The Bandera Tools for Model-checking Java Source Code: A User's Manual

John Hatcliff and Oksana Tkachuk

March 7, 2001

Contents

List of Figures

1 Introduction

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to allow experimentation with model-checking properties of Java source code. Bandera takes as input Java source code and a specification written 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, dSpin, SMV, JPF). 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.

This document describes the basic functionality of Bandera as presented by Bandera's graphical user interface. Section 2 describes the overall architecture of the tool set and summarizes the rationale and functionality of each component. Section 3 provides a brief description of how to install Bandera. Section 4 describes basic aspects of the Bandera User Interface (BUI). Section 5 gives a guided tour of the BUI and tool components using several different examples. Section 6 gives a detailed discussion of the Bandera Specification Language (BSL). Section 11 summarizes the subset of Java that is currently handled by Java, and provides a list of known bugs and limitations.

We assume that the reader is familiar with the general concepts of compiling and model-checking and possesses a basic understanding of temporal logics such as LTL [13] or CTL [1]. The reader *does not* need to know how to use a particular model-checker tool such as Spin or SMV, although familiarity with such a tool will aid in understanding of Bandera's internal processing.

For a short description of Bandera and motivation of its architecture see [2]. For an overview of temporal logic and model-checking techniques, see [11].

2 Overview

2.1 Challenges of model-checking program source code

Modern computing applications increasingly require concurrent/distributed software systems that are extremely reliable. Unfortunately, current software validation techniques, such as inspections and testing, are failing to provide high levels of assurance of correctness for these systems due to system size and complexity as well as the fundamental difficulties of reasoning about state/event sequences in concurrent behavior.

Model-checking techniques (now widely used for hardware verification) hold promise for establishing crucial behavioral properties of complex software because they can automatically check to see if an abstract finite-state transition system model of the software conforms to a given state/event sequence property. If the model fails to satisfy the property, the model-checker gives a counter-example — a path through the model's transitions that violates the property. This can be used to locate and correct the corresponding software defect.

Although it holds great promise, we believe that there are four problems that are currently preventing model-checking technology from being successfully applied to software.

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

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

- **The requirement specification problem:** the difficulty of expressing software requirements in the temporal specification languages of existing model-checking tools. Although model-checker property specification languages are built on theoretically elegant temporal logics, practitioners and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug.

  Moreover, model-checker specification languages are designed to state properties of mathematical models rather than software source code. Most software specifications include references to program features such as control-points (e.g., method entry/exit), local and instance variables, array access, nested object dereferences. However, current tools provide little or no support for the intricate mappings that are often required to bridge the gap between source code features and their corresponding model realizations. This means that the user is often forced to state the specifications in terms of the *model's representation of program features* such as coded *e.g.*, in Spin's Promela input language, 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.

- **The output interpretation problem:** When a property fails when checking large models (and software systems typically produce very large models), the counter-example traces produced by the checker can be hundreds or even thousands of steps long.[1] Manually matching up these counterexamples with source code is extremely tedious for several reasons. First, the length is quite long and it may require hours to walk through the trace. Second, the error trace is expressed in terms of the low-level, possibly highly optimized model representations. Thus, one has the reverse of the "representation gap" issue mentioned in the property specification problem: the analyst must understand the model's representation of complex program features in order to accurately project the model error trace back to the source level. Typically, one "step" in the source program may correspond to as many as ten steps in the low-level model representation.

Although Bandera provides multiple forms of tool support to address the problems above, we believe that each of these problems is significant and it is unclear at present exactly which technologies and forms of tool support are *best* suited for solving these problems. Thus, the ultimate aim of the Bandera project is not to provide "silver bullet" solutions to the problems above, but, instead, to provide several different forms of tool support along with an open infrastructure that allows for easier experimentation with new techniques.

2.2 When might someone be interested in learning about Bandera?

If you would like to automatically verify properties of Java source code, you should consider using Bandera to model-check these properties. The strength of model-checking is its ability to aid in detection of subtle software defects due to unanticipated interleaving of concurrently executing threads. One can also use model-checking to check assertions, pre/post-conditions, and simple data invariants. However, it must be remembered that model-checking will always work better for verifying control-related properties rather than data-related properties. The more data-related a property is, the larger the state-space becomes,

[1] For example, in a software model-checking experiment performed by researchers at NASA Ames and Honeywell, checking properties of the DEOS real-time operating system scheduler produced SPIN counterexamples over 2700 steps long [14].

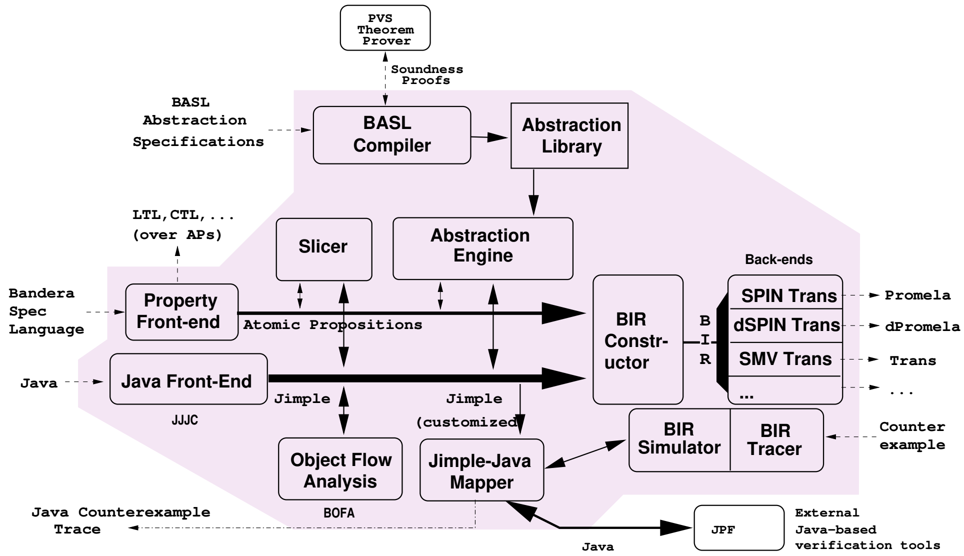

Figure 1: Internal architecture of the Bandera Tool Set

and thus resources required for model-checking. Model-checking has been used to verify data properties related to "container" objects like stacks and queues. For example, it can be used to show that the code for a stack does indeed maintain a LIFO ordering [4]. However, it seems impossible to use model-checking to verify that a sorting algorithm is correct since sorting correctness is a data-oriented property involving several quantifications and data structures. For verifying this type of property, other formal methods such as theorem proving would need to be used. Similarly, other verification problems such as checking for array-bounds errors, buffer overruns, and null-pointer dereference probably are best done using other forms of static dataflow analysis.

If you would like to slice and perform data abstraction on Java programs, you should consider using Bandera. Slicing and data abstraction are also useful techniques for analysis and testing. Since Bandera can carry out both slicing and abstraction as source-to-source transformations, you can bypass Bandera's back-end for model-checking and instead use transformed Java files as the input to, e.g., a testing tool.

If you have a model-checker that you would like to use to check Java source code, you should consider incorporating it into Bandera by writing a translation from BIR to the input language of the model-checker.

2.3 Architecture of Bandera

Figure 1 presents the internal architecture of Bandera.

Java infrastructure and intermediate representation: Bandera is built on top of the *Soot* Java compiler framework developed by Laurie Hendren's Sable group at the University of McGill [15]. In the Soot framework, Java programs are translated to an intermediate language called *Jimple*. Jimple was originally developed to be the target language of a Java decompiler (*i.e.*, the Soot tools provide a component for decompiling Java class files (byte code) to Jimple). Thus, Jimple is essentially a language of control-flow graphs where (a) statements appear in three-address-code form (the explicit stack manipulation inherent in JVM instructions has been removed by introducing temporary variables), and (b) various Java constructs such as method invocations and synchronized statements are represented in terms of their virtual machine counterparts (such as `invokevirtual`, and `monitorenter`, `monitorexit`).

Java front-end: Based on an initial prototype from the Soot group, the Bandera group has developed it's own *Java front-end* called JJJC (Java-to-Jimple-to-Java Compiler) that translates from Java to Jimple. JJJC also maintains data structures that are used by a *Jimple-Java Mapper* to move back and forth between Java and Jimple. For instance, JJJC can also decompile Jimple that has been transformed by Bandera's slicing and abstraction components back to Java. Thus, the Bandera slicer and abstraction tools can be viewed as source-to-source transformations. This is useful if one desires to use other verification tools that work at the Java source level in conjunction with Bandera. The Jimple-Java Mapper is also invoked during the process of mapping a model-checker counterexample back to the Java source level.

Property specification: Source code properties to be checked are written in the Bandera Specification Language (BSL). BSL is based on a collection of field-tested temporal specification patterns [7] that allow users to write specifications in a stylized English format. These patterns essentially are parameterized macros that can be instantiated to one or more temporal logics such as LTL or CTL. Thus, this pattern system addresses the specification problem mentioned in the previous section by providing the user with temporal structures commonly used in specifications.

BSL specifications are parameterized by *observables* (predicates on program state) that are defined in Java source code using Javadoc comment notation. From a foundational standpoint, BSL specifications are instantiated to assertions and temporal logic formulas, and user-declared observables are the primitive propositions that can appear in those formulas. The *property front-end* of Figure 1 calls the Java front-end to extract all the observables declared in the given source program, it type checks the declared observables, and it instantiates the BSL specification to a particular temporal logic, and it translates the observables used in the input specification to the lower-level model representation. This last step addresses the representation gap issue of the specification problem by automatically translating properties described in terms of source-level features to the low-level optimized model representation.

Static analysis: To enable further optimization and transformation, Bandera performs a number of common data/control analysis. In particular, the *Bandera Object Flow Analysis* (BOFA) statically collects information about the objects that may flow into each program expression at run-time. In a typical configuration, BOFA will associate a set of tokens $\{t_1, ..., t_n\}$ for a particular expression e where a token t_i is a pair (C, s) where C is a class name and s is the allocator site where the object was created. Thus, BOFA is similar to *points to* analyses used for convention imperative languages and *closure* analyses used for functional programming languages. BOFA is typically run in a context-insensitive flow-insensitive mode, but is designed to allow the precision to be increased along several different axes.

Approach to model construction: Bandera's approach to model construction is to generate one model for each property to be checked. This approach is based on the observation that, given a specific property ϕ, many parts of the software may not influence ϕ at all. This allows Bandera to employ optimizations and abstractions that remove program components irrelevant to ϕ and thus generate a highly compacted model. Note that this customization approach generally is infeasible when one is generating models from programs by hand. One might naively complain that generating one model per property incurs significant overhead. However, it is often the case that checking a particular property ϕ without customizing the model is infeasible due to the exponential cost of model-checking. In addition, Bandera's philosophy is to design the customization so that cost of customization is always dominated by model-checking (i.e., in practice, the time to customize should be shorter than the time to model-check).

Slicing: Bandera uses both *program slicing* and *data abstraction* (abstract interpretation) to customize models. The Bandera program slicer takes as input all the observables mentioned in the input property ϕ. These observables may reference particular program variables and control points. The semantics of these program features *must* be preserved for correctly checking ϕ, but all other program components that don't influence the semantics of the observable features can be eliminated in the generated model. The program slicer builds a *program dependence graph* representing several different forms of dependence, and it will generate an executable residual program (the program slice) where components that do not influence the execution of the observables in ϕ have been removed. The sliced program is created at the Jimple level, but it can also be decompiled to Java source using JJJC. For more details, we refer the reader to formalizations of the Bandera slicer's approach to property-driven slicing [10], and the notions of program dependence required for slicing the concurrent features of Java [9].

