Applying Model Checking in Java Verification

Klaus Havelund[1] and Jens Ulrik Skakkebæk[2]

[1] NASA Ames Research Center, Recom Technologies, Moffett Field, CA, USA
havelund@ptolemy.arc.nasa.gov, http://ase.arc.nasa.gov/havelund
[2] Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
jus@cs.stanford.edu, http://verify.stanford.edu/jus

Abstract. This paper presents our experiences in applying the Java PathFinder (Jpf), a recently developed Java to Promela translator, in the search for synchronization bugs in a Chinese Chess game server application written in Java. We give an overview of Jpf and the subset of Java that it supports and describe an initial effort to abstract and analyze the game server. Finally, we evaluate the results of the effort.

1 Introduction

Model checking has increasingly gained acceptance within hardware [15, 1] and protocol verification [13] as an additional means to discovering bugs. However, verifying programs is different from verifying hardware or protocols: the state space is often much bigger and the relationships harder to understand because of asynchronous behavior and a more complicated underlying semantics. The size and complexity of software pushes current formal verification technology beyond its limits. It is therefore likely that effective application of model checking to software verification will be a debugging process where smaller, selected parts of the software is model checked. The process will draw on multiple abstraction and verification techniques under user guidance, and is currently not well understood.

In order to investigate the challenges that software poses for model checking, we have applied the Java PathFinder (Jpf) [12], a recently developed Java to Promela translator, in the analysis of a game server application written in Java [8]. Promela is the modeling language of the Spin model checker [13]. We performed the abstractions by hand and translated the simplified Java program to Promela using Jpf. Although the example is not big (16 classes and about 1400 LOC), it is still non-trivial and is not written with formal verification in mind. In the process, we developed a suspicion of a deadlock bug in the software which was confirmed using Spin. Spin also produced an even simpler error scenario than we had found ourselves.

Few attempts have been made to automatically model check programs written in real programming languages. The most recent attempt to model check software that we are aware of is the one reported in [2], which also tries to model check Java programs by mapping into Promela. This work does, however, not handle exceptions, nor polymorphism (passing, e.g., an object of a subclass of a class C to a method requiring a C object as parameter). The work in [3] defines a translator from a concurrent extension of a very limited subset of C++ to PROMELA. The drawback of this solution is that the concurrency extensions are not broadly used by C++ programmers. Corbett [4] describes a theory of translating JAVA to a transition model, making use of static pointer analysis to aid *virtual coarsening*, which reduces the size of the model.

The VeriSoft tool [7] is a an exhaustive state-space exploration tool for detecting synchronization errors between processes. As a major advantage, it also verifies the actual code. However, since the visited states are not stored in the verification process, each time a new branch of possible execution is verified it has to rerun from the beginning, making it less efficient. Furthermore, each process is treated as a black box and properties have to be specified at the process interfaces. In contrast, model checking allows for more efficient verification (since states are saved along the way) and for specifying internal process properties, but requires more abstraction. It is still unclear which has the most advantages.

Data flow analysis has been applied to verify limited properties of concurrent programs, including JAVA [16]. These methods are useful for ruling out certain behaviors of the concurrent system but are much less precise than model checking. However, as we will argue later, they can potentially be useful in identifying problem areas for verification by state space exploration methods.

The JPF tool is introduced in Section 2 and the application to the game server is described in Section 3. We evaluate the results in Section 4 and conclude with a discussion in Section 5.

2 The JAVA PATHFINDER

JPF [12] is a translator that automatically translates a non-trivial subset of JAVA into PROMELA, the modeling language of the SPIN verification system [13]. JPF allows a programmer to annotate his JAVA program with assertions and verify them using the SPIN model checker. In addition, deadlocks can be identified. Assertions are specified as calls of methods defined in a special class (the `Verify` class), all of whose methods are static.

A significant subset of JAVA is supported by JPF: dynamic creation of objects with data and methods, static variables and static methods, class inheritance, threads and synchronization primitives for modeling monitors (synchronized statements, and the `wait` and `notify` methods), exceptions, thread interrupts, and most of the standard programming language constructs such as assignment statements, conditional statements and loops. However, the translator is still a prototype and misses some features, such as packages, overloading, method overriding, recursion, strings, floating point numbers, some thread operations like `suspend` and `resume`, and some control constructs, such as the `continue` statement. In addition, arrays are not objects as they are in JAVA, but are modeled using PROMELA's own arrays to obtain efficient verification. Finally, the libraries are not translated. Note that many of these features can be avoided by small modifications to the input code, and we expect the current version of JPF to be useful on a large class of software. The game server application described in Section 3 fits in the currently translated subset of JAVA with a few modifications.

