Checking Java Concurrency Design Patterns Using Bandera

Cleidson R. B. de Souza and Roberto S. Silva Filho
Department of Information and Computer Science
University of California, Irvine
{cdesouza, rsilvafi}@ics.uci.edu

Abstract

Software patterns express a generic solution for a specific design problem, conveying some knowledge and expertise from designers. Model checking is an automatic technique for verifying finite state systems that determines if a property holds of the given finite state machine. The paper describes the experience of the authors in modeling three concurrency design patterns using the Bandera toolset for model checking. The concurrency properties of these patterns were specified using the BASL language and submitted for checking after the relaxation of the code. Due to some problems related to the immaturity of the tool, only one model could be checked, in one special property. The main contribution of the paper is the formalization of the patterns and the sequence of steps followed in order to formalize and check these patterns.

Key Words*: Pattern verification, Design Patterns, Model Checking, Software Specification.*

1. Introduction

Software patterns [Gamma94] facilitate reuse of well-established solutions to known problems. Although there are several different ways of presenting a pattern, there is an agreement in the community that a pattern description must include, at least its name, description, problem and the solution proposed. Therefore, when a designer identifies the same problem, he can apply the correspondent pattern to that problem and easily solve it.

There are several levels of abstraction for a software pattern [BMRSS96]: in an implementation level, they are called idioms; in a design level, design patterns; and finally if a higher-level of abstraction is used, they are called architectural patterns. In our work, we are dealing specifically with design patterns, therefore, in the rest of the paper patterns and design patterns will be assumed as synonyms.

It is also a common practice to present the information about the advantages of using a pattern. This helps the designer in the process of choosing a design pattern, when there are more than one of them available to solve the same problem. These advantages are usually written by the pattern developer and are based on his experience. In some cases, before being published the patterns are discussed in conferences by experts who evaluate the pattern, i.e., these experts try to validate them. However, this process is completely informal, it does not use any kind of formal method or tool in order to validate those advantages. Of course, this process is error-prone and suggests that some kind of formal approach must be used. This paper addresses this issue.

We present a formal specification of the properties of three concurrency design patterns implemented in Java. The analysis was performed using a Java source code model checker tool called Bandera [CDH+00]. Three different design patters were implemented and submitted for checking by this tool. These implementations had to be modified due to problems in Bandera's parser. We were not completely successful in checking all patterns because of bugs in the tool such as this parser problem.

We specified some properties using the specification language provided by the tool in order to validate the concurrency control features which the pattern claimed to address. In this case, we used an approach for checking properties called model checking. Model checking is an automatic technique for verifying finite state systems that determines if a property holds in a finite state machine (Chapter 1) [CGP99]. This finite state machine is automatically generated by Bandera, based on the source file presented to this tool. During the checking, an exhaustive search is done in the states to check if the property holds at every state.

1.1. Paper Description

The paper is organized as follows. Section 2 presents the motivation to our work. The next section, briefly discusses the concept of design patterns, as well as presents the three design patterns that are checked. Section 4 pre-

sents the Bandera model checker toolset. Then, section 5 presents the annotations that were inserted in each pattern with the tests that we developed to check the satisfaction of these conditions. Section 6 summarizes our results and our experience using Bandera. Finally, some conclusions are presented.

2. Motivation

Software patterns express a generic solution for a specific design problem, conveying some of the knowledge and expertise from designers. These patterns are abstracted in a way that other designers can reuse.

Design patterns, such as that provided by [Gamma94] are typically presented and documented using textual descriptions and problem motivations. It also uses object-oriented diagrams (class and sequence), and an implementation in a specific programming language, using a simple problem example. This approach has been successfully applied in other books and conferences. In fact, there are several conferences and workshops devoted to patterns.

The approach presented in this paper addresses this issue. We argue that validating design patterns using some kind of formal method could help the adoption of patterns, providing a way to check their reliability and correctness. By validating, we mean checking if the patterns provide the features that they claim. For example, does the Read/Write Lock [Grand98] pattern guarantee mutual exclusion? Since nowadays there are thousands of patterns, can one trust that a pattern provides all advantages described in their specification? These questions are addressed in this paper through the formalization of three concurrent design patterns. By formalizing, we mean the process of mathematically verifying properties of these patterns. Using this approach, the knowledge that is expressed in a design pattern can be validated, i.e., we avoid the problem of someone defining a pattern that does not provide the advantages claimed.

In order to validate design patterns, we suggest the following approach:

1. First, the patterns to be checked must be identified. In our case, we selected a small (three) number of concurrency patterns proposed by [Grand98] in his book.
2. The properties to be checked are identified. Since we are interested in checking concurrency patterns, the most important properties identified were deadlock freedom and mutual exclusion.
3. Then, according with these properties, the adequate formal method is selected. We selected a model checking approach to validate the patterns because it provides support for checking deadlock-freedom as well as other runtime properties of the system.
4. Now, a specific tool supporting the method should be selected. In our case, Bandera [CDH+00] was chosen because of it provides support for Java programs checking using annotations in Java code.
5. Usually, the tools used in model checking have some specific input language, such as SPIN's Promela formal language. Therefore, the patterns must be specified in this language. Since Bandera uses Java source code as its input, this step should not be necessary. We discuss it in more details in section 4.
6. Finally, the properties to be checked are specified according to the tool selected. Bandera has a language called BSL (Bandera Specification Language) which supports different constructs such as pre and post conditions, invariant expression definition, predicate evaluation and so on. These annotations are provided in the Java code to be analyzed as JavaDoc notes.

Of course, this sequence of steps is not mandatory, i.e., it is just a recommendation. It can be modified to accommodate other objectives. For example, the selection of Bandera influenced our decisions in the opposite way: we wanted to test the Bandera toolset, therefore we selected concurrency patterns previously implemented in Java.

3. Design Patterns: History, Definition and Java concurrency Patterns

Design Patterns are design solutions adapted in the resolution of frequent problems in the software development phase. In the following sections, we present a brief history of design patterns, the concepts that comprise their use and documentation, and we describe the patters used in the project.

3.1. Brief History of Design Patterns

Software patterns have their origin in the ideas published in 1977 and 1979 by Christopher Alexander in the field of architecture patterns for urban planning [Alexander79]. In his work, the design rationales of common design solutions were presented in a structured way. In 1987, Ward Cunnigham and Kent Beck verified that the approach of documentation and reuse of ideas, presented in this work, could be applied in the Software Engineering field. In 1987, Ward Cunnigham and Kent Beck described five patters for user interface design. Such ideas were based on the initial work of Christopher Alexander.