Abstract interpretation: The Bandera abstraction components provide automated support for reducing model size *via* data abstraction. This is useful when a specification ϕ to be checked does not depend on the program's concrete values but instead depends only on *properties* of those values. For example, an application might store a set of items in a vector, but if the property being verified depends only on whether a particular item is in the vector, we could abstract the large number of vector states onto a small set $\{ItemInVector, ItemNotInVector\}$.

The user guides the abstraction process by binding variables to entries from an *abstraction library.* The library entries are indexed by concrete type, and each entry implements an abstract version of its corresponding concrete type. Each abstraction in the library is defined using the Bandera Abstraction Specification Language (BASL). A BASL specification for a base type abstraction consists of a declaration of a finite set of abstract tokens, an abstraction function that maps each concrete Java value to an abstract token, and an abstract operation for each operation of the concrete type. A rule-based format that incorporates pattern matching simplifies the definition of abstract operations. For base type abstractions, the BASL compiler allows the user to supply an abbreviated BASL specification containing only the abstract token set and abstraction function. It then uses the PVS theorem prover to automatically synthesize the abstract versions of operations such as `+`, `-`, etc. This makes it extremely easy to define new abstractions on base types. Given a (possibly synthesized) BASL specification, the BASL compiler generates a Java class containing methods that implement the defined abstraction function as well as abstract versions of the relevant concrete operators. The Java class is held in the library, and is linked with rest of source code during model-generation if the user has selected that particular abstraction.

Since abstractions are incorporated on a per variable basis, two different variables of the same concrete type can have different abstract types. For example, if I_1 and I_2 are both `int` abstractions, then variable `int x` may be bound to I_1 and variable `int y` may be bound to I_2. After the user has chosen abstractions for a few key variables that are relevant to the property being checked, a type inference phase propagates this information throught the program and infers abstraction types for the remaining variables and for each expression in the program. Type inference also informs the user when there is an abstraction type error.

Once abstract type inference is run, the abstraction engine will transform the source code into a specialized version where all concrete operations and tests on the relevant objects (e.g., method calls on the vector class) are replaced with abstract versions that manipulate tokens representing the abstract values $\{ItemInVector, ItemNotInVector\}$. Since information is lost in this transformation, operations and tests that relied on the lost information can no longer be determined completely in the abstract program. For instance, in the vector example the length of the abstracted vector cannot be determined. Any conditional test that consulted the vector's length would be transformed into a non-deterministic choice between the true and false branches. As with slicing, the abstracted code is represented at the Jimple level, but JJJC can decompile it back to Java.

Back end model generation: The Bandera back end is like a code generator, taking the sliced and abstracted program and producing verifier-specific representations for targetted verifiers. The back end components communicate through BIR, the Bandera Intermediate Representation, an intermediary between compiler-based representations and verifier-based representations. As shown in Figure 1, the back end has one fixed component called BIRC (Bandera Intermediate Representation Constructor) that accepts a restricted form of Jimple and produces BIR. For each supported verifier, there is also a translator component that accepts the program represented in BIR and generates input for that verifier. Currently, translators for SPIN, dSPIN, and SMV are complete, and translators for FDR2, XMC, and Stanford's forthcoming SAL model checker are under construction.

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

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

3 Installation

3.1 Requirements

Machine requirements:

1. Minimum Processor: Intel Pentium III 600 MHz or equivalent.
2. Memory: 128 MB (minimum), 256 MB or more is recommended.
3. Hard drive free space: 60 MB (minimum). May need some space in user's home directory for logging and custom files, typically less than 1 MB.

Supported operating systems:

- Windows^{TM} 95/98/Me
- Windows^{TM} 2000
- Windows^{TM} 4.0
- Sun OS 5.6 or 5.7
- Linux with kernel 2.2.14 or better.

We are aware that Java programs can be run on any platform with Java compiler. Although we endeavor to make Bandera run in all Java-supporting platforms, we have not yet tested it in platforms other than those mentioned above.

We currently use Sun's standard JDK version 1.3 for Bandera development. We are aware that there are many Java compilers. Even if it claims compatibility with the standard JDK, we are not certain that Bandera would compile or run. Earlier versions of standard JDK are also not recommended.

Bandera supports JPF, Spin, dSpin, and NuSMV for this release. However none of these checkers are bundled with the distribution — you need to separately install those model checkers if you intend to use Bandera with them.

For Spin[2], we expect that the main model checker filename is `spin`. For SMV[3], it should be `NuSMV`. For DSpin[4], it should be `dspin`. Please make sure that all of these are installed in the directories mentioned in the path. To use JPF[5] with Bandera, simply put it with all its supporting packages into the same directory with Bandera.

Make sure that you have your C compiler and C preprocessor properly installed and their directory is mentioned in the path. This is necessary because the Spin model checker produces C files that have to be preprocessed and compiled. We assume the C compiler is GNU's [6] `gcc` and that the C preprocessor is GNU's `cpp`. The `gcc` and `cpp` versions are not specific, but our testing shows that version 2.91 or above is fine. For Windows environment, DJGPP's [7] `gcc` is just working fine.

[2]http://netlib.bell-labs.com/netlib/spin/whatispin.html
[3]http://sra.itc.it/tools/nusmv/
[4]http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml
[5]http://ase.arc.nasa.gov/visser/jpf
[6]http://gcc.gnu.org/
[7]http://www.djgpp.com

3.2 Installing

Bandera is distributed in jar format. It should be installed into a directory with the command: `jar xvf bandera-x.y.jar`, where `x.y` is the version number. For example: `jar xvf bandera-0.1.jar` The following descriptions assume that you will have unpacked the distribution in a directory named `BANDERA`.

3.2.1 UNIX

Assuming that you have the Java SDK installed in `/usr/bin/local/java` set the `CLASSPATH` as follows and export it:

```
CLASSPATH=BANDERA:usr/bin/local/java/jre/lib/rt.jar:usr/bin/local/java/lib/tools.jar:$CLASSPATH
```

You need to include the current directory as your `PATH`, e.g., `PATH=.: ...`

3.2.2 Windows ^{TM} 95/98/Me

Add the following line to your `AUTOEXEC.BAT` file:

```
SET CLASSPATH=C:\BANDERA;C:\JDK1.3\JRE\LIB\rt.jar;C:\JDK1.3\LIB\tools.jar;%CLASSPATH%
```

This assumes that you unzipped Bandera in `C:\` drive and that the JDK is installed in `C:\JDK1.3`. Don't forget to reboot your machine before using Bandera. We assume your familiarity of Windows operating system when installing Bandera. If you are not sure, please consult your system administrator.

3.2.3 Windows ^{TM} NT 4.0 or Windows ^{TM} 2000

Set the classpath as follows:

Right-click "My Computer" icon then choose "Properties". Open the "Environment" tab. In "System Variables" box, look for "ClassPath" variable. Highlight it. At the bottom of the dialog box, there is a "Value" input text box. Add the following text in front of the text box:

```
C:\BANDERA;C:\JDK1.3\jre\lib\rt.jar;C:\JDK1.3\LIB\tools.jar;
```

If you do not have ClassPath variable in your system, you can add it by typing `ClassPath` in the "Variable" input text box at the bottom and type the above text in "Value" input text box. Please delete the last semicolon.

This assumes that you unpacked Bandera in `C:\` and the JDK is installed in `C:\JDK1.3`. We also assume your familiarity of Windows operating system when installing Bandera. If you are not sure, please consult your system administrator.

3.3 Package Contents

Upon unpacking, Bandera should create `bandera` directory under which the following files and directories are present:

`Bandera.java`	Main program file
`Bandera.class`	Main program class file
`COPYING`	License notice
`README.html`	Release information
`filelist.txt`	Complete list of distribution files
`ca`	McGill University Soot Framework
`doc`	Documents, tutorials and examples (downloaded separately)
`edu`	All programs reside here
`java_cup`	Necessary files for parsing

For the complete list of files, please consult `filelist.txt` file.

3.4 Compiling

Bandera is distributed along with the class files. So, you don't have to recompile it. However, you could compile it by invoking Java compiler: `javac Bandera.java`, after you have set the classpath as mentioned previously. You should invoke the compiler in Bandera's installed directory if you want to recompile it. Please note that recompiling is absolutely not recommended for the current release.

3.5 Running and The First Launch

After everything has been set up, invoke `java Bandera` to run Bandera in UNIX. For Windows ^{TM}, you can open your Command Prompt or MS-DOS Prompt window, and invoke `java Bandera`.

Upon the first launch, Bandera will initialize itself. This involves in creating `.bandera` directory which will reside in user's "home directory". The table below describes the location of the home directory depending on the operating system.

Operating System	`.bandera` Location
UNIX variant	`~/.bandera`
Windows^{TM} 95/98/Me	`C:\WINDOWS\Profiles\(username)\.bandera`
Windows^{TM} NT 4.0	`C:\WINNT\Profiles\(username)\.bandera`
Windows^{TM} 2000	`C:\Documents and Settings\(username)\.bandera`

We assume that for Windows^{TM} environment, the Windows^{TM} is installed in `C:\`. You should replace the `(username)` with the appropriate user login name. In some variants of Windows^{TM} 95/98, the `.bandera` directory location is placed in `C:\WINDOWS\` directory.

Inside the `.bandera` directory, we have a preferences file `.banderarc`, an abstraction file `Abstractions`, and three directories `integral`, `real`, and `log`.

The preferences file is meant to store user preferences, e.g. directory locations and colors. Later versions of the tool will add more functionality. We currently have no GUI to modify these values. You can hack the color values if you wish, but manually modifying the other entries of this file is highly discouraged.

The `Abstractions` file is used by Abstraction Manager to store user's custom abstraction library. The directories `integral` and `real` are used to categorize these files. However, this functionality is not fully supported in the current version of Bandera.

The `log` subdirectory is used to store all log files upon Bandera's failure. Please send all these logs along with their related files when you submit a bug report. Whenever you need a spare disk space, you can safely delete these log files.

If you experience some problems in running Bandera, you can try to delete `.bandera` directory and re-run Bandera. This is usually due to the growing requirements for Bandera setup settings.

4 Overview of the Bandera User Interface

Invoking Bandera brings up the main window of the Bandera User Interface (BUI), and in this section we'll describe the function of the main window panes, tool-bar buttons, and pull-down menus.

4.1 The Project and Code Panels

Figure 2 displays the main window of the BUI with some example code loaded (we'll see how to load an example in Section 5). The main window contains two panels: the left panel is the *project panel* and the right panel is the *code panel*. The Project panel contains a tree that organizes the packages, classes, fields and methods of the Java software being analyzed.

- Left-clicking on a *package* node in the project panel expands the tree to show the classes in the package.

- Left-clicking on a *class* node in the project panel expands the tree to show the fields and methods in the class.

Figure 2: Bandera main window

- Left-clicking on a *field* node in the project panel shows the declaration of the field in the code window.
- Left-clicking on a *method* node in the project panel shows the code of the method in the code window.

It is important to note that the code panel does not display the actual text of the corresponding `.java` file. Rather, one should think of the display as providing the Java abstract syntax of the source code. When JJJC compiles a program, it creates a Java abstract syntax tree (AST) representation as well as the underlying Jimple intermediate representation. A system of internal *annotations* implemented using hash tables associates each node in the Java AST with its counterpart in the Jimple representation (and vice versa). The display in the code window is generated by dynamically decompiling the internal Jimple representation back to the corresponding Java using the annotations. This display strategy makes it easier to implement the visualization tools associated with counterexample traces, slicing, and abstraction. However, it has the disadvantage that textual features of the actual source code (*e.g.*, comments, indentation, etc.) are not shown in the window.[8]

Now we'll give a brief description of each tool-bar button and menu in the main Bandera window.