We shall illustrate JPF with a small, but non-trivial, example. The example is inspired by one of five concurrency bugs that were found in an effort by NASA Ames to verify, using SPIN, an operating system implemented in a multi-threaded version of COMMON LISP for the DEEP–SPACE 1 spacecraft [11]. The operating system is one component of NASA's Remote Agent [17], an experimental artificial intelligence based spacecraft control system architecture. The bug, found before launch, is concerned with lock releasing on a data structure shared between several threads.

2.1 The Lock Releasing Problem Cast into JAVA

The Main Code The operating system is responsible for executing *tasks* on board the space craft. A task may for example be to run a camera. A task may *lock* properties (states to be maintained) in a *lock table* before executing, *releasing* these locks after execution. For example, one such property may be to "keep the thrusting low" during camera operation. For various reasons tasks may, however, get *interrupted* during their execution, and the particular focus here will be on whether all locks always get released in that case.

Figure 1 shows the `Task` class. Its constructor (the method with the same name as the class) takes three arguments: a *lock table* `t` which contains the locks, a property `p` (here just an integer) to be locked before the activity is executed, and the *activity* `a` to be executed by the task. An `Activity` object is required to provide an `activity()` method which when executed will perform a given task. Note that this is the way that JAVA supports higher order methods taking methods as arguments, and that JPF handles this.

The `run` method of the `Task` class specifies the behavior of the task; this method has to be part of any `Thread` subclass. The behavior is programmed using JAVA's `try-catch-finally` exception construct, which executes the locking and then the activity, unless an exception is thrown. This exception will be *caught* locally if it is a `LockException`, or thrown further out. In any case, the `finally` clause is always executed, releasing the lock.

Figure 1 furthermore shows the JAVA class `LockTable`, which models the table of all locks. It provides an array mapping each property (here a number between 0 and 2) into the task that locks it, or null otherwise. The method `lock` locks a property to a particular task, throwing an exception if the property has already been locked. The `release` method releases the lock again. These methods are defined as *synchronized* to obtain mutual safe access to the table when executed. A `LockTable` object will then work as a monitor, only allowing one thread to operate in it, that is: call its methods, at any time.

A Test Environment Typically an exception is thrown explicitly within a thread using the `throw(e)` statement, where `e` is an exception object (a normal

```
class Task extends Thread{
  LockTable t; int p; Activity a;

  public Task(LockTable t,int p,Activity a){
    this.t = t; this.p = p; this.a = a;
    this.start();
  }

  public void run(){
    try {t.lock(p,this);a.activity();}
    catch (LockException e) {}
    finally {t.release(p);};
  }
}

class LockTable{
  Task[] table = new Task[3];

  public synchronized void lock(int property,Task task)
  throws LockException{
    if (table[property] != null){
      LockException e = new LockException();
      throw(e);
    };
    table[property] = task;
  }

  public synchronized void release(int property){
    table[property] = null;
  }
}
```

Fig. 1. Task Execution and Lock Table

object of an exception class which may include data and methods). Alternatively, one thread S may throw a *ThreadDeath* exception in another thread `T` by executing `T.stop()`. This is exactly what the `Daemon` task shown in Figure 2 does. The daemon together with the `Main` class with the `main` method, that starts all threads, constitute an *environment* that we set up to *debug* the task releasing. The daemon will be started to run in parallel with the task, and will eventually stop the task, but at an unspecified point in time. The task is started with the property 1 and some activity not detailed here. The `assert` statement is "executed" after the termination of the task (the `join` method waits for the termination). The assertion states that the property is no longer locked. The assert statement is expressed as a call to the static method `assert` in the `Verify` class:

```
class Verify{
  public static void assert(boolean b){}
}
```

Since this method is static it can be called directly on the class without making an object instance first. It takes a Boolean argument, the validity of which will be checked. The body of this method is of no real importance for the verification since only the call of this method will be translated into a corresponding PROMELA assert statement. A meaningful body, like raising an exception for example, could be useful during normal testing though, but it would not be translated into PROMELA.