(OOPSLA-87 – "*Using Pattern Languages for Object-Oriented Programs*").

In 1994, the book Design Patterns by Reich Gamma, Richard Helm, John Vlissides and Ralph Johnson [Gamma94], also known as "Gang of Four", popularized the concept of software design pattern.

In 1998 the book "*Patterns in Java*", by Mark Grand [Grand98], was published. It releases as an evolution to the "Gang of Four" book. The book presents several patterns, many of them not present in the Gamma's book. In this book, the UML notation is used to describe the general solutions and the design patterns. The examples are coded and described using the Java programming language.

3.2. Concepts

According to Alexander [Alexander79], "A Pattern could be defined as a three part rule that expresses a relation between a certain context, a problem and a solution".

Design patterns represent reusable structures and concepts, applied during the design phase of Software Engineering process. They improve the software development by presenting generic solutions that provide flexibility and understanding that facilitate future extensions to the software.

Design patterns present many advantages as follows.

1. They improve reuse and generality. Experienced developers can use recurring and generic solutions, instead of implementing a proprietary and not so generic solution each time it is necessary.
2. They provide a common vocabulary to developers. Design patterns allow the collaboration of developers, which can use these patterns to exchange knowledge and discuss problems in terms of well-known patterns, in a higher abstraction level.
3. They improve the design documentation. A project can be expressed in a higher level, using patterns that are well known by the developers, instead of using a proprietary code and practice.
4. They allow the expert programmers to represent their knowledge in a reusable and more documented way, creating a medium to teach good practice design to novice developers.

The use of design patterns is supported by the experience and knowledge of the developers, which should be able to understand and specify systems using this approach. This approach, however, can be difficult to use in environments in which this practice is not so widespread. It also requires an additional effort from the programmers to document, use and keep these patterns.

Design patterns can vary according to the granularity and its abstraction level. Their classification, according to some criteria, makes it easy to understand, document and be identified. In this work, we used some of the Concurrency Design patterns described in the Mark Grand's book.

3.3. Concurrency Design Patterns

These patterns present generic solutions to frequent concurrency problems found in concurrent and distributed systems. They focus on two kinds of problems, the sharing of resources, focusing on the deadlock management, and the concurrent execution of operations. These operations must follow a correct sequence of operations, for example, the insertion of a data element in a data structure should happen before its removal. These are the patterns described in the Book: (1) Single Threaded Execution, (2) Guarded Suspension, (3) Balking, (4) Scheduler, (5) Read/Write Lock, (6) Producer-Consumer, and (7) Two-Phase Termination.

We selected three of these patters. They were the Single-Threaded Execution, Read/Write Lock and Producer-Consumer. These patters are described in the following subsections.

3.3.1. Single Threaded Execution [Grand 98]

This design pattern describes a solution for the concurrency control problem in the case of multiple readers and multiple writers to a single resource. It prevents problems that may occur when concurrent callers invoke an operation and both calls access the shared resource at the same time. The most common problems in this situation are lost updates and inconsistent reads.

3.3.2. Context

Consider a system that monitors the flow of cars in a highway. Sensors in each lane of the highway monitor the passage of cars, sending this information to a local controller. This controller is attached to a transmitter that periodically sends the total information to a central computer. The class diagram of this system is presented in Figure 1 below.

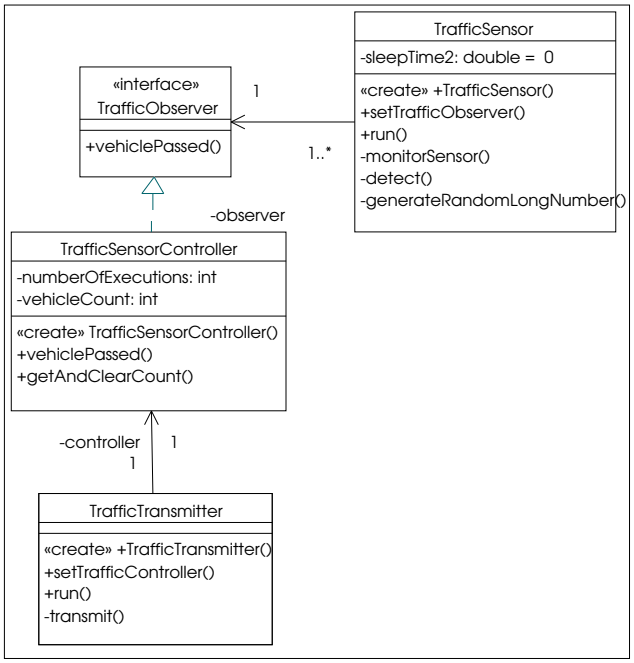

Figure 1 Single Threaded Execution - Traffic sensor classes.

Instances of the *TrafficSensor* class represent the sensors in the lanes. Each time a car is detected, a sensor calls the *vehiclePassed()* operation in the *TrafficObserver* interface. A *TrafficTransmitter* instance collects the number of vehicles passed in a road periodically. It does this operation, calling the *getAndClearCount()* operation in the *TrafficSensorControler* interface.

Concurrency situations:

A lost update may occur when two *TrafficSensor* instances call the *vehiclePassed()* operation in a *TrafficSensor* controller instance at the same time. In this case, the *vehicleCount* can be incremented only once, instead of two times.

Race conditions may occur when both a *TrafficTransmitter* and a *TrafficSensor* instances try to access the *vehicleCount* variable at the same time. This access is indirectly provided by the *vehiclePassed()* and *getAndClearCount()* operations. The final value of this counter depends on the order of the execution of these two concurrent events. If the *TrafficTransmitter* executes first, the *vehicleCount* is set to zero and the operation invoked by the *TrafficSensor* changes the *vehicleCount* value to its previous value plus one, instead of the current value (zero), to one.

To avoid these two problems, the operations *vehivlePassed*() and *getAndClearCount*() are guarded with the synchronized modifier, which ensure that only one process can invoke one operation at a given time.