4.2 Sessions

Runs of Bandera are configured using *sessions*. A session is a record holding information about the file name(s) of the source code to be checked during the run, the property to be checked, the tool components

[8]We hope to display the actual text in future versions of Bandera.

that are to be enabled during the run, options and settings for the selected components, the particular back-end model-checker to be used, and other miscellaneous information such as location of working directories into which temporary output should be dumped.

Multiple session records are held in a *session file*. When performing a new run of Bandera, the session record can be saved in a session file and loaded at a later time. This allows the user to avoid restating all option information, etc. Session records in a session file can also be processed in batch mode using a command line flag. This is useful for performing regression tests on software under development. For example, you might consider creating a session file holding all the checks that you usually run on a piece of software, then using the batch mode facility to run all of the model-checks specified in the session file overnight.

The upper-left toolbar buttons are used for manipulating session files such as **New**, **Open**, **Save As**, and **Save**.

4.3 Component Buttons

The four buttons at the right side of the upper tool-bar (labeled **JJJC**, **Slicer**, **SLABS**, **Checker**) are used to enable/disable the four main components of Bandera. When a component is enabled, the corresponding button icon is presented in color with a check-mark. A non-checked black-and-white icon indicates that a component has been disabled. Clicking on the **Run** button causes the enabled components to be executed in the order in which they are presented in the tool-bar.

4.4 Session Manager

Starting on the lower toolbar, the first button opens the Session Manager window. The window is used for creating, modifying, and deleting of session records within the currently selected session file.

4.5 Property Specification Buttons

The next four buttons on the lower tool bar are associated with property specification.

- The **Assertion** button invokes the Assertion Browser which allows you to browse the assertions that you have declared in the code specified by the current session.

- The **Predicate** button invokes the Predicate Browser which allows you to browse the predicates (observables) that you have declared in the code specified by the current session.

- The **Pattern** button invokes the Pattern Manager. Recall that Bandera's temporal specification language is based on a collection of *temporal specification patterns*. The Pattern Manager allows the user to browse the patterns, and to add, modify or remove patterns. Typically, only expert users will change the collection of patterns.

- The **Property** button invokes the Property Manager which allows the user to specify properties to be checked for the code specified by the current session.

4.6 The PDG Browser

The next button in the lower toolbar invokes the PDG Browser. The PDG Browser allows the user to navigate the *program dependence graph* – the main data structure produced by the slicer. Several facilities are provided in the PDG Browser that aid in the selection of abstractions.

4.7 Abstract Type Inference

The next button in the lower toolbar invokes the Abstraction Manager which is used to bind abstract types to source code variables. The Abstraction Manager communicates this information about abstraction selection to the abstraction engine, and this information is used to drive the program transformation associated with abstraction.

4.8 Pull-down Menus

The functionality provided by the tool-bar buttons can also be found in pull-down menus **Session**, **Property**, **Abstraction**, and **Help**.

- The **Global** pull-down menu includes **General Options** option and the **Exit** option.
- The **Session** pull-down menu includes the session file operations operations **New**, **Open**, **Save as**, and **Save**, as well as the **Manager** option which invokes the Session Manager.
- The **Property** pull-down menu includes the **Pattern Manager** option, the **Predicate Browser** option, the **Assertion Browser** option, and the **Property Manager** option.
- The **Abstraction** pull-down menu includes the **Abstraction Manager** option.
- The **Help** pull-down menu includes the **About** option.

5 Guided Tour

This section illustrates the basic features of Bandera with some simple examples included in the distribution. First, we'll start by running some sessions that have already been defined. This will illustrate the code browser, slicing, and counterexample displays. Next, we'll step through the process of creating several new sessions. This will illustrate the process of using the property specification tool (Property Manager) and the abstraction component. The remaining sections of the tutorial will explain the components illustrated here in greater detail.

Before you can step through the examples below, the Spin model checker must be installed, and you'll need to copy the `tutorial` directory included with Bandera into your own directory hierarchy.

5.1 UNIX or Linux

Suppose that Bandera is installed in `/usr/local/bin/bandera`. To copy the examples directory, give the following commands.

```
mkdir ~/bandera-tutorial
cp -Rf /usr/local/bin/bandera/tutorial ~/bandera-tutorial
```

To invoke Bandera, type:

```
cd ~/bandera-tutorial
java Bandera
```

5.2 Windows^{TM}

Suppose Bandera is installed in `C:\Bandera`. To copy the examples directory into `C:\Tutorial`, open your Command Prompt or MS-DOS prompt window and type:

```
md C:\Tutorial
xcopy /e C:\Bandera\tutorial C:\Tutorial
```

Now, invoke Bandera as follows:

```
C:\
cd Tutorial
java Bandera
```

```
/************************************************************************
 *
 * B A N D E R A   T U T O R I A L   E X A M P L E S
 *
 * File: Deadlock.java
 *
 * Purpose: Illustrates the use of Bandera to check for deadlocks in
 * multi-threaded Java programs.  Illustrates the use of session files,
 * slicing, abstraction, and the counterexample (error trace display).
 *
 ************************************************************************/

public class Deadlock {
    static Lock lock1;
    static Lock lock2;
    static int  state;

    public static void main() {
        lock1 = new Lock();
        lock2 = new Lock();
        Process1 p1 = new Process1();
        Process2 p2 = new Process2();
        p1.start();
        p2.start();
    }
}

class Process1 extends Thread {
    public void run() {
        Deadlock.state++;
        synchronized (Deadlock.lock1) {
            synchronized (Deadlock.lock2) {
                Deadlock.state++;
            }
        }
    }
}

class Process2 extends Thread {
    public void run() {
        Deadlock.state++;
        synchronized (Deadlock.lock2) {
            synchronized (Deadlock.lock1) {
                Deadlock.state++;
            }
        }
    }
}

class Lock {
}
```

Figure 3: Simple Deadlock Example

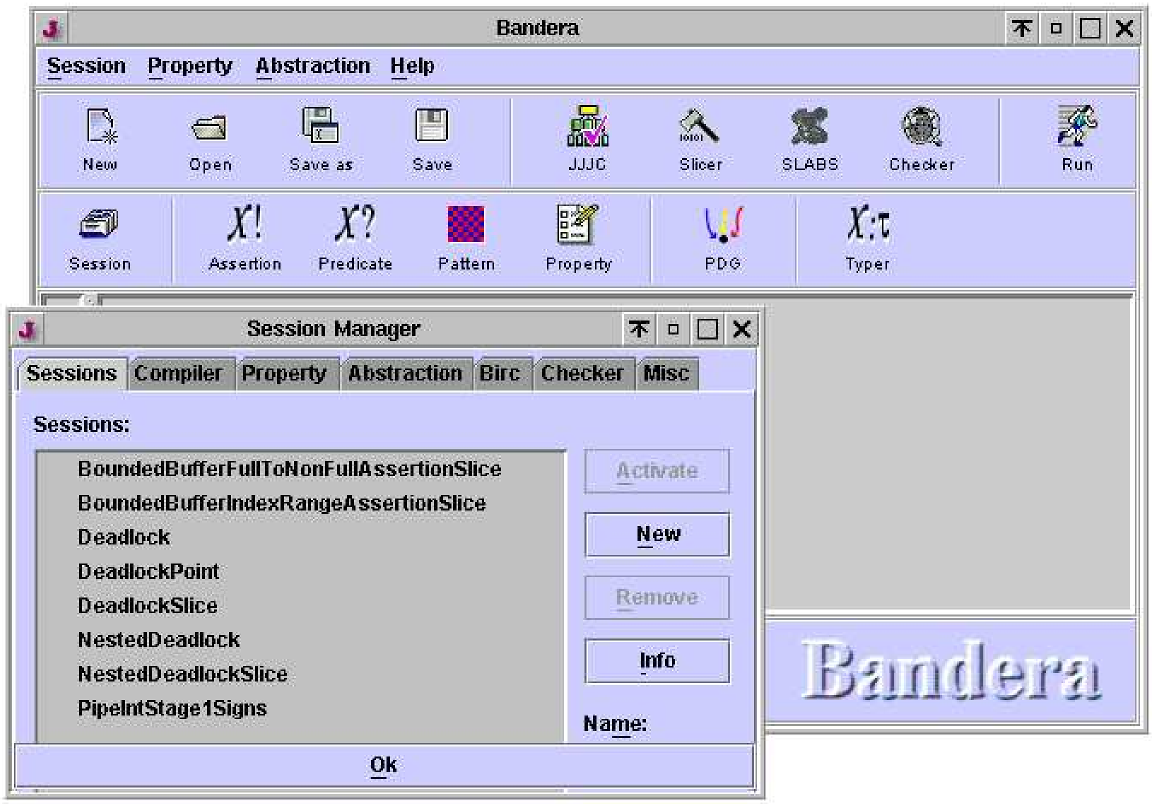

Figure 4: Main Frame and Session Manager Window

5.3 Simple Deadlock

Figure 3 displays the code for a simple multi-threaded Java program. The program's execution in the `main` method of class `Deadlock` proceeds as follows. Two `Lock` objects are created, and then thread objects of type `Process1` and `Process2` are created. Finally, the threads corresponding to `Process1` and `Process2` are `started` (this initiates the execution of the `run` method in `Process1` and `Process2`).

The `run` methods of `Process1` and `Process2` have a similar structure. Each `run` method first increments a variable called `state` that has been included to illustrate slicing and abstraction. Next, `run` of `Process1` tries to acquire the lock of `lock1` and then the lock of `lock2`, while `Process2` tries to acquire the lock of `lock2` and then the lock of `lock1`. There are some schedules for these threads that do *not* result in a deadlock – for example, if `Process1` successfully acquires both locks then releases them at the end of the `synchronized` block, and then `Process2` is free to acquire both locks. However, there are schedules which lead to a deadlock – for example, if `Process1` acquires `lock1` and then before it can acquire `lock2`, `Process2` acquires `lock2`.

We'll now use Bandera (with the Spin model-checker) to detect a deadlocking schedule.

- First, choose the **Open** option from the **Session** menu (or click the **Open** icon in the top tool-bar). This will bring up a file dialogue window. If you were in the appropriate directory when you started Bandera, you should see a file called `tutorial-examples.session` in the file dialogue window. Left-click on the file `tutorial-examples.session`.

- Now left-clicking the **Open** button in the file dialogue window will bring up the Session Manager window. You should now have two windows on your screen as shown in Figure 4.

- Select the session named `Deadlock`. This now gives the options of activating the session (making it the current session of Bandera), removing the session (deleting it from the session file), or bringing up information summarizing the settings of the session.

- Choose the **Activate** option. You can now see that the **JJJC** and **Checker** options are enabled in the top tool-bar (they changed from grey to color and they have a check-mark over the icon). You can also click the tabs (**Compiler**, **Property**, **Abstraction**, etc.) in the Session Manager window to see the various settings for the session. For example, under the **Checker** tab, note that Spin has been chosen as the back-end model-checker. If you wish, you can also control Spin's command-line options by clicking on the **Options** button beside of the check-box for Spin. Note the **Resource Bounded** option at the right bottom part of the **Spin Option** window. When checked, the **Resource Bounded** option will put bounds on the range of the resources specified by Birc constants under the **Birc** tab of the Session Manager. If your program violates the bounds set by Birc constants, the violation will be ignored and the model check will be performed on the model with the bounded resources. If you want to be notified of the resource bounds violation, uncheck the **Resource Bounded** option in the **Spin Option** window.