One can consider other kinds of `Verify` methods, in general methods corresponding to the operators in LTL, the linear temporal logic of SPIN. Since these methods can be called wherever statements can occur, this kind of logic represents what could be called an *embedded temporal logic*. As an example, one could consider statements of the form: `Verify.eventually(year == 2000)` occurring in the code. The major advantage of this approach is that we do not need to change the JAVA language, and we do not need to parse special comments. JAVA itself is used as the specification language. Note, that at this point, only the `assert` method is supported.

```
class Daemon extends Thread{
  Task task;

  public Daemon(Task task){
    this.task = task;
    this.start();
  }

  public void run(){
    task.stop();
  }
}
```

```
class Main{
  public static void main(String[] args){
    LockTable table    = new LockTable();
    Activity activity = new Activity();
    Task task = new Task(table,1,activity);
    Daemon daemon   = new Daemon(task);
    try {task.join();}
    catch (InterruptedException e) {};
    Verify.assert(table.table[1] == null);
  }
}
```

Fig. 2. Environment

The Error Trace When running the SPIN model checker on the generated PROMELA program, the assertion is found to be violated, and the error trace illustrates the kind of bug that was identified in the Remote Agent. The error highlighted is the following. Although the main activity of the task is protected by a `try ... finally` construct such that the lock releasing will occur in case of an interrupt (stop), the `finally` construct itself is not protected the same way. That is, if the task is stopped when within the `finally` construct, e.g. just before the lock releasing, the releasing never gets executed. The generated error trace exhibits exactly this behavior. This has been called the "unwind-protect" problem in LISP, obviously also present in JAVA, and is causing a real bug as illustrated here.

2.2 Translation to Promela

This section shortly describes the translation of Java into Promela. A more detailed description of the translation can be found in [12].

Classes and Objects Each class definition in Java introduces data variables and methods. When an object of that class is created with the `new` method, the Java Virtual Machine lays out a new data area on the heap for the data variables. Since Promela does not have a dynamic heap, a different solution has to be adopted. For each class an integer indexed array of some fixed static size is declared, where each entry is a record (*typedef* in Promela) containing the variables of the class. Hence, each entry represents one object of that class. A pointer always points to the next free "object" in the array. An object reference is a pair (c, i), where c is the class and i is the index of the object in the corresponding array (the pair is represented as the integer $c*100+i$). Inheritance is simply modeled by text inclusion: if a class B extends (inherits from) a class A, then each entry in the B array will contain A's variables as well as B's variables.

Methods Java method definitions are simply translated into macro definitions parameterized with an object reference – the object on which the method is called. That is, a Promela program is allowed to contain macro definitions, which are expanded out where called. For example, when a thread calls a method on an object, it "calls" the macro with the object identifier as parameter. The drawback with macros is their lack of local variables. Hence, Java method local variables have to be translated to global variables (within the calling thread), prefixed with their origin (class and method). Promela has recently been extended with inline procedures (motivated by some of our work presented in [11]), and these could be used instead, although it would not make a difference in the principles.

Threads Threads in Java are naturally translated to Promela processes. That is, any class being defined as extending the `Thread` class, such as `Task` and `Daemon` in the example, is translated to a `proctype`. The body of the process is the translation of the body of the `run` method. The main program (the `main` method) will be translated into an `init` clause in Promela, which itself is a special process.

Object Synchronization Java supports mutually exclusive access to objects via synchronized methods. That is, if a thread calls a synchronized method on an object, then no other thread can call synchronized methods on the same object as long as the first thread has not terminated its call. We model this by introducing a `LOCK` field in the data area of each object, hence in particular in each entry of the array representing the `LockTable` objects. This field will either be `null` if no thread has locked the object, or it will be equal to the process identification of the thread (Promela process) that locks it (via a call to a synchronized method). Some of the macros generated from our example, modeling locking, unlocking and synchronization, are shown in Figure 3.

The macro `LockTable_release` is the translation of the `release` method in the `LockTable` class. It executes the body of the method (a statement) with

```
#define LockTable_release(obj,property)
  synchronized(obj,LockTable_set_table(obj,property,null))

#define synchronized(obj,stmt)
  if
  :: get_LOCK(obj) == this -> stmt
  :: else ->
       lock(obj);
       try(stmt) unless {d_finally(unlock(obj))}
  fi

#define lock(obj)                      #define unlock(obj)
  atomic{                               set_LOCK(obj,null)
    get_LOCK(obj) == null ->
      set_LOCK(obj,this)}
```

Fig. 3. Synchronization Macros

synchronized access to the object. The `synchronized` macro executes the statement directly if the lock is already owned by the thread (equal to `this`), and otherwise it locks the object and executes the statement, finally releasing the lock after use. The `lock` macro sets the lock to `this` as soon as it gets available (equals `null` – note that expressions in PROMELA are blocking as long as they evaluate to false).