3.3.3. Consequences

Guarding methods can reduce the performance of the application, since threads have to wait for other ones in order to reach the shared resource;

The use of guarded methods makes the application thread-safe;

The use of guards in methods can enable the opportunity to threads become deadlocked

3.3.4. Code Examples

The following code snippet describes the main points of the implementation of the Figure 1example, specially the use of synchronized modifiers in the operations.

```
/**
* This method is called when a traffic
* sensor detects a passing vehicle.
* It increments the vehicle count by one.
*/
public synchronized void vehiclePassed() {
       vehicleCount++;
} // vehiclePassed()

/**
* Set the vehicle count to 0.
* @return the previous vehicle count.
*/
public synchronized int getAndClearCount()
{
       int count = vehicleCount;
       vehicleCount = 0;
       return count;
} // getAndClearCount()
```

3.4. Read/Write Lock [Lea 97]

This design pattern implements a solution for the concurrent control problem existing when multiple access to *read()* and *write()* operations, in an shared object, are performed concurrently. It implements a read/write lock acquisition protocol that enable multiple concurrent read calls whenever a write operation is not being performed. During a write operation, the writer must have exclusive access to the variable being modified. No simultaneous reads or writes are allowed during this operation.

3.4.1. Context

Consider a piece of software that controls electronic auctions. Items are put up for auction. People participate in the auction observing the bid values for each item. They also contribute with their own bids for some items. In a given moment, the auction of an item will close. In this example, there are many people reading bids from the items, but only one person can make a bid at a time.

The use of the Single Threaded Execution Desing Pattern [Grand98] solve the concurrency problem described above, but does not allow the multiple read of the bids at the same time. In this situation, the Read/Write Lock design pattern allows the single execution of the write operations allowing, however, the concurrent read of the data. The exchange of the messages in this scenario is described in Figure 2 below.

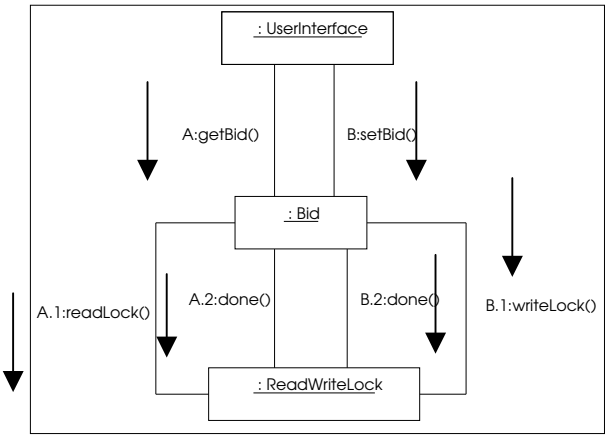

Figure 2 *ReadWriteLock Object Diagram.*

The *setBid()* operation modifies the current bid and prevents the *getBid()* operation to read inconsistent data. This operation waits for the return of the last *getBid()* or *setBid()* operations, before modifying the current bid value. The *getBid()* operation can be invoked at any time, as soon as the *setBid()* operation is not being executed.

The Read/Write Lock design pattern encapsulates the concurrency control described above, together with the lock acquisition and relinquishment protocol, assuring concurrent reads and exclusive writes.

The *readLock()* operation returns immediately, unless there is a *writeLock()* operation being executed, or waiting to be executed. The *writeLock()* operation signalizes the start of a reading cycle, and goes to a waiting state If: there is another *writeLock()* operation executing at this moment, or if there is a *readLock()* operation in execution.

The *done()* operation relinquishes a read or a write lock. This operation is invoked whenever the acquired locks are not necessary anymore.

3.4.2. Consequences:

The Read/Write Lock design pattern increases the concurrency of the reading operations and achieves mutual exclusion. It also allows the reuse of the concurrency control logic, increasing the concurrency whenever there are more read than write operations. However, this pattern does not perform well when the number of write operations is bigger than the number of read operations. For this last case, the use of the Single Threaded Execution pattern is recommended.

3.4.3. Solution:

The generic pattern for the concurrent read write problem is depicted in the Figure 3 below.

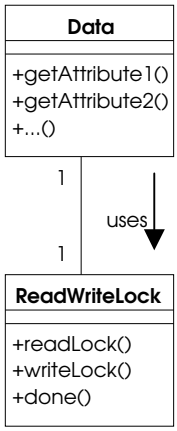

Figure 3 ReadWiteLock Pattern

The set and get operations call the operations *writeLock()* and *readLock()* of a ReadWriteLock object before getting access to the shared resource. This invocation is performed using delegation. The done operation is called at the end of this operation. For each object of type Data, there is an associated object from type ReadWriteLock

3.4.4. Code Example:

The following code snippet describes the main points of the implementation of the Figure 2 example.

```
public class Bid {
    private int bid = 0;
    private ReadWriteLock lockManager =
        new ReadWriteLock();

public int getBid() throws
```

```
        InterruptedException{
    lockManager.readLock();
    int bid = this.bid;
    lockManager.done();
    return bid;
} // getBid()

public void setBid(int bid) throws
        InterruptedException {
    lockManager.writeLock();
    if (bid > this.bid) {
        this.bid = bid;
} // if
     lockManager.done();
} // setBid(int)
} //class

public class ReadWriteLock {
    private int waitingForReadLock = 0;
    private int outstandingReadLocks = 0;

    private Thread writeLockedThread;
    private ArrayList waitingForWriteLock =
           new ArrayList();

    synchronized public void readLock()
throws InterruptedException {
    waitingForReadLock++;
    while (writeLockedThread != null) {
        wait();
    } // while
    waitingForReadLock--;
    outstandingReadLocks++;
} // readLock()

public void writeLock() throws
      InterruptedException {
    Thread thisThread;
    synchronized (this) {
        if (writeLockedThread==null && out-
standingReadLocks==0) {
            writeLockedThread =
Thread.currentThread();
            return;
        } // if
        thisThread =
Thread.currentThread();
        waitingForWrite-
Lock.add(thisThread);
    } // synchronized(this)
    synchronized (thisThread) {
        while (thisThread != writeLock-
edThread) {
            thisThread.wait();
        } // while
    } // synchronized (thisThread)
    synchronized (this) {
        int i = waitingForWrite-
Lock.indexOf(thisThread);
        waitingForWriteLock.remove(i);
    } // synchronized (this)
} // writeLock

synchronized public void done() {
    if (outstandingReadLocks > 0) {
        outstandingReadLocks--;
        if ( outstandingReadLocks==0
             && waitingForWrite-
Lock.size()>0) {
            writeLockedThread =
(Thread)waitingForWriteLock.get(0);
            writeLockedThread.notifyAll();
        } // if
    } else if (Thread.currentThread() ==
writeLockedThread) {
        if ( outstandingReadLocks==0
             && waitingForWrite-
Lock.size()>0) {
            writeLockedThread =
(Thread)waitingForWriteLock.get(0);
            writeLockedThread.notifyAll();
       } else {
            writeLockedThread = null;
            if (waitingForReadLock > 0)
              notifyAll();
       } // if
    } else {
        throw new IllegalStateExcep-
tion("Thread does not have lock");
    } // if
} // done()
} // class ReadWriteLock
```

3.5. Producer-Consumer

This design pattern coordinates the concurrent production and consumption of information among producer and consumer objects.