  It is important to note that if no temporal property or assertion has been specified, Bandera will automatically check for deadlock.

- Close the Session Manager window by clicking the **OK** bar at the bottom of the window. If you would press the tool-bar **Run** option now in the main window, Bandera would parse the `Deadlock.java` file, generate output for Spin, and invoke Spin to check the generated model. Before we do this however, we want to demonstrate the code browsing facilities of Bandera. Therefore, disable the checking engine by clicking on the **Checker** option in the top tool-bar of the main window (the **Checker** button should now turn grey).

- Now click the **Run** option to have Bandera load and parse the `Deadlock.java` code.

Now take some time to browse the code following the instructions given in the preceding section. For example, if you click on the default package node, then the switch for the `Deadlock` class, and then the `main` method, your main Bandera window should look like the window of Figure 5.

Now let's run a model-check. Click on the **Checker** tool-bar button to enable the selected checker, then press the **Run** button. Bandera will now crunch a way for a while. If you want to see (to some extent) what is happening behind the scenes, watch the contents of the terminal window that you used to invoke Bandera.

After Spin is finished checking the generated model, for this example Bandera will determine that an error-trace has been found. Two new windows should pop up on the screen as shown in Figure 6 (you may need to reposition the new windows to get them to appear as in Figure 6).

The window on the bottom right is the Counter-example Control window. At the bottom of this window are buttons that are used to navigate along the path of the counter-example (error trace). **Forward** steps one step forward in the error trace, **Backward** steps one step backward in the error trace, **Reset** goes to the beginning of the error trace.

During error trace display, Bandera creates a new code display window for each thread when the thread's run method is originally called. This allows the user to monitor the current position of the program counter within each thread. Sometimes, there are so many threads in the program that the screen becomes cluttered. In this case, you can "iconize" some of the of the thread windows. Clicking on the **Close** button in the Counter-example Control window closes all the windows associated with counter-example display (the control window is closed along with the code window for each thread).

- Now step forward along the trace by pressing the **Forward** button in the Counter-example Control window *twice*. As you do this, a green bar will highlight the current position of the program counter in the main Bandera code display window and also in the code display window for each thread. The program counter bar sits on the next statement to be executed (i.e., the colored statement has not been executed yet). Bandera allows you to examine the contents of the (abstracted) store (including heap values) by clicking on variables listed in the Store section of the Counter-example Control window. In particular, clicking on a variable of Object type allows you to see the thread

Figure 5: Code Browse

that is holding the object's lock and the threads that are waiting on the object's lock. Click on some of the variables to see their values. At this point in the trace, the lock objects show no threads waiting or holding the lock, and the `state` variable has a value of 0.

- Now step forward along the trace by pressing the **Forward** button in the Counter-example Control window until the program counter bar is at `p1.start()`. With the next push of the **Forward** button, Bandera will pop up the window for the `Process1` thread. Now the program counter bar sits at the first line in the `Process1` thread.

- Move forward two more steps until the program counter bar is at `synchronized (Deadlock.lock2)`. By this point, the `Process1` thread has acquired `lock1`. You can see this by clicking on `lock1` in the store pane of the Counter-example Control window. You can also see that the value of the `state` variable has been incremented (it was 0 before, now it should be 1).

- Move forward until the display indicates step 9 of 12, and the code display window for the `Process2` thread pops up. Move forward two more steps. The program counter bar in the window for `Process2` should be at the line `synchronized (Deadlock.lock1)`. Clicking on `lock2` in the store pane shows that `Process2` holds the lock for `lock2`.

- Moving forward one more step reaches the deadlock state. `Process2` tries to execute the statement `synchronized (Deadlock.lock1)` but cannot complete it because `Process1` holds the lock.

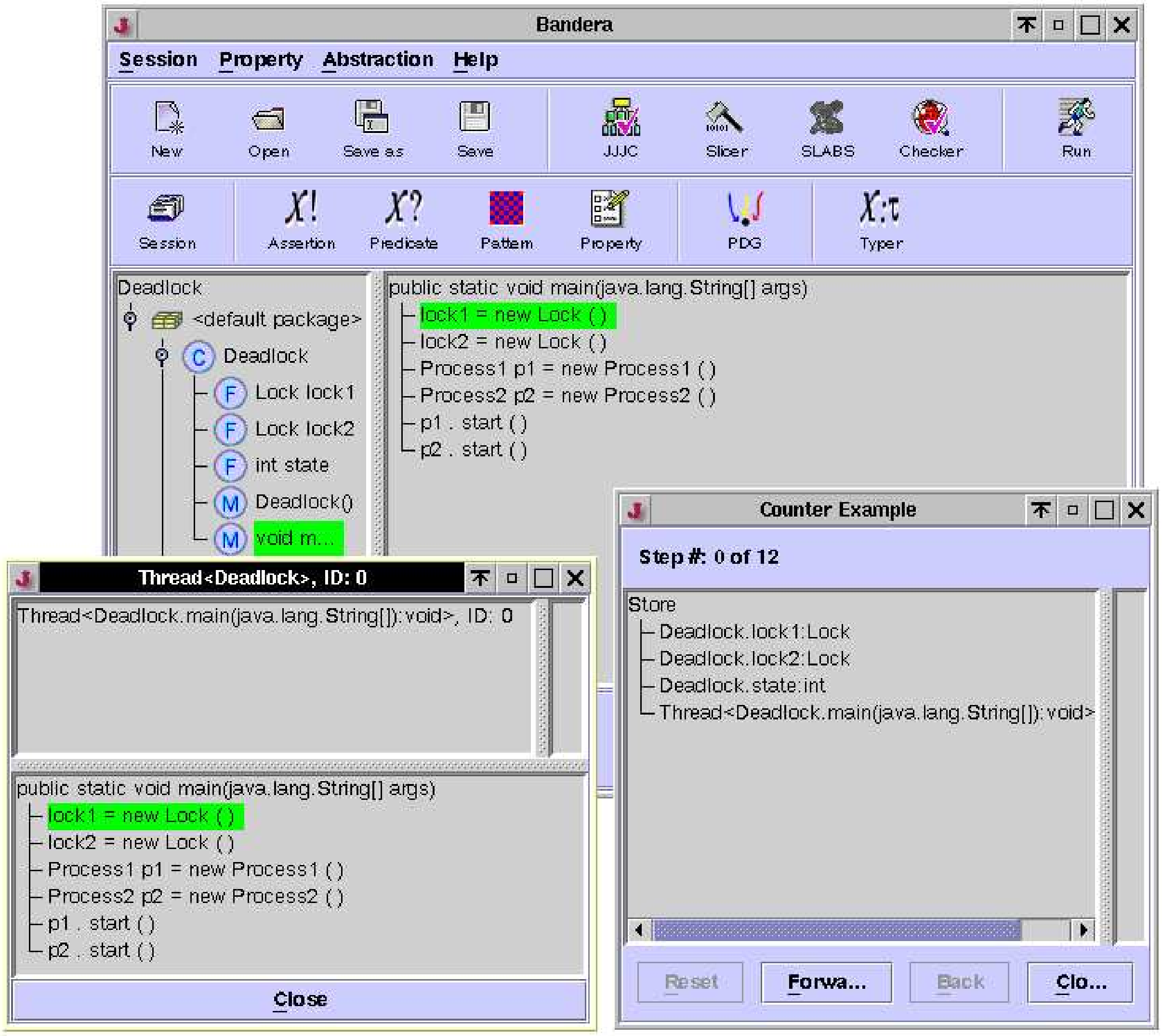

Figure 6: Error Trace

- Now, you can move backward along the counterexample if you wish. When you have finished experimenting with the display, press the **Close** button in the Counter-example Control window.

5.4 Simple Deadlock with Slicing

Now you are ready to experiment with the slicer. We will look at the same Deadlock example as in the previous section.

To work with the ready-to-use session that enables the slicer, click on the Session Manager tool bar button (the left-most button on the lower tool-bar (labelled **Session**)). Now, select and activate the session called `DeadlockSlice`. Notice how after activating the session, **JJJC**, **Slicer**, and **Checker** options on the top tool-bar change color indicating that those options are enabled.

Close the Session Manager window. At this point you can run the session by clicking on **Run** option on the top tool-bar or you can take a few steps to take a closer look at the slicer. Let's disable the **Checker** for a moment and run the session without it.

After you click on the **Checker** option on the top tool-bar to disable it, click on the **Run** option. Bandera will compile and parse the `Deadlock.java` again. If you watch the window where Bandera was invoked you will see the output of the slicer that begins with words: Building Program Dependence Graph. If you invoke the slicer when a property has not been specified, Bandera will slice for deadlock detection. When slicing for deadlock, Bandera will collect all the constructs in the program associated with blocking/waiting (e.g., the enter/exit monitor statements associated with synchronized methods and

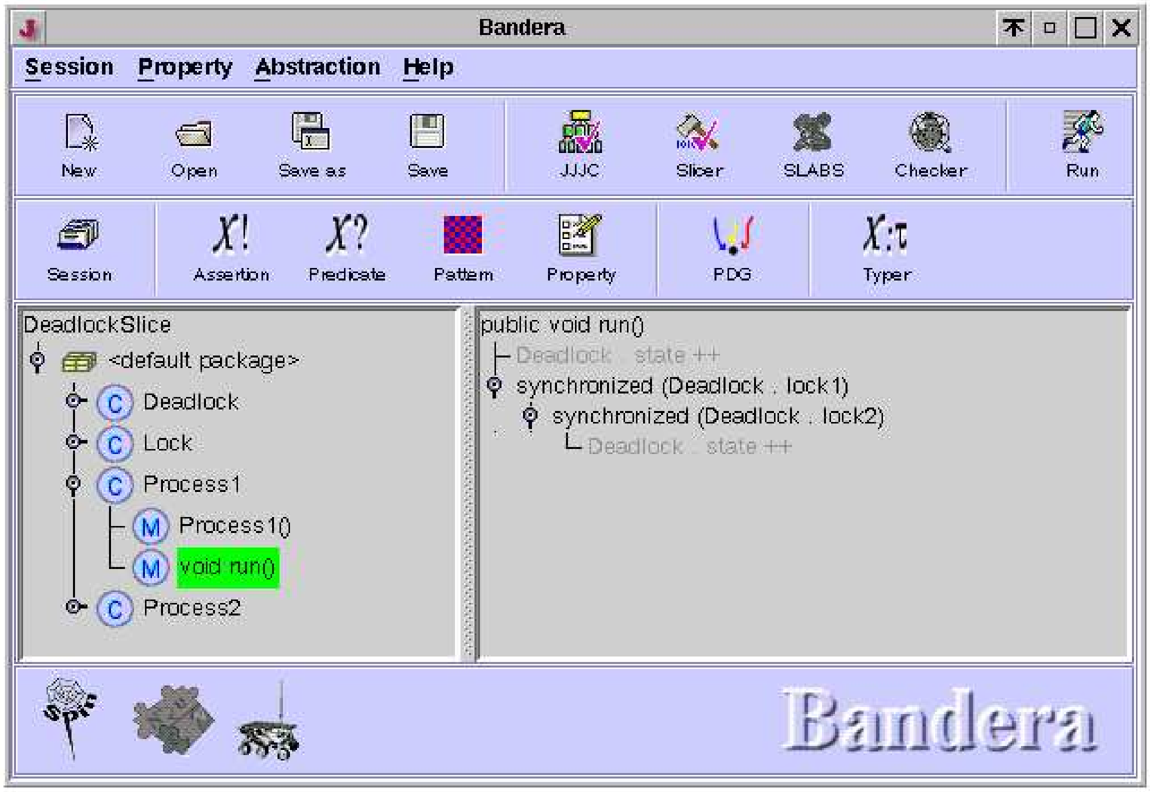

Figure 7: Deadlock Sliced

wait and notify statements) and these form the *slicing criterion*. The resulting slice will consist of all the other statements in the program that may possibly influence the statements in the criterion.

