

```
(!var_selected U (var_selected && P_var)) || []!var_selected
```

Checking the modified formula on the modified model exploits the exhaustive nature of model checkers. For each trace of the unmodified model that would be generated by Bandera, the modified model creates a trace in which each instance of the `QuantifiedClass` will be bound to `var`. At the state in which the binding is established the modified temporal formula will trigger the checking of `P_var`.

Note that when nested quantification is used the `var_selected` condition is defined such that it is true only when all of the bound variables in the quantifiers have been assigned a non-null value.

5.3.2 Implementation Issues

There are several advantages to implementing support for quantification early in the process of extracting finite-state models from Java programs : checker independence and optimization.

The technique described above is applied to the internal representation of Java code prior to the generation of model checker input, e.g., Promela for Spin. Thus, quantified temporal specifications can be checked with any of the supported verifiers. Furthermore, since it is possible to generate Java from our internal

Property	Sliced	Never-claim States	States Stored
Deadlock	No	-	240047
Deadlock	Yes	-	51757
BufferAssertions	No	-	280575
BufferAssertions	Yes	-	17797
IndexRangeInvariant	Yes	14	45215
IndexRangeInvariant, BufferAssertions	Yes	14	115387
FullToNonFull	Yes	25	64687
FullToNonFull, BufferAssertions	Yes	25	154842

Figure 7: BoundedBuffer Property Check Data

representation, it is possible for Java model checkers, such as Java Pathfinder 2 from NASA's Ames Laboratory, to check quantified temporal specifications as long as they map the `Bandera.choose` method calls to non-deterministic choice in the underlying model.

The scopes in specification patterns define the end-points of regions in which the pattern should be checked. If those end-points do not occur, then the specification is vacuously true. For this reason, by performing object flow analyses [14] we can determine the set of objects that can possibly influence the satisfaction of scope delimiting predicates. This information can be used to restrict the set of objects over which a quantifier must range. Consider the specification

```
forall[s:SuperType].
  {Pred(s)} is absent after {init.Return(s)}
```

in this case we need only calculate the set of instances of `SuperType` for which the `init` method is called. An upper approximation of this set can by calculated in terms of the sub-types of `SuperType` that can appear as the receiver object of an `init` invocation. The code from step 3, described above, need only be added to those sub-types; this may significantly reduce the cost of checking the quantified property by reducing the number of values that will be assigned to the bound variable, `s`.

6 Implementation and Preliminary Results

BSL support has been implemented in the latest version of the Bandera toolset [4]. The user interface provides a significant advance over the previous interface for specifying program properties [5] by extracting assertions and predicates and presenting the latter as building blocks for instantiating temporal pattern specifications.

To illustrate the benefits of selectively analyzing program properties we built a simple environment for the `BoundedBuffer` class. The environment consists of four threads : a main thread that instantiates two `BoundedBuffers` loads one of the buffers until it is full, then passes the pair of buffers to threads that read from one buffer and write to the other such that a *ring* topology is established. An additional thread repeatedly polls the state of the two `BoundedBuffers` to determine if they are empty. With this environment all of the properties in Figure 3 will be true, yet the buffers will be forced through all of their internal states given their `bound`.

Figure 7 shows the size of the state space for checks of several properties in Figure 3. We only used Bandera's slicing capabilities in these checks; Bandera can also abstract program data to extract compact models. Slicing is driven by the expressions used in assertions and predicates in the properties being checked and is fully automatic. We omit timing data noting only that the total time to perform these checks was less than a minute in all cases (including slicing, model extraction, and model checking). By default, Bandera will extract a model suitable for deadlock checking. We used SPIN's deadlock checking ability for the first two checks; slicing has a modest impact on the state space since most of this small program can influence the execution of Java locking operations and `wait` statements. The `BufferAssertions` checks were obtained by enabling assertion checking with SPIN; slicing has a dramatic effect here since the program dependences [15] from the assertion expressions allow the thread that polls for emptiness to be sliced away. The last four checks illustrate that the overhead of BSL's quantification is not prohibitive. We only show data for sliced models, the non-sliced models are significantly larger. It is interesting to note that the embedding of the temporal formula to be checked in the formula from step 4 in Section 5.3.1 causes a non-trivial increase in the size of the automaton (never-claim) used in checking the property. For `IndexRangeInvariant`, the basic invariant property requires a 2 state automaton and this increases to 14 states with quantification. For `FullToNonFull`, the response property requires a 4 state automaton and this increases to 25 states with quantification. It is difficult to assess the impact that this blowup has on the cost of checking properties since we optimize the model for each property. We are studying this question empirically to understand the cost of quantification over a broader collection of programs and properties.