3.5.1. Context

Consider a trouble ticket (bug) dispatching system scenario. Consumers submit trouble tickets through web pages. Dispatchers review this information and forward the tickets to the appropriate person in the organization.

Tickets stay in a queue until dispatchers read them. Dispatchers read the queue periodically. If the queue is empty, the dispatchers wait until the first message comes in.

The system is represented in the Figure 4 below.

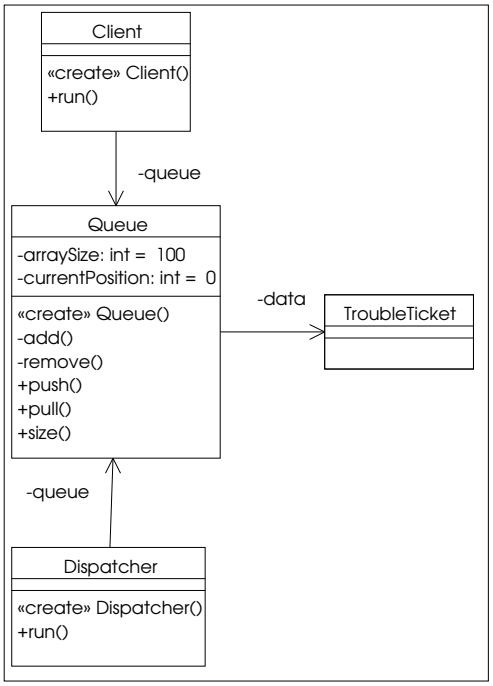

Figure 4 Producer-Consumer classes - Trouble Ticket example

Instances of client (the producer), asynchronously supply objects (TroubleTicket instances) to the queue. Asynchronous consumers (dispatchers) read these objects from the queue whenever they are available. The queue detaches the producers and the consumers, allowing their asynchronous indirect communication.

3.5.2. Consequences

Producer objects are detached from the consumer objects. They produce objects to a queue without the necessity to wait for the consumers response or availability;

When there are objects in the queue, the consumer can pull the objects immediately, if the queue is empty, the pull call waits until a new object is pushed by the producer.

3.5.3. Code Example

The concurrent access to the pull operation is controlled using a synchronized modifier in this operation.

Below, the main parts of the code are presented. This code is based on the example of the Figure 4.

```
public class Queue {
    private ArrayList data = new Array-
List();

synchronized public void push(TroubleTicket
tkt) {
    data.add(tkt);
    notify();
} // push(TroubleTicket)

synchronized public TroubleTicket pull() {
    while (data.size() == 0){
        try {
            wait();
        } catch (InterruptedException e) {
        } // try/catch
    } // while
    TroubleTicket tkt = (Trou-
bleTicket)data.get(0);
    data.remove(0);
    return tkt;
} // pull()
```

4. The Bandera Toolset

Bandera [CDH+00] is a toolkit for model checking Java programs. Model checking is an automatic technique for verifying finite state systems that determines if a property holds of the given finite state machine (Chapter 1) [CGP99]. In this technique, an exhaustive search is done in the model states to check if the property holds at every state.

Model checking, has been showed to be a successful technology for verifying hardware [WF95]. In fact, hardware manufacturers frequently use them to validate their designs. Furthermore, it can also be used in software systems. In the literature we can find papers describing the checking of: cache coherence protocols used in distributed file systems [WF95], electronic commerce protocols [HTWW96], and so on.

The Bandera toolset allows users to check static and dynamic properties in Java programs, allowing the tailoring of the analysis to a select set of properties in order to minimize analysis time. The Bandera architecture has the following main features:

- It reuses existing technologies as the model checkers;
- It Provides automated support for the abstractions used by experienced model designers; and
- Provides an open design, which can be extended according to future needs.

4.1. Bandera Architecture

In general, one can say that the Bandera architecture is similar to an optimized compiler: the input is translated

into intermediate representations, which are augmented with useful information. Bandera uses Java source code as input that is translated into a Jimple program. This program is sliced and abstracted and translated into another intermediate representation called BIR (Bandera Intermediate Representation). Finally, this BIR code is used to generate a program in the language of one of the three model checkers currently supported. For example, it generates output in Promela formal language if the model checker selected is the SPIN [Hol97] checker. In fact, Bandera was built on top of the *Soot* compiler framework [CDH+00].

Figure 5 presents the Bandera architecture. The main components of Bandera architecture are the Slicer, the Abstraction Engine, the Back End and the User Interface. Each one of these components is briefly described as follows. More information regarding these components are presented in the references [CDH+00], [DHR+01] and [PDW01].

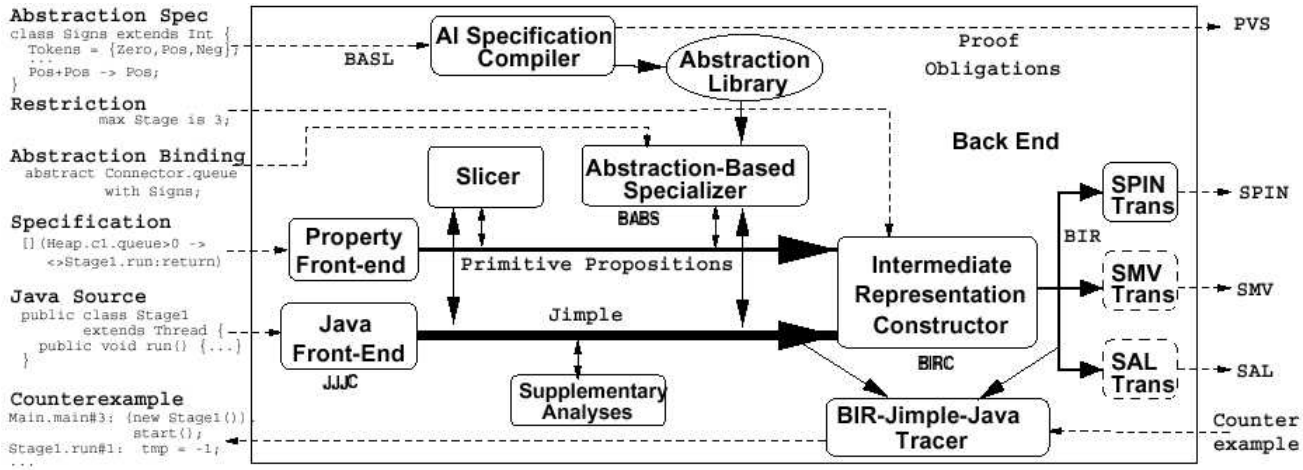

Figure 5: The Bandera Arquitecture

4.1.1. Slicer

The *Slicer* component is responsible for removing irrelevant source code for the checking of a given property. The slicing criteria are automatically extracted from the observable predicates that reference variables and predicates in the predicate being analyzed.