To see the sliced code for the deadlock example in the BUI, click on the default package node in the left panel of the main frame. Display the code for the `run` method of one of the processes by expanding the `run` method of, let's say, `Process1` class. You should have a picture in front of you as shown in Figure 7. Although no classes were eliminated in the slicing phase, in the `run` method of `Process1` (and `Process2`), you can see that the slicer eliminated the update to the variable `Deadlock.state` (code that the slicer removes is displayed in red, the user can change the color settings in the `.banderarc` file in the `.bandera` directory created after the invokation of Bandera as explained in Section 3) since this variable's value does not influence the blocking behavior of the program.

Now you can finish running this session by enabling the **Checker** option and clicking on the **Run** option. As before, you will see the Counter-example windows pop up. You can step through the counterexample as before.

Note that while in the previous session the counter-example trace had 12 steps, in this session the counter-example trace has only 10 steps. The slicer eliminated some of the irrelevant to the deadlock property statements and therefore made the error trace shorter as the program counter bar always steps over the statements that have been sliced away. This shows how the slicer can contribute to making the model more compact and the error trace shorter and therefore easier to analyze.

5.5 Pipeline Example

This next example will demonstrate how to create a new session, how to specify a property to be checked, and how to use abstractions to extract a smaller model out of the program to be model checked.

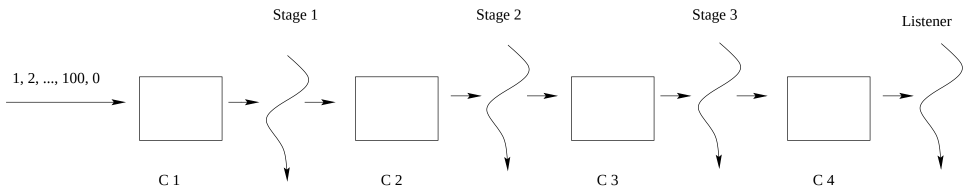

Figure 8: Pipeline Structure

1. Write an informal English statement of the property (we will refer to this as the *requirement*).
2. Examine the requirement to see which particular source code features (control points, variable values) are explicitly mentioned or alluded to in the requirement.
3. Use Bandera's Java-doc-based *predicate definition language* to declare the identified source code features to be *observable* to Bandera's property specification tool. These observables form the primitive propositions that can appear in the temporal property that we will ask Bandera to check.
4. Use the BUI's property manage to construct the temporal property the formalizes the given requirement.

Figure 9: Methodology of property specification supported by Bandera

5.5.1 The structure of the Pipeline example

The code to be checked in this example implements a pipeline of threads where each stage in the pipeline processes inflowing data and then passes the transformed data onto the next downstream stage. Figure 8 shows the basic structure of the pipeline system whose code is contained in the file `PipeInt.java` in the tutorial examples directory. This code is displayed in Figure 10. Figure 8 shows that the pipeline has three processing stages (threads) followed by a "listener" thread that simply receives the incoming data and prints it to the screen. The data fed into the beginning of the pipeline is generated in the `main` method of the `PipeInt` class. The threads in the pipeline are connected by instances of a `Connector` class. For this example, a connector is a simple queue that can hold up to one integer.

Let's look at the actual code in Figure 10 beginning with the `main` method of class `PipeInt`. After the instances of all four threads (`Stage1`, `Stage2`, `Stage3`, and `Listener`) and connectors are created, a `for` loop places integers from 1 to 100 into the `queue` of the first connector `c1` by calling `c1`'s `add` method. Following the `for` loop, `c1`'s `stop` method is called. Examing the code for the `Connector` class shows that this causes a `0` to be placed into the `queue` of `c1`. Each of the stage threads considers an incoming `0` value to be a shutdown signal. Thus, if `Stage` number i receives a `0` value from its upstream connector, it (a) issues a `stop` on its downstream connector which causes a `0` value to be placed in downstream connector's `queue`, and (b) completes (i.e., shuts down). If `Stage` number i receives a non-zero integer from its upstream connector from a call to `take` on `Connector` number i, it grabs the integer, increments it by 1, and puts it into the `queue` of the `Connector` number $i + 1$ via a call to the `add` method of `Connector` number $i + 1$. The `Listener` thread simply takes the integers out of the last `Connector`'s `queue`, and when it receives a `0`, the `Listener` shuts down.

Let's now consider how to use Bandera to see if the system satisfies a simple temporal property. In general, the methodology for rendering temporal properties in Bandera proceeds as outlined in Figure 5.5.1.

5.5.2 Writing a specification using the Bandera Specification Language

In the implementation of Pipeline system, a useful requirement would be that if the `main` method of the `PipeInt` class is done (i.e., issues a `stop` to its down-stream connector), then each of the remaining stages in the pipeline shuts down. Let's concentrate on formalizing and checking this requirement for the first stage: if the `main` method of the `PipeInt` issues a `stop` on its down-stream connector `c1`, then `Stage1` will eventually shutdown. The source code features mentioned in this requirement are the control point associated with the call to the `stop` method of `c1` and the control point associated with the printing of the shutdown message in the `run` method of `Stage1`. We can use Bandera's `LOCATION` observable form to define predicates that become true when control reaches these points. To declare the observable control point in `PipeInt`'s main method, we will (a) introduce a Java label (e.g., `c1StopCallLoc`) at the line of the control point (i.e., the line where `Heap.c1.stop()` appears), and (b) give the following Javadoc comment immediately above the method declaration.

```
/**
 * @observable
 *   LOCATION[c1StopCallLoc] c1StopCall;
 */
```

Similarly, to declare the observable control point in `Stage1`'s run method, you would (a) introduce a Java label (e.g., `stage1ShutdownLoc`) at the line of the control point (i.e., the line where `System.out.println("Stage1 shutdown")` appears, and (b) give the following Javadoc comment immediately above the method declaration.

```
/**
 * @observable
 *   LOCATION[stage1ShutdownLoc] stage1Shutdown;
 */
```

Now that we have identified the relevant control points (one can also think of the execution of these control points as observable *events*) and declared predicates that hold true when execution reaches these control points, we are ready to state the temporal relationship between these events using Bandera's temporal specification pattern system. Note the example requirement has the following form:

if P occurs then eventually Q occurs

where P corresponds to the predicate `PipeInt.main.c1StopCall` and Q corresponds to the predicate `Stage1.run.stage1Shutdown`. In Bandera's pattern system, temporal properties that have this form are called response properties — because Q must respond to P. Each pattern use must include a scope specification which delineates the segments of the execution trace where the property should hold. The example requirement should hold throughout the entire execution so it is said to have *global* scope. The actual rendering of the requirement in BSL is as follows.

```
Response:  forall [s:Stage1].
           {Stage1.run.stage1Shutdown(s)}
             responds to {PipeInt.main.c1StopCall} globally
```

Let's look at several aspects of this specification.

- The `responds to` identifies the property as instance of the response pattern instantiated with propositions `Stage1.run.stage1Shutdown` and `PipeInt.main.c1StopCall`.
- The `globally` specifies a global scope for the property (i.e., the property should be checked along the entire execution).
- The `forall [s:Stage1]` uses BSL's support for a form of *object quantification* that allows the property above to be stated for all instances `s` of the `Stage1`.
- Each predicates in BSL can be classified as either a *static* or an *instance* predicate. Static predicates are used to refer to static fields or program points of static Java methods; instance predicates are used to reason about instance fields or program points of nonstatic Java methods. The predicate `Stage1.run.stage1Shutdown` is an instance predicate because it refers to a location in the non-static method `run` of class `Stage1`. The predicate `PipeInt.main.c1StopCall` is a static predicate

because it refers to a location in the static method `main` of class `PipeInt`. For instance predicates we need to specify the instance to which the predicate refers. In our case, the property holds for any instance of `Stage1`. Thus, the quantified variable `s` is supplied as a predicate to `stage1Shutdown`. In general, any time an instance prediate refering to an attribute of a class C is used in a BSL specification, the specification must include quantification over instances of C, and the quantified variable must be passed as a parameter to the instance predicate. Currently, BSL supports multiple universal quantifications at the outermost level. Existential quantification is not supported.

When processing this specification for Spin, Bandera will compile the response property above down to an LTL property of the following form

```
[](P -> <>S)
```

where P is replaced with a lower-level predicates on internal representations that hold true at event of calling `Heap.c1.stop()` (similarly for S). The compiled formula for Spin also includes some additional structure needed to properly implement the quantification. Section 6 gives additional information about the syntax, semantics, and implementation of BSL specifications and object quantification.

5.5.3 Entering a specification using the Bandera User Interface

Now we are ready to create a new session and to check whether the property we specified above holds for this program. When creating a new session, we have the choice of creating a new session file or adding the session to the existing session file. Let's add a new session to the existing session file.

- Click on the **Open** option on the main frame top tool-bar. This will bring out the Session Manager window and display the current sessions. At this point you can click on the **New** option on the left of the window. This will enable the rest of the options such as **Activate** and **Remove**. Your session is called `Untitled1` by default. You can rename it by positioning the cursor in the **Name** field and typing the name you like as if you were working with a text editor. For simplicity, let's call this session `MyPipeInt`. Before you can proceed, you need to **Activate** this session.
- Now you need to specify the file that you want Bandera to work with. At the top of the Session Manager window you will see the following tabs: **Sessions**, **Compiler**, **Property**, **Abtraction**, **Birc**, **Checker**, and **Misc**. Click on the **Compiler** option. You should see that Bandera automatically chooses the classpath to be set to the directory where you invoked Bandera. If you followed the instructions above, you should have all your examples in the same directory (this is done for simplicity in this tutorial presentation, you don't necessarily need to have your files in the same directory). To specify the name of the file, click on the **Edit** option in the **Source** row. This will pop up another window that will list all of the JAVA source code files in the current directory. If you wanted to point to a file in another directory, you would simply navigate through the hierarchy of directories using the options at the top of the window. Left-click on `PipeInt.java`, then click on the **Select** option at the bottom of the window. You are done with choosing the file.
- If the specification that you want to check has been been entered into Bandera before or if you have written it manually using a text editor, you can choose the **Property** tab at the top of the Session Manager window and point to the existing specification file for the specification. However, for this example, we will create the specification file holding the specification to be checked from scratch using the BUI Property Manager. The BUI Property Manager provides a set of pull-down menus that allow you to insert declared predicates into specification patterns.

  Note: In order to build the database of declared predicates, Bandera first needs to load the `.java` source files containing the declared prediates. This will cause the predicates to be parsed and type-checked.

  Accordingly, close the Session Manager window by clicking on **OK** at the bottom. If you look at the top tool-bar of the main frame, you should see that only **JJJC** option is enabled. To load the source file, click on the **RUN** option. You are now ready to use the BUI Property Manager to create a specification to be checked.