7 Related Work

There is a long history of work on temporal logics for specifying properties of concurrent systems. LTL [22] is one of the most popular such logics. Efforts to make such logics easier to use have taken several directions including developing diagrammatic forms [2, 9] and using stylized English [6, 13]. To our knowledge, there has been no published work on the specific problems associated with the application of temporal logics to specifying the behavior of program code.

In recent years, increasing attention has been devoted to formally specifying properties of design or object models, such as those supported in the unified modeling language (UML). UML's object constraint language (OCL) [23] provides for specification of program invariants and pre/post conditions on operations. Weaknesses of OCL-based approaches include the lack of ability to reason about such specifications and the fact that current tool support does not provide a mechanism to transfer to such specifications to the code. Both of these are being addressed by different research efforts. Alloy [19] is an object-modeling language that has an associated tool for checking specifications written about the model. Tools that support design-by-contract for Java programs such as iContract [20], support a subset of OCL that has been tailored to Java syntax. Such tools instrument the program to dynamically check invariants and method pre/post conditions. Our work fits in this context by pushing OCL-like Java-based specifications in several directions to make them more checkable (ala Alloy) and to incorporate temporal specifications. The added benefit is that by exploiting model checking technology we can perform restricted checks for defects or exhaustive verification of properties; this makes our approach more suitable for the subtle defects that are found in multi-threaded Java systems.

The JML project [21] is attempting to provide a fully featured specification system for Java code. It provides for invariant, pre and post-conditions as do the approaches discussed above, but it aims to be enable users to completely describe the behavior of the code. In contrast, our work is aimed at supporting the description of partial correctness properties that can be automatically checked against code descriptions.

8 Conclusion

We have defined a language framework for expressing correctness properties of dynamic Java programs. A property expressed in this framework can be rendered in terms of commonly available features in the input languages of existing model checking tools. This framework is integrated into the Bandera system so that information about the property to be checked can be exploited to reduce the size of the program model. We have implemented the core functionality of BSL and are beginning to use it to check properties of Java code.

Work on the core of BSL has led us to pursue several extensions to allow for a wider range of properties to be expressed. One of the most interesting of these extensions is the application of our approach to class instance quantification, from Section 5.3.1, to quantify over the elements of an array. The basic idea is to use non-determinism to choose from a range of indices and then bind the array element at that index for use in subsequent temporal formulae. We believe that this can be applied to any container built using arrays, such as `java.util.Vector`. We are experimenting with these extensions to find the ones that add useful new capabilities to BSL and we plan to report on those in the future.

References

[1] Thomas Bal and Sriram K. Rajamani. Boolean programs : A model and process for software analysis. Technical Report 2000-14, Microsoft Research, 2000.

[2] I. A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. *Lecture Notes in Computer Science*, 1026, 1995.

[3] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000. to appear.

[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. Bandera : A source-level interface for model checking Java programs. In *Proceedings of the 22nd International Conference on Software E ngineering*, June 2000.

[6] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 179–190, October 1996.

[7] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, July 1999.

[8] 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.

[9] Laura K. Dillon, G. Kutty, Louise E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[11] Matthew B. Dwyer, Corina S. Pasareanu, and James C. Corbett. Translating Ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.

[12] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. A System of Specification Patterns. `http://www.cis.ksu.edu/santos/spec-patterns`, 1998.

[13] 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.

[14] David Paul Grove. *Effective Interprocedural Optimization of Object-oriented Languages*. PhD thesis, University of Washington, 1998.

[15] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, September 1999.

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

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

[18] Gerard J. Holzmann and Margaret H. Smith. Software model checking : Extracting verification models from source code. In *Proceedings of FORTE/PSTV'99*, November 1999.

[19] Daniel Jackson. Alloy: A lightweight object modelling notation.

[20] Reto Kramer. iContract—the Java Design by Contract tool. In *Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS-USA*. IEEE Press, 1998.

[21] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, October 1998.

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

[23] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.