The process of slicing is based on the calculation of the dependency graph, which supports visualization of data, control and synchronization dependencies. This dependency graph also helps the process of selecting abstractions. Since all variables in the dependency graph are important to the property being checked, this information can be used to select the variables that appear most often in a checked predicate in order to simplify the checking.

4.1.2. Abstraction Engine

The *Abstraction Engine* copes with the reduction of the cardinality of data sets associated with variables. For example, if the property being verified depends only on whether or not a particular item is in the vector, instead of using a large number of vector states, we could use {*ItemInVector, ItemNotInVector*}[CDH+00].

Bandera provides a powerful specification language called Bandera Abstraction Specification Language (BASL) which can be used to create abstractions. However, the abstractions must be *safe*, i.e., they must over-approximate the set of true executable behavior of the system checked [DHR+01]. In order to guarantee this safeness a formal check must be performed. This checking is not a trivial process, therefore most Bandera's users can select abstractions from the Abstraction Library provided by this tool. Therefore, Bandera can be used by non-experts because it provides abstractions reuse.

The Abstraction Engine supports type inference in order to check conflicting abstractions [DHR+01]. For example, if a program with an assignment like "*y = z;*" and with "*z abstracted with {Neg, Zero, Pos}*" and "*y is abstracted with {Even, Odd}*" were checked, it would raise an error. The problem is that if *z* is a *Pos* value, it can not be determined if *z* is *Even* or *Odd*. In a case like that, Bandera reports the conflict to the user who can adjust the abstractions chosen.

It is important to note that slicing and abstractions are important because they reduce the total number of states to be checked helping to minimize the "state explo-

sion problem". The idea is that as the number of system components grows the size of a finite-state model increases exponentially. This is one of worst problem in applying model checking and it is much more difficult when model checking is applied to software systems, because these systems tend to have much more states than hardware components [CDH+00].

4.1.3. The Back End

At this point, an abstract Jimple program that was sliced and abstracted replaced the Java source code. This program will be used as input to the BIRC (Bandera Intermediate Representation Constructor) which creates a BIR representation of this program. BIR is a command-guarded language for describing state transition systems which also that abstracts the common model checker input language. This output is sent to model checkers specific translators. For example, the SPIN Translator accepts a BIR representation and produces a Promela model of the system.

Finally, the program is executed in the model checker and if a counter-example is found it is translated back by the *BIR-Jimple-Java Tracer* component into the original source code. Therefore, the user can check the behavior of his program as in a debugger [CDH+00].

5. Modeling and Checking Patterns

This section describes our experience in specifying and checking the concurrent design patterns previously described in Section 3.3.

The approach that we adopted was the following. First, we implemented each one of the patterns with Sun JDK1.3. Then, the programs were tested in Bandera in order to verify their absence of deadlock. After that, in order to test the Bandera ability to detect concurrency problems, the concurrency optimizations of this code were relaxed. In other words, the *synchronized* modifiers were removed from the code.

Finally, using the Bandera Specification Language (BSL) we inserted labels and expressions as javadoc comments in the source code. It is important to note that the BSL is very powerful: we created several versions of the same checking.

This section is organized as follows. For each pattern, we describe the conditions that need to be checked and the different versions of the checking using BSL as well as the part of the code annotated.

5.1. Single-Threaded Execution Pattern

The Figure 6 describes this pattern, implemented in order to be properly parsed by Bandera.

This concurrent design pattern was specially designed to prevent the following two concurrent problems.
(i) Two different instances of *TrafficSensor* can not notify the *TrafficSensorController* object at the same time. In other words, "If both calls execute at the same time, they produce an incorrect result. Each call to the *vehiclePassed()* method is supposed to increase the vehicle count by one. However, if two calls of this method execute at the same time, the vehicle count is incremented by one instead of two."[Grand98]; and
(ii) The *TrafficTransmitter* and the *TrafficSensor* can not access the *TrafficSensorController* at the same time. In both cases, one update will be lost if the described situation happens.

The absence of these problems is guaranteed by the use of the synchronized modifier in the *getAndClearCounter()* and *vehiclePassed()* operations.

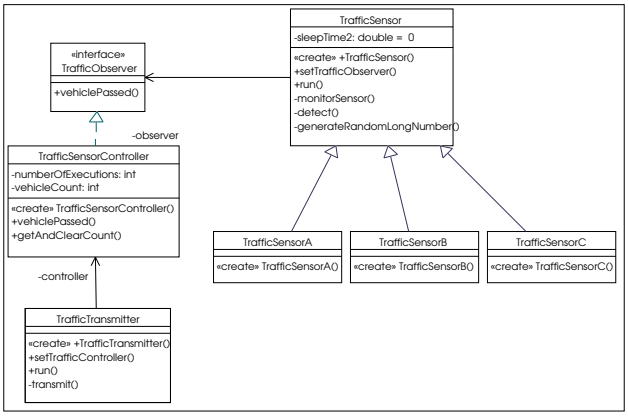

Figure 6 Single Threaded Execution Example

5.1.1. First Checking

Using the Bandera tool, we specified a logical predicate against which the model was checked. The predicate states that the method *vehiclePassed()* can not be invoked (is absent) between the invocation and the end (return) of the method *getAndClearCounter()*. Since these methods are not synchronized anymore, this situation could happen, and therefore Bandera should present a counter-example. The predicate, in BLS, is described as follows.

```
checkReadAndWriteAtSameTime2: forall
[s:TrafficSensorController]. {TrafficSen-
sorControl-
ler.vehiclePassed.vehiclePassedIsInvoked(s)
} is absent between
```

```
{TrafficSensorCotroler.getAndClearCount.
getAndClearInvoked(s)} and
 {TrafficSensorController.getAndClearCount.
getAndClearReturned(s)};
```