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

class PipeInt {
  /**
   * @observable
   *   LOCATION[c1StopCallLoc] c1StopCall;
   */
  static public void main (String[] args) {
    Heap.c1 = new Connector();
    Heap.c2 = new Connector();
    (new Stage1()).start();

    Heap.c3 = new Connector();
    (new Stage2()).start();

    Heap.c4 = new Connector();
    (new Stage3()).start();

    (new Listener()).start();

    for (int i = 1; i < 100; i++)
      Heap.c1.add(i);

    c1StopCallLoc:
    Heap.c1.stop();
  }
}

final class Connector {
  int queue = -1;
  public final synchronized int take() {
    int value;
    while (queue < 0)
      try {
        wait();
      } catch (InterruptedException ex) {}
    value = queue;
    queue = -1;
    return value;
  }

  public final synchronized void add(int o) {
    queue = o;
    notifyAll();
  }

  /**
    * @observable
    *   INVOKE inv1: (this == Heap.c1);
    * @assert
    *   POST post1: (queue == 0);
    */
  public final synchronized void stop() {
    queue = 0;
    notifyAll();
  }
}
```

```java
final class Stage1 extends Thread {
  /**
   * @observable
   *   LOCATION[stage1ShutdownLoc] stage1Shutdown;
   */
  public void run() {
    System.out.println("Stage1 startup");
    int tmp = -1;
    while (tmp != 0) {
      if ((tmp = Heap.c1.take()) != 0)
        Heap.c2.add(tmp + 1);
    }
    Heap.c2.stop();

    stage1ShutdownLoc:
    System.out.println("Stage1 shutdown");
  }
}

final class Stage2 extends Thread {
  /**
   * @observable
   *   RETURN stage2run;
   */
  public void run() {
    System.out.println("Stage2 startup");
    int tmp = -1;
    while (tmp != 0) {
      if ((tmp = Heap.c2.take()) != 0)
        Heap.c3.add(tmp + 1);
    }
    Heap.c3.stop();

    System.out.println("Stage2 shutdown");
  } // end public void run()
}


...
...

final class Listener extends Thread {
  /**
   * @observable
   *   LOCATION[listenerShutdownLoc] listenerShutdown;
   */
  public void run() {
    System.out.println("Listener startup");
    int tmp = -1;
    while (tmp != 0) {
      if ((tmp = Heap.c4.take()) != 0)
        System.out.println("output is " + tmp);
    }

    listenerShutdownLoc:
    System.out.println("Listener shutdown");
  } // end public void run()
}
```

Figure 10: Java Source for Pipeline Example

- Find the **Property** option on the tool-bar of the main frame and click on it. This will bring out the Property Manager window. You will see three tabs at the top of the window: **Property**, **Temporal Logic**, and **Assertion**. Choose **Temporal Logic** tab. At this point there is only one option enabled, you can only **Add** a new formula. Click on **Add**. By default the name of the formula is `Untitled1`. After activating the property by clicking on the **Activate/Deactivate** you can rename it by editing the **Name** field of the window. Let's call this property `stage1ShutdownProp`.
- Now move your attention to the bottom part of the Property Manager window. Before we proceed to the picking of the predicates, we will edit the **Quantifications** field. One of the predicates we want to use in the formula appears in the `run` method of thread `Stage1`. We need to quantify this predicate for all instances of `Stage1`, therefore, enter the following line in the **Quantifications** field:

  ```
  forall[s:Stage1].
  ```

  The property we want to check is a response property with a global scope, so under the **Pattern Name** choose **Response**. Under the **Pattern Scope** choose **Globally**.
- Now we are ready to pick the predicates. We will start with the first proposition of the formula, so under **Proposition** choose **P** and right-click in the field on the right. You should see the **Add a predicate** window pop up. Left-click on this little window. It will bring out another window that will let you choose the predicate for **P**. You should see a picture in front of you as shown in Figure 11.

  Choose `PipeInt.main.c1StopCall` out of the list of predicates. Now you need to specify **S**. Follow the same steps as for **P**, only choose `Stage1.run.stage1Shutdown` predicate out of the list of predicates. Once this predicate name appears in the text box, add `(s)` at the end of the predicate name to reflect that we are working with a predicate for all instances of `Stage1`. After you are done with defining S, the field you edited should look like:

  ```
  Stage1.run.stage1Shutdown(s)
  ```

  It is always a good idea to click on **Expand** option to check whether your formula is valid. Go ahead and expand the formula, it should be valid.
- You can save the property you generated into the file. Click on the **Property** tab at the top of the Property Manager window and choose **Save as** option at the bottom. You can save the file in any directory under any name you like, *but you must give your specification file the extention* `.specification`. You can close the Property Manager window now by clicking on **OK** option at the bottom.

5.5.4 Using a simple data abstraction

We will turn our attention back to the main frame of the BUI. If you look at the code in the `main` method of class `PipeInt`, you will see that the `for` loop sends integers from `1` to `100` through the pipeline, and this relatively wide range of values will cause the model-check of the given property to run for many hours. In fact, because a zero value is used to indicate shutdown, proving the given property only requires distinguishing zero from non-zero queue values.

To dramatically reduce the cost of checking this property, we will abstract the queue values to three abstract tokens `neg`, `zero`, and `pos` using the Abstraction Manager. This abstraction is called the *Signs* abstraction.

- Click on the **Typer** option on the tool-bar of the main window. This will bring out the Abstraction Manager window. Expand the Default Package by double-clicking on it. Then double-click on `PipeInt`, then expand `main` method to see the variables used in the method.
- We need to use the abstraction for the integer i. Go to the **Abstraction** column of the i's row and left-click there. You should see the menu with different options pop up, as shown in Figure 12.

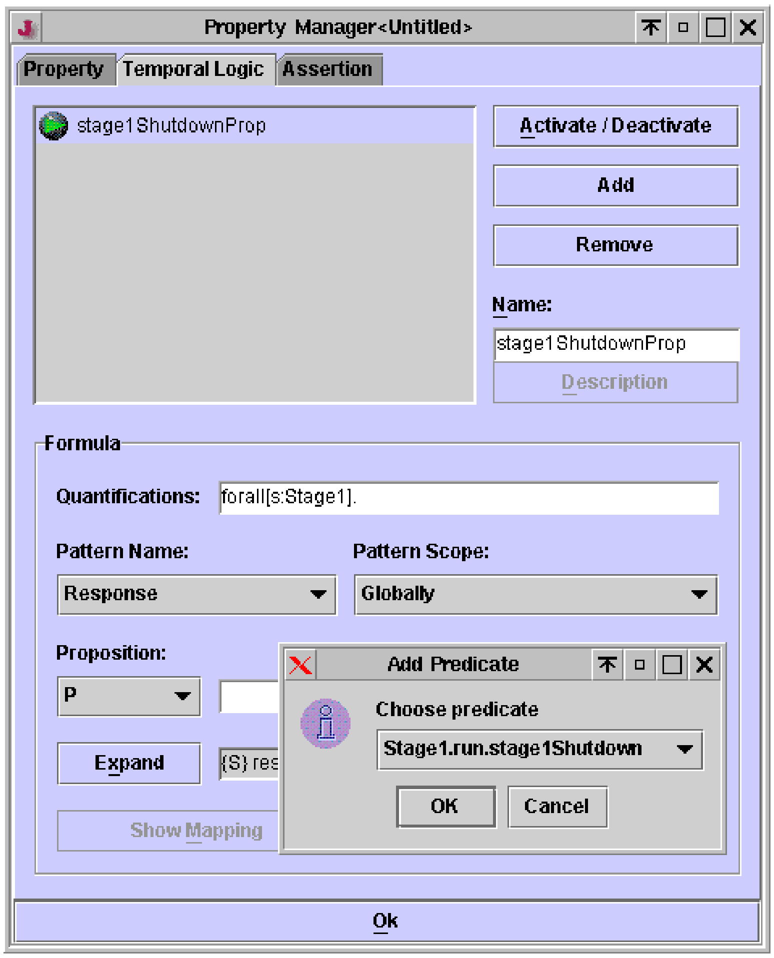

Figure 11: Property Manager Window

- Choose the `Signs` abstraction, then click on the running man at the left upper corner of the Abstraction Manager window to run the type inference. This will bring out another window that will let you know whether the type inference failed or succeeded. You should get an `OK` message this time. Dismiss the little window.
- You can save the abstractions you used to a file by clicking on the **Save** option next to the running man. This will bring out the window that will let you save the abstraction selection in your working directory. Navigate through the hierarchy of directories to save the file in the directory of your choice. You should use the extension `.abstraction` for your file. You can close the Abstraction Manager window now.

To finish this session check the **SLABS** option on the upper tool-bar of the main window to enable the use of abstractions.

To pick the model checker you want to use at the back end of Bandera, bring out the Session Manager window by clicking on the **Session** option of the main window. Click on the **Checker** tab at the top of the Session Manager window and pick the model checker Spin.

Enable the **Checker** on the top tool-bar and **Run** Bandera again. After some period of time, you should get a confirmation that your property is verified as shown in Figure 13.

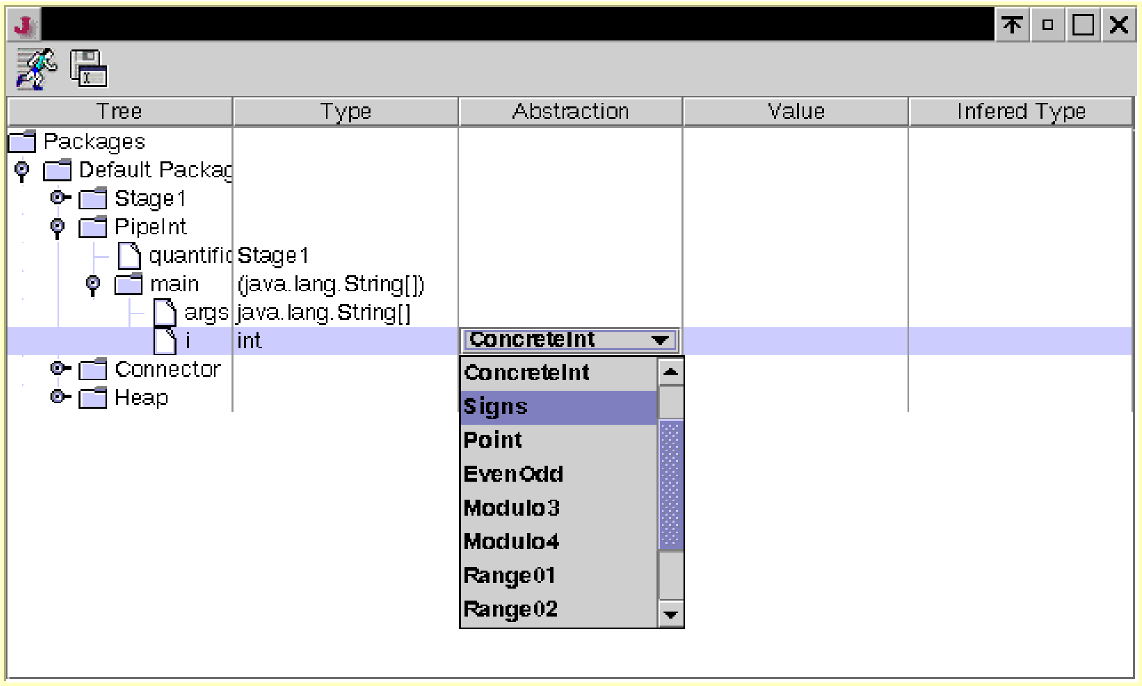

Figure 12: Abstraction Manager Window

5.6 Bounded Buffer Example

5.6.1 The structure of the Bounded Buffer example

In this section we will look at another multi-threaded JAVA program and use it to further illustrate construction of temporal specifications written in BSL. We'll start with a program that omits the Javadoc declaration of predicates and assertions, and you'll add appropriate declarations as an exercise. Once you've made these additions to the source code, we'll step through the definition of a few of the corresponding sessions and leave the remaining ones as exercises.

Figure 14 shows the program that we consider: it implements a simple bounded buffer (in the `BoundedBuffer` class) and contains several other auxiliary driver classes used to exercise the behavior of the buffer.

The `BoundedBuffer` class has two methods used for adding and removing elements from the buffer. You can add to the buffer if the buffer is not full and if you have a lock of the `BoundedBuffer` that guarantees an exclusive access to the buffer. If the buffer is not empty and you have access to the buffer, you can take an element out of the buffer. The `BoundedBuffer` has 4 fields: `buffer_` represented by an array of `Elements`, `bound_`, `head_`, and `tail_` represented by an `int`. The `bound_` variable represents the number of slots in the `buffer_`. The `head_` variable points to the next available slot in the buffer, and `tail_` points to the last available slot.

The main driver class is called `IncompleteBoundedBuffer` because we have omitted the various BSL predicate definitions for the properties that we will check later. After you have inserted the definitions into your file, you can double check your work by looking in the `CompleteBoundedBuffer.java` file in your tutorial examples directory – that file contains code that is annotated with the complete BSL definitions.

The driver creates two `BoundedBuffers` and two threads called `InOut1` and `InOut2` that work with

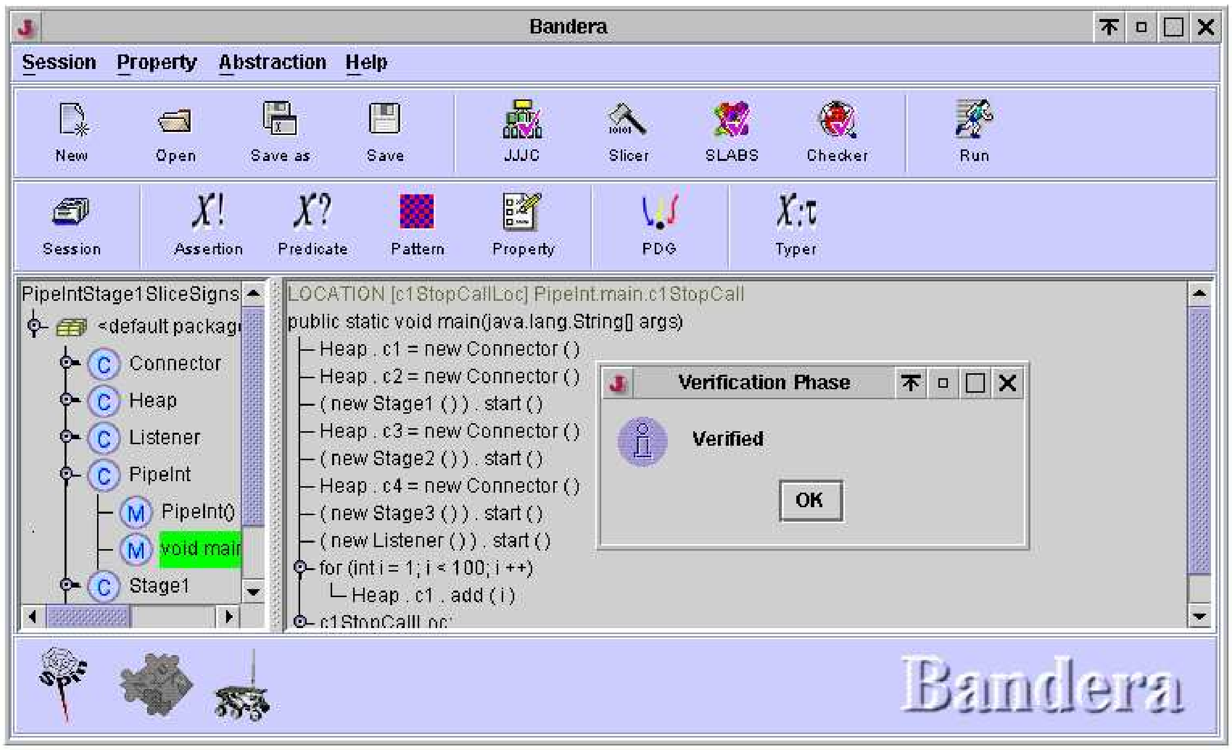

Figure 13: Pipeline Verified

the buffers. Thread `InOut1` constantly takes an element out of the first buffer and puts in into the second buffer. Thread `InOut2` takes an element out of the second buffer and puts it into the first one.

5.6.2 Bounded Buffer requirements

What are some of the properties we might want to check for the `BoundedBuffer`? And what are some of the constraints we might want to put on the `BoundedBuffer`? Below is a list of properties that we can check with Bandera.

1. Buffers are constructed with positive bounds.
2. Elements are always added in correct position.
3. Indices always stay in range.
4. Full buffers eventually become non-full.
5. Empty buffers must be added to before being taken from.

In the next subsections, we discuss the BSL predicate and assertion definitions necessary for specifying each of these requirements. Then we will return to using the BUI to define appropriate sessions for checking specifications of Requirements 1 and 3.

5.6.3 Requirement one

The first two requirements can be expressed using BSL's assertion facility. There are three kinds of assertions in BSL: precondition, location, and postcondition assertions. For example, to state Requirement 1 we can use a precondition assertion by inserting the following Javadoc comment immediately above the constructor method for the `BoundedBuffer` class.

```
class BoundedBuffer {
   Element [] buffer_;
   int bound_;
   int head_, tail_;

   public BoundedBuffer(int b) {
     bound_ = b;
     buffer_ = new Element[bound_];
     head_ = 0;
     tail_ = bound_-1;
   }

   public synchronized void add(Element o) {
    while ( tail_ == head_ )
      try { wait(); } catch
        ( InterruptedException ex) {}

    buffer_[head_] = o;
    head_ = (head_+1) % bound_;
    notifyAll();
   }

   public synchronized Element take() {
     while ( head_ == ((tail_+1) % bound_) )
       try { wait(); } catch
         ( InterruptedException ex) {}

     tail_ = (tail_+1) % bound_;
     notifyAll();
     return buffer_[tail_];
   }

   public synchronized boolean isEmpty() {
     return head_ == ((tail_+1) % bound_);
   }
}
```

```
public class IncompleteBoundedBuffer {
   public static void main (String [] args) {
     BoundedBuffer b1 = new BoundedBuffer(3);
     BoundedBuffer b2 = new BoundedBuffer(3);

     b1.add(new Element());
     b1.add(new Element());

     (new InOut1(b1,b2)).start();
     (new InOut2(b2,b1)).start();
   }
}

class Element {
}

class InOut1 extends Thread {
  BoundedBuffer in_,out_;
  public InOut1(BoundedBuffer in,
                BoundedBuffer out) {
    in_ = in;
    out_ = out;
  }
  public void run() {
    Element tmp;
    while(true) {
      tmp = in_.take();
      System.out.println("Take 1");
      out_.add(tmp);
      System.out.println("Put 1");
    }
  }
}

class InOut2 extends Thread {
  BoundedBuffer in_,out_;
  public InOut2(BoundedBuffer in,
                BoundedBuffer out) {
    in_ = in;
    out_ = out;
  }
  public void run() {
    Element tmp;
    while(true) {
      tmp = in_.take();
      System.out.println("Take 2");
      out_.add(tmp);
      System.out.println("Put 2");
    }
  }
}
```

Figure 14: Java Source for BoundedBuffer Example

```
/**
 * @assert
 *   PRE PositiveBound: (b > 0);
 */
```

The role of the tag `@assert` is similar to that of `observable` but is used indicate that an assertion is being defined instead of a proposition to be used in a temporal formula (the two roles cannot be interchanged without changing the tag). The label `PRE` indicates that this is the precondition assertion that needs to hold right before we enter the constructor code. The name of the assertion is `PositiveBound`.

Since we are expressing this requirement as an assertion, we do not need to instantiate a temporal pattern as we did for the requirement in the Pipeline example. Rather, Section 5.6.8 will show that we simply define a session and enable this assertion for checking.

5.6.4 Requirement two

Requirement 2 can be expressed using a post-condition assertion for the `add` method. Specifically, the following assertion declaration would be inserted immediately before the `add(Element o)` method of the `BoundedBuffer` class.

```
/**
 * @assert
 *   POST addPost: (head_==0) ? buffer_[bound_-1]==o : buffer_[head_-1]==o;
 */
```

The assertion states that if the `head_` was zero before an element was added to the buffer, then the element will appear in `buffer_[bound_-1]`, if the `head_` was non-zero, then the element will appear in `buffer_[head_-1]`.

5.6.5 Requirement three

The word "always" in requirement three suggests that the requirement is a simple invariant property. Following the methodology of Figure 5.5.1, we examine the code and see that the notion of *indices in range* can be expressed by an appropriate constraint on the `BoundedBuffer` instance variables `head_`, `tail_`, and `bound_`. Thus, we define the range notion as in instance predicate by inserting the following declaration to the header of the `BoundedBuffer` class.

```
/**
 * @observable
 *   EXP IndexRange: (head_ >= 0 && tail_ >= 0 && head_ < bound_ && tail_ < bound_);
 */
```

The tag `@observable` indicates that a predicate for a temporal property is being defined. The label `EXP` is used to define a location insensitive predicate, i.e., a predicate that only defines some constraint on data without mentioning a particular control point. The predicate above is an *instance predicate* (and thus must be quantified in the temporal property) because it refers to instance variables in the `BoundedBuffer` class. Typically, an invariant is defined using `EXP` predicates since by the very nature of an invariant the property should hold across multiple control points. Now we can specify the temporal property that will use `IndexRange` predicate. Positive invariants correspond to the universal temporal pattern. Since the invariant is to hold through the entire execution of the program, the scope of the property is global. We also need to quantify the property for all instances of the `BoundedBuffer`. The final version of the property expressed in BSL is as follows:

```
IndexRange: forall[b:BoundedBuffer].
          {IndexRange(b)} is universal globally;
```

5.6.6 Requirement four

The words "eventually become" in the requirement suggest that the property should be specified not as an assertion, but as temporal property. Following the methodology of Figure 5.5.1, we examine the code and see that the notion of *full* can be expressed by a BSL `EXP` instance predicate on the variables `head_` and `tail_` of the `BoundedBuffer` class. Specifically, the buffer is full when the `head_` is equal to the `tail_`. Thus, we add the following definition to the header of the `BoundedBuffer` class.

```
/**
 * @observable
 *   EXP Full: (head_ == tail_);
 */
```

Now we are ready to specify the temporal property that expresses "whenever a buffer is full, eventually it will become non-full". Notice that this property as the same temporal structure as the Pipeline shutdown property from the previous section: "whenever the main method of the Pipeline class issues a stop call, eventually the stage one thread will shutdown". Thus, we will use the same temporal pattern here, i.e., we use response property with global scope. Since will use the instance predicate above which refers to variables in the `BoundedBuffer` class, we need to quantify over all instances of `BoundedBuffer`. The final version of the specification is as follows.

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

5.6.7 Requirement five

For the final requirement *Empty buffers must be added to before being taken from*, there are three notions for which we need to define source code predicates: (1) the notion of *empty buffer*, (2) the event of adding an object to a buffer, and (3) the event of taking an object from a buffer.

The notion of empty buffer can be defined using the following instance predicate to code the test implemented in the `isEmpty` method.

```
/**
 * @observable
 *   EXP Empty: head_ == ((tail_+1) % bound_;
 */
```

The event of adding an object to a buffer can be defined by usin an `INVOKE` observable in the header of the `add` method.