The annotation of the code used in this predicate is presented as follows.

```
/**
* @observable
* LOCATION [getAndClearCountCallLocation]
* getAndClearCountCall;
* @observable
*    INVOKE getAndClearInvoked;
* @observable
*    RETURN getAndClearReturned;
*/
public int getAndClearCount() { (…) }

/**
 * @observable
 * LOCATION[vehiclePassedCallLocation]
 * vehiclePassedCall: (numberOfExecu-
 tions<=1);
 * @observable
 * INVOKE vehiclePassedIsInvoked;
 * @observable
 * RETURN vehiclePassedIsReturned;
 */
public void vehiclePassed()
{ (…) }
```

5.1.2. Second Checking

Using this same relaxed model, the second checking specifies that the method *vehiclePassed()* can not be invoked (is absent) after the invocation of the method *getAndClearCounter()* until this second method finishes (returns). Again, these methods are not *synchronized*, therefore Bandera should present a counter-example. The BSL code is presented as follows.

```
checkReadAndWriteUsingAfterUntil: forall
[s:TrafficSensorController]. {TrafficSen-
sorControl-
ler.vehiclePassed.vehiclePassedIsInvoked(s)
} is absent after {TrafficSensorControl-
ler.getAndClearCount. getAndClearIn-
voked(s)} until {TrafficSensoControl-
ler.getAndClearCount. getAndClearRe-
turned(s)};
```

The same annotations used in the First Checking were used in this checking.

5.1.3. Third Checking

This last checking uses a different approach: a variable called *numberOfExecutions.* This variavle was declared and inserted inside the implementation of the method *vehiclePassed()*. It is responsible for counting the number of clients that are executing the operation simultaneously. Hence, a concurrent use of the operation is detected whenever this counter is greater than one.

The Bandera predicate that checks the code for this occurrence is presented as follows.

```
raceConditionWithAttr: forall
[c:TrafficSensorController]
.{TrafficSensorController .lessThanTwo(c)}
is universal globally;
```

The globally clause specifies that this predicate must be true in the full scope of the program.

In this example, the *numberOfexecutions* variable was incremented and decremented as follows.

```
public void vehiclePassed() {
       vehiclePassedCallLocation:
       this.numberOfExecutions++;
       this.vehicleCount++;
       this.numberOfExecutions--;
} // vehiclePassed()
```

It is also necessary to present the part of the code responsible for declaring the variable *numberOfExecutions* and the related annotation that describes the expression about this variable:

```
/**
 *  @observable
 *     EXP lessThanTwo:
 (numberOfExecutions <= 1;
 */
class TrafficSensorController implements
TrafficObserver {
       private int numberOfExecutions = 0;
       private int vehicleCount;
```

5.2. Producer-Consumer

The goal of this concurrent design pattern is to avoid that one “consumer” reads data that it was not produced yet. In this case, the “consumer” must wait for a “producer” to provide some data. When this happen, the “consumer” is allowed to read the data.

This problem can be solved if expressed using the wait() an the while loop in the class Queue. This class implements the shared resource where producers add data, and from consumers read data.

```
synchronized public TroubleTicket pull() {
   while (data.size() == 0){
       try {
           wait();
```

```
       } catch (InterruptedException e) {
       } // try/catch
   } // while
   TroubleTicket tkt =
          TrobleTicket)data.get(0);
    data.remove(0);
    return tkt;
} // pull()
```

Another problem that can happen in this pattern is the lost of one update, similar to the condition (i) in the Single-Threaded pattern. In this case, the pattern must avoid that two producers provide the information at the same time, because in this case one of the updates would be lost.

The absence of these problems is guaranteed by the use of the synchronized modifier in the method:

```
synchronized public void push(TroubleTicket
tkt) {
  …
} // push
```

After the specification, these models were checked with Bandera as described in the next subsections.

5.2.1. First Checking

The first condition can be checked is if we remove the code responsible for blocking the consumers while some producer produces some data. In the method presented above it is reflected as the bold part.

In order to check this program, we can create a precondition that establishes that a consumer can only call the pull method when a producer already produced some data. The new code to be tested, as well as the annotation in the code is the following:

```
/** @assert
 * PRE dataAvailable: data.size() > 0;
 */
public TroubleTicket pull() {
   while (data.size() == 0){
       try {
           wait();
       } catch (InterruptedException e) {
       } // try/catch
   } // while
   TroubleTicket tkt =
          TrobleTicket)data.get(0);
    data.remove(0);
    return tkt;
} // pull()
```

The gray code means that this part was removed from the program during the tests.

Since this is an assertion, there is no need to instantiate it as temporal pattern (absent or universal globally, for example) as we have been doing in the previous checking.

5.2.2. Second Testing

This test was created to check the second condition, i.e., if there are two concurrent calls to the method *push()*, then one update is lost. In other words, we want to guarantee that there is just one process executing the method push all the time.

Here, we used the same approach described in the third checking of the pattern Single-Threaded. A variable, called *numberOfConcurrentPushes*, was created and inserted inside the implementation of the method *push()*. This variable is responsible for counting the number of producers that are providing data and adding this objects to the queue simultaneously. Hence, a concurrent use of the operation is detected whenever this counter is greater than one.

The Bandera predicate that checks the code for this occurrence is presented as follows.

```
lostUpdate: forall
[q:Queue].{Queue.lessThanTwo(q)} is univer-
sal globally;
```

Again, the globally clause specifies that this predicate must be true in the full scope of the program. The part of the annotated Java source code that is related to this checking is presented below:

```
/**
 *   @observable
 * EXP lessThanTwo:
(numberOfConcurrentPushes < 2);
 */
class Queue {
    private int arraySize = 100;
    private TroubleTicket[] data;
    private int currentPosition = 0;

     private int
         numberOfConcurrentPushes = 0;

 public void push(TroubleTicket tkt) {
       numberOfConcurrentPushes++;
       TroubleTicket ticket = tkt;
       add(ticket);
       notify();
       numberOfConcurrentPushes--;
} // push(TroubleTicket)
```

5.3. Read-Write Lock

The goal of this concurrent design pattern is to coordinate the acquisition of locks and avoid that:

1. Two clients get the write lock at the same time, and

2. Prevent the acquisition of read locks from the moment when a write lock is acquired, until the moment it is relinquished.

This concurrency problem is implemented by the code in section 3.4.4. There are three critical sections guarded by the *synchronized* clause in the *writeLock()* operation, as well as *synchronized* modifiers in the declaration of the operations *readLock()* and *done()*.

Using the same idea of the other two patterns, in order to check this pattern, these clauses were relaxed. Invoke and return expressions are inserted in the headers of these operations. In order to check the synchronized clauses inside the operation, we can use the definition of labels. In Bandera, label conditions become true as soon as this point of the code is reached. Another approach is to use variables and counters as in the example in section 5.2.2.

We have to check if two or more concurrent processes reach a critical section at the same time. If it is possible to occur, we could reach inconsistent reads and lost updates.

As the checking expressions are analogous to the ones presented in the previous subsections, they are not presented here.

Here it is an example of the user of label inside the *writeLock()* operation. The labels *inWaitList*, waiting and *outWaiting* were inserted in the code.

```
/**
* @observable
*  RETURN writeLockReturn;
*  INVOKE writeLockCall;
*  LOCATION[inWaitList] waitList;
*  LOCATION[addingToList] addingList;
*  LOCATION[readingThread] readingTh;
*  LOCATION[waiting] waitingCond;
*  LOCATION[outWaiting]outWaitingCond;
*/
public void writeLock() throws
      InterruptedException {
    Thread thisThread;
     { // previous synchronized(this)
        if (writeLockedThread==null && out-
standingReadLocks==0) {
           inWaitingList:
           { writeLockedThread =
Thread.currentThread(); }
            return;
        } // if
        readingThread:
        thisThread =
Thread.currentThread();
        addingToList:
        waitingForWrite-
Lock.add(thisThread);
    } // synchronized(this)
    { // previous synchronized (thisThread)
        while (thisThread != writeLock-
edThread) {
           waiting:
           thisThread.wait();
        } // while
    } // synchronized (thisThread)
    { // previous synchronized (this)
        int i = waitingForWrite-
Lock.indexOf(thisThread);
        outWaiting:
        waitingForWriteLock.remove(i);
    } // synchronized (this)
} // writeLock
```

In this relaxed code, suppose the example in which two threads call writeLock() at the same time. If the write lock is already taken, they skip the first if guard. The thisThread variable is updated by one thread and, is modified, just after, by the other thread. The result is that only one thread is inserted at the waiting list, instead of two of them.

This situation can be detected with the following expression:

```
lostUpdate: forall
[q:ReadWriteLock].{ReadWriteLock.readingTh(
q)} is absent
after {ReadWriteLock.readingTh(q)}
until {ReadWriteLock.addingList(q)}
 globally;
```

In a similar way, other expressions can be composed.

6. Experiences using Bandera

This section describes the experience of the authors using the Bandera toolset. Basically, there are three subsections: problems identified, results and the features that we identified as promising in Bandera.

6.1. Problems Identified

We had many different problems with the parser used in Bandera. These problems were reported to the support group of Bandera, but up to now, we did not have any answer from them. These problems are briefly described below:

1. All source code must be in a unique file in order to be read and parsed by the Bandera. This approach seems to compromise the modularity of the code, and hinders the scalability of the tool for big Java programs.
2. Some classes that are declared in the code, but are not instantiated, generated a parsing error. Surpris-

ingly, abstract classes are supported.

3. The tool did not properly parse some classes from the package java.util such as ArrayList, found in the implementation of two of the design patterns analyzed. The Bandera also did not recognize the invocation of operations as *vector.size()* when *vector* is a *java.util.Vector* instance.
4. In some cases, the use of variables passed as parameters to methods generates errors in the parsing. For example, if we invoke the method sleep(globalVar) in a subclass of Thread, where globalVar is a global variable, the parser produces a parsing exception.
5. The static invocation of operations as Thread.sleep(Math.random()*100), used in our examples were not parsed.

Due to these problems, the patterns had to be rewritten several times in order to overcome these limitations of the tool. Special care was taken in order to keep the semantic of the examples after these changes. For example, in the Producer-Consumer example, instead of using the *java.util.ArrayList* class, we defined an alternative array of *TroubleTicket* objects.

Furthermore, since the tool does not handle correctly multiple instances of the same class, we had to define many subclasses of *TrafficSensor*, as described in Figure 6. This approach was inspired in the examples provided in the Bandera manual [MT01], in which class declarations are repeated in order to define many class types, with different names, but with similar implementation.

In fact, the following patterns were not tested: (i) the Read-Write Lock pattern was not tested because its implementation uses a static call: T*hread.getCurrentThread();* and (ii) the Producer-Consumer pattern was not tested because, after several tests, we were still not able to rewrite the code "properly", to be successfully parsed by Bandera.

During this rewriting phase, one could add errors to the code. Therefore, in order to avoid that each time we rewrote the code, we checked it with the JDK1.3 environment. Note that we are not claiming that this process was fault-tolerant or without errors, instead we are saying that we tried to avoid this problem. In fact, this is one of the limitations of our approach: the best solution is to formally prove that the programs are semantically equivalent.

Another problem identified was the Slicer. Although very useful in removing irrelevant parts of the code, it can also slice parts of the program that are essential for its execution. For example, it can slice classes that are part of the main method and are responsible, for example, for the instantiation of the main trhreads of the program to be analyzed. In this case, the model check results in success since the code analyzed does not have all the objects and the properties of the original one.

Bandera allows the specification of temporal logical predicates using templates. These templates are not simple and, sometimes, do not allow expressing simple statements. For example, we would like to check if two methods are called at the same time. The INVOKE annotation specifies that the following condition is true when the method is invoked. Then, we would like to be able to do something like:

```
/** @ observable
  * INVOKE firstMethodCalled;
*/
    public void firstMethod() { … }

/** @ observable
  * INVOKE secondMethodCalled;
*/
    public void secondMethod() { … }

/** @observable
  *  EXP BothMethodsCalled:
(firstMethodCalled && secondMethodCalled);
*/
```

However, we can not express this predicate because, there is not the temporal logic pattern supported by Bandera that could express this idea of simultaneous execution of operations.

6.2. Results

In this session we summarize our experience using Bandera and present some of the results we obtained.