```
/**
  * @observable
  *   INVOKE Call;
  */
```

The event of removing an object from a buffer can be defined by usin a `RETURN` observable in the header of the `take` method.

```
/**
  * @observable
  *   RETURN Return;
  */
```

To specify the temporal property, we will use the *absence* pattern with *after/until* scope. All of the predicates defined above are instance predicates on `BoundedBuffer` members, so we'll quantify over all `BoundedBuffers`.

```
NoTakeWhileEmpty:
   forall[b:BoundedBuffer].
     {take.Return(b)} is absent
         after {Empty(b) until {add.Call(b)}
```

5.6.8 Building a session for requirement one

After you insert the specifications into your file, you can double check your work by looking in the `CompleteBoundedBuffer.java` file that is annotated with BSL.

Now you are ready to run the Bounded Buffer program through Bandera and check various properties that we talked about in the previous section. We will check the first requirement only in this section. We will use the complete version of the source code file, i.e. we will have assertions and predicates ready for our use.

- **Open** an existing session file. It will bring out the Session Manager window.

- Click on **New** option to create a new session. Activate it and name it `PositiveBound` for instance. Click on the **Compiler** tab at the top of the Session Manager window and specify the file to be used. Make sure to use the `CompleteBoundedBuffer.java`. Close the Session Manager window and run the example through with only the **JJJC** option enabled on the top tool-bar of the main window. This will load the code into Bandera which will extract assertions and predicates specified in the file.

- Now you are ready to enable the assertion `PositiveBound`. Click on the **Property** option on the second tool-bar to bring out the Property Manager window. Click on the **Assertion** tab at the top of the window. Click on the **Add** option at the top and then on the **Activate/Deactivate** option in the window. This will activate the **Add** button at the bottom part of the window. Clicking on it will bring out the **Enable Assertion** window that will let you choose an assertion from a set of assertions specified in the program. Choose `BoundedBuffer.BoundedBuffer.PositiveBound` assertion and click on **OK** button to dismiss the **Enable Assertion** window.

- To save the requirement into a file click on the **Property** tab at the top of the Property Manager window. Choose **Save as** button at the bottom of the window. If you want to keep different specifications in seperate files, you can give this specification file a name that indicates the kind of requirement it contains, for example, `PositiveBound`. If you intent to keep all of specifications in the same file, you might want to name it `BoundedBuffer`, for instance. Make sure to use `.specification` extension for your specification file.

- Dismiss the Property Manager window by clicking on **OK** at the bottom.

- Open the Session Manager window by clicking on the **Session** option on the second tool-bar of the main window. Choose the model checker SPIN at the back end of Bandera by clicking on the **Checker** tab at the top of the Session Manager window and checking the SPIN's check box.

- Use the slicer for this session as it significantly reduces the state space of the model. Choose the **Slicer** at the top tool-bar to enable the slicer. Click on **Run** to obtain the results of slicing. After Bandera stops running, browse through the code to check what has been eliminated during slicing. Check the new code for the `BoundedBuffer` class. Note that the buffer array called `buffer_` has been removed from the program as well as any references to it.

  Slicing will not produce similar results for the second assertion from the previous section: elements are always added in the correct position. In this case the slicer can not eliminate the buffer array since it is reffered to in the assertion. This is a very important point to remember whenever you have a few assertions inserted into the program. You might not want to have all of the assertions enabled at once for different sessions. You might want to selectively enable some of the assertions only.

- Now you are ready to complete the model check. Bring out the Session Manager window again and click on the **Checker** tab at the top. Choose Spin as the model checker. Dismiss the Session Manager window. Click on the **Checker** option of the top tool-bar to enable the model checker and finally click on the **Run** option.

5.6.9 Building a session for requirement three

In this section we will describe how to create a session to check requirement three: indices always stay in range.

- **Open** an existing session file.

- Add a new session to it by clicking on the **New** option of the Session Manager window. Activate the new session and name it `IndexRange` for instance. Click on the **Compiler** tab at the top of the Session Manager window and choose `CompleteBoundedBuffer.java` as the target file. Close the

Session Manager window and click on **Run** on the top tool-bar to run the example through **JJJC** and load the code and specifications into the system.

- Click on the **Property** option on the second tool-bar to bring out the Property Manager window.
- The steps for creating the temporal property are similar to the steps taken for the pipeline example. Click on the **Temporal Logic** tab at the top, then click on **Add** to create a new property. **Activate/Deactivate** the property and give it a name, `IndexRangeProp`, for example.
- In the **Quantifications** field enter

  ```
  forall[b:BoundedBuffer].
  ```

- Remember that we are using a universal pattern with global scope for this property, so under the **Proposition** choose **Universality** and under the **Pattern Scope** choose **Globally**. Now you are ready to choose the predicate. You will need to choose only one. Choose

  ```
  BoundedBuffer.IndexRange
  ```

  predicate and add `(b)` at the end.

- **Expand** the formula to make sure it is correct
- To save the property specification to a file click on the **Property** tab at the top of the Property Manager window and click on **Save as** option at the bottom. You can either save this specification into a separate file or add it to an existing specification file, for example, the one you created in the previous section. To save into a separate file, give it new name. To save into already exisiting file you would have to edit the existing specification file using a text editor. Go ahead and save your property into a separate file.
- Dismiss the Property Manager window.
- Click on the **Slicer** option on upper tool-bar of the main window to use the slicer for better performance. **Run** the example through and browse the code to see which components get sliced away. You shoud see that as in the previous section, the slicer eliminates the buffer array. Note that for checking this property we don't want to have the second assertion enabled, as it would spoil the slicing criteria. This is a point to remember when creating sessions with multiple temporal properties and assertions enabled.
- To pick the model checker at the backend of Bandera click on the **Session** option of the main window and choose the **Checker** tab at the top of the Session Manager window. Pick the checker and dismiss the Session Manager window.
- Click on the **Checker** option to enable the model checker and at last click on the **Run**. You should get a confirmation that your property has been verified.

5.6.10 A complete specification file

As we already pointed out in the previous section, a user can create temporal properties and enable/disable assertions using the BUI. However, if a user wants to save specifications on the per session basis, she might end up with many different specification files. We encorage to keep all of the specifications for the same example in the same file.

You can create all of your specifications at once using the Property Manager window and save them in one file.

Another option would be to directly write out the specifications into a file using a text editor such as Emacs.

Figure 15 displays a single specification file that holds the property declarations for each of the five bounded buffer requirements.

The syntax for enabling multiple assertions is as follows:

```
import assertion BoundedBuffer.BoundedBuffer.*;
import assertion BoundedBuffer.add.*;
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 {Empty(b)}
                     until {add.Call(b)};
```

Figure 15: Bounded Buffer Properties rendered in BSL

```
BufferAssertions: enable assertions { PositiveBound, AddToEnd };
```

Once you have the full specification file created and saved, use the following steps to select the property you want to check in a single session:

- Bring out the Session Manager window. Click on the **Property** tab at the top of the window. Click on the top **Edit** button to select the specification file. Opening of the file will bring out the Property Manager window.
- Use the Property Manager window to select the property. Click on the **Temporal Logic** tab at the top. The Property Manager window will display the set of the temporal properties wriiten in the file. **Activate** the property you like to check in your session.
- Make sure to check the assertions enabled. As we pointed out in the previous sections, you might not want to have all of the assertions enabled at once, as some of them might be too strict and spoil the performance of slicing, for example.
- To control the set of assertions, click on the **Assertion** tab at the top of the Property Manager window. **Activate** the set of assertions displayed in the top part of the window. You will see the set of assertions displayed individually at the bottom part of the window. Use **Remove** option of the bottom part of the window to remove the assertions you don't want to be checked during the execution of your session.

6 The Bandera Specification Language

The previous sections have given some simple examples to illustrate the basic concepts of the Bandera Specification Language (BSL). This section gives a more indepth overview of the motivation and design rationale of the BSL, and includes a detailed discussion of BSL's syntax and semantics.

6.1 Introduction

Why have we defined our own specification language? Within the context of the Bandera project, another option would be to have developers 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 section, 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 the bounded buffer implementation from the previous section that states that *if a buffer is full, it will eventually become non-full.* 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 16.

- 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 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 [7]. 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 that allows temporal specifications to be quantified over all objects/threads from particular classes.

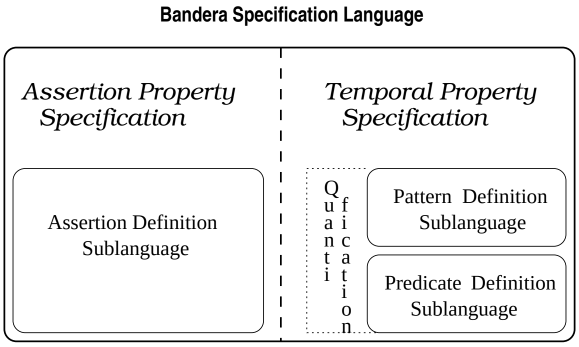

Figure 16: BSL organization

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 BSL overview is organized as follows. Section 6.2 discusses some of the salient issues that influenced our design of the language. Section 6.3 describes the design and implementation of the assertion sublanguage. Section 6.4 describes the design and implementation of the temporal specification sublanguage.

6.2 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 [5] and with a system of temporal specification patterns [7] 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.

```
<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 17: BSL assertion syntax

Criterion (1) derives from experience with the Specification Patterns Project. BSL provides a macro facility that implements all the patterns from [7]. 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 [12]. 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.

6.3 The Assertion Sublanguage

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.

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 [12], support definition of assertions in Java method header comments. BSL also adopts this approach. For example, we saw in Section 5.6 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.

Syntax and informal semantics Figure 17 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 post-condition 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).[9] `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`.

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

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 been 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 15 presents a BSL file where the first specification `BufferAssertions` enables checking of the `PositiveBound` as declared in Section 5.6. 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.

Implementation issues As with other Java assertion tools, the BSL assertion implementation acts as 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.

6.4 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.

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, Section 5.6 gave 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 the `BoundedBuffer` class. 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,

```
<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 18: BSL predicate definition syntax

it must be passed a parameter to indicate the instance to which the predicate applies. For example, the `FullToNonFull` property of Figure 15 shows the `Full` predicate being parameterized by the quantified variable `b`.

6.4.1 Syntax and informal semantics

Figure 18 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 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 15, 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 15, the instance `RETURN` predicate `take.Return` holds iff the `take` method was invoked on the object referenced by the quantified variable `b` and control is immediately after the `return` statement of `take`.

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 with each having a different set of visible local variables.

6.4.2 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.

6.5 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 [3] 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 [6] 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

- *global*ly 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 [7], 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,

```
<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 19: BSL Pattern and Quantifier Syntax

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 19. Common synonyms are supported. For example, *invariance* and *universal* patterns and *leads to* can be used to express the `FullToNonFull` property from Figure 15 as

```
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.

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 19. 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.

6.5.1 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.

6.5.2 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 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 [8] 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`.

7 The BUI Predicate Manager

Forthcoming

8 The BUI Pattern Manager

Forthcoming

9 The Bandera Slicer

Forthcoming

9.1 Appropriate notions of dependence for concurrent Java

9.2 How slicing criteria are generated

9.3 Using the dependency browser

10 The Bandera Abstraction Tools

Forthcoming

11 Limitations and Known Bugs

References

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

[2] 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.

[3] 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.

[4] Matthew B. Dwyer and Corina S. Pasareanu. Model checking generic container implementations. In *Proceedings of the 1st Symposium on Generic Programming*, May 1998.

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 1999. To appear.

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

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

[14] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nichloas Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[15] 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.