Bandera seems to be a very powerful tool, however, one of the main claims of the authors, that they solve the semantic gap between artifacts and tools is still not true. The JJJC parser was not able to properly parse all the design patterns specified in Java, without modifications in the source code. Due to this problem, our code had to be modified many times, making this process of specifying existing code, a non-trivial task.

Due to the use of advanced techniques inside the Read/Write Lock code as invocation of static operations, we were not able to modify this example and, at the same time, keep the original semantic of the application. The use of the static invocation did not let us parse this code, although it was successfully compiled by the JDK1.3.

The Read/Write lock example and the Producer-Consumer could not be parsed in the non-sliced mode. The system generated a runtime exception during the model checking, which could be a result of an error in the conversion of the internal JJJT compiler to the formal language of the SPIN model checker.

The main results of our experiments are presented in Table 1 as follows:

	SingleThreadedExecution	Read/Write Lock	Producer-Consumer
Parsed Without Modification	No	No	No
Parsed With Modification	Yes	No	Yes
Success Checked W/O Code Slicer	Yes	No	No
Success Checked With Code Slicer	Yes	No	Yes
Detected Counterex. W/ Slicer	Yes	No	No
Detected Counterex. W/O Slicer	No	No	No

Table 1 A Summary of the results ruining the tests in the models.

We believe that many of the errors found during the execution of the tests presented in this paper, was a result of the immaturity of the tool. Bandera is still in 0.428 pre-alpha version.

According to [MT01], 600 property specifications were studied for this project: 94% were instances of the patters, in which 70% of the properties were 'universal' or 'response' properties. Our experience shows that these Bandera templates allow the specification of many temporal logic predicates. However, they are not simple to define and, to the extent of our knowledge, do not allow the expression of simple statements as the simultaneous execution of processes.

The authors also claim that Bandera was used to specify many systems, as a Java version of a space-craft control system, a scheduler for real-time systems, a generic framework for implementing multithreaded staged calculations and so on. In these examples, it identified some faults in some of the applications, but not identified seeded faults in others. This is in concert with our experience since Bandera was not able to produce counterexamples to the majority of our models.

6.3. Bandera Features

Despite of the errors found during the execution of the Bandera toolset, the tool presents several important features that will be discussed below.

One important feature provided by Bandera is its language for specifying abstractions is BASL. This language supports the specification of pre and post conditions, assertions and temporal properties for defining predicates using common Java control points and events (such as method invocation and return). It also allows the definition of expressions using Java code variables. In fact, we could express the checking for the Single-Threaded pattern in three different ways: sections 5.1.1 and 5.1.2 present two of three ways.

On the other hand, according to the Bandera's authors [MT01], "although temporal logic, 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 by software.". We completely agree with this position and, perhaps, that is the reason why we had problems using the temporal patterns provided by Bandera (see section 6.1.1). The process of compilation from the source code annotation, together with the program logic expressed by the Java code was not always successful. Many parsing and model checking errors were detected, some of them, related to the translation of the Bandera's intermediate language, Jimple, to the Promela language. If the Bandera user interface did not obligated us to use their specific temporal patters, perhaps, we could have been successful specifying and checking the properties of the concurrent patterns.

7. Conclusions

This paper presents the validation of three different concurrency design patterns. The patterns were implemented in Java and were extracted from [Grand98], while the validation was performed using the Bandera[CDH+00] toolset. Bandera supports the verification of properties specified in its specification language using model checking. Slicing and abstraction are provided in order to reduce the problem of state space explosion.

Unfortunately, our results were not useful since the version of Bandera that we used had several problems. In fact, we could not run tests in one of the three design patterns and in another one, we just could run partial tests.

The approach presented in this paper can be applied in other patterns using other model checkers like JavaPathFinder or SAL. We argue that the checking of design patterns can validate the advantages claimed by them. It also can help to increase the adoption of other patterns.

8. References

[ACL96] P.S.C. Alencar, D.D. Cowan and C. J. P. Lucena. A formal approach to architectural design patterns. In FME'96: Industrial Benefits and Advances in Formal Methods (Eds. M.C. Gaudel and J. Woodcook), pp. 576-594, Springer-Verlag LNCS 1051, 1996.

[Alexander79] C. Alexander. The Timeless way of

Builtind. Oxford University Press 1979.
[BMRSS96] Franch Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Sal. A System of Patterns. Chichester, U.K.: John Wiley & Sons, 1996.
[CDH+00] Bandera : Extracting Finite-state Models from Java Source Code, James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, Hongjun Zheng in Proceedings of the 22nd International Conference on Software Engineering, June, 2000, pages 439-448.
[CDH00] Bandera: a source-level interface for model checking Java programs, Corbett, J.C.; Dwyer, M.B.; Hatcliff, J. in Proceedings of the 22nd International Conference on Software Engineering, June, 2000, pages 762-765.
[CGP99] Edmund M. Clarke Jr., Orna Grumberg and Doron A. Peled. Model Checking, MIT Press, Cambridge, Massachusetts, 1999.
[DHR+01] Tool-supported Program Abstraction for Finite-state Verification, Matthew Dwyer, John Hatcliff, Robby Joehanes, Shawn Laubach, Corina Pasareanu, Robby, Willem Visser, Hongjun Zheng. in Proceedings of the 23rd International Conference on Software Engineering, May, 2001.
[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley, 1994.
[Grand 98] Grand, Mark. Patterns in Java: a catalog of reusable design patterns illustrated with UML, Wiley Computer publishing: John Wiley & Sons. Inc, 1998.
[Hol97] Gerard J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5):279-295, May 1997.
[HTWW96] Nevin Heintze, Doug Tygar, Jeannette Wing, and Hao-Chi Wong, Model Checking Electronic Commerce Protocols, Second USENIX Workshop on Electronic Commerce, 1996.
[Larmann98] Crig Larman, Applying UML and Patterns, Upper Saddle River, N. J.: Prentice Hall PTR, 1998.
[Lea97] Dough Lea. Concurrent Programming in Java. Reading, Mass.L Addison-Wesley, 1997.
[MT01] J. Matcliff and O. Tkachick. The Bandera Tools for Model-cheking Java Source Code: A User's Manual. March 7, 2001.
[PDW01] Finding Feasible Counter-examples when Model Checking Java Programs, Corina S. Pasareanu, Matthew B. Dwyer and Willem Visser, submitted to TACAS'2001.
[WF95] Jeannette M. Wing and Mandan Vaziri-Farahani, A Case Study in Model Checking Software Systems, Foundations of Software Engineering Conference, 1995.