Exceptions One of the major capabilities of the translator is that it handles exceptions. Java exceptions are complicated when considering all the situations that may arise, such as method returns in the middle of `try` constructs, the `finally` construct, interrupts (which are exceptions thrown from one thread to another) of threads that have called the `wait` method, and the fact that objects have to be unlocked when an exception is thrown out of a synchronized method. PROMELA's `unless` construct seems related to an exception construct, except for the fact that it works "outside in" instead of "inside out", the latter being the case for JAVA's `try` construct. That is, suppose a JAVA program contains two nested `try` constructs as indicated in the left part of Figure 4.

```
try{                            {
  try{S1} catch (E x){S2}         S1 unless {catch(exn_E,x,S2)}
}                               }
catch (E y){S3}                 unless {catch(exn_E,x,S3}
```

Fig. 4. Exceptions in JAVA (left) and PROMELA (right)

If `S1` throws an exception object of class `E`, then this exception should be caught by the inner `catch` statement, and `S2` should be executed. On the right hand side of the figure is a simplified version of how we model exceptions in PROMELA. However, with the traditional PROMELA semantics of the `unless` construct, the outermost `catch` would be matched, and `S3` would be executed. Gerard Holzmann, the designer of SPIN, implemented a `-J` (J for JAVA) option giving the needed "inside out" semantics. Now, in the data area for a thread, in addition to the `LOCK` variable mentioned earlier, there is also an exception variable `EXN`. Throwing an exception, which is itself an object, is now modeled by setting the `EXN` variable to contain the exception object reference, and this will then trigger the `unless` statements. Even with addition of the `-J` option to SPIN, the translation is quite elaborate.

3 A Game Server Application

To investigate the practical usefulness of JPF, we have applied JPF in partial verification of a game server for Chinese Chess, developed by An Nguyen at Stanford University. The code is an older and simpler version of the code that is currently running. It was used for 3 weeks and was not written with formal verification in mind. The code was later rewritten because it was unstable and deadlocked too frequently.

Compared to industrial applications, the server code is fairly small: it consists of 11 JAVA classes of about 800 LOC in total. The client code is another 5 classes and 600 LOC. The code of each thread and method in the game server is not very complicated. The complexity stems from the communication interaction between multiple types of threads.

As expected, even though it is relatively small, the state size still drastically exceeds the limits of any model checker. It is possible that the example is manageable by tools like Verisoft [7]. This is, however, besides the point: we are using the application to investigate the limits and trade-offs for model checking, as well as studying viable approaches to abstraction.

3.1 Overview of the Code

The overall code is divided into server side code and client side code. The client code consists of JAVA Applets that are used to display the game boards, the game pieces, and to relay the user commands to the game server. There is no direct communication between players. All communication between players is done via the server.

The multiple players and game boards are naturally handled by a multi-threaded JAVA architecture in the server code. Although the client side code in effect is also multi-threaded to handle multiple requests from the user, the multi-threading is hidden in the browser application and each individual Applet is written as sequential code. We focus on the server code and leave verification of the multi-threaded user-interface code to future work.

Threads The thread structure is illustrated in Figure 5. At any point in time several game boards can be active, each served by a `ServerBoard` thread. For each participant of a game there is a `Connection` thread to handle the communication between the `ServerBoard` and the network connection associated with the player. Each `ServerBoard` can have multiple `Connections` associated with it (2 players and multiple observers). Inter-board communication messages are stored in a FIFO queue `ServerObjectQueue` and handled by the `ServerObjectHandler` thread. `ServerBoard` threads are the only producers of messages, and `ServerObjectHandler` is the only consumer.

`Server` is the main thread. It handles initialization and contains the main data structures of the server. Finally, there are two kinds of "vulture" threads for cleaning up global data structures that become obsolete when players log out: `Server` has an associated `ServerVulture` thread, and each `ServerBoard` thread has an associated `ConnectionVulture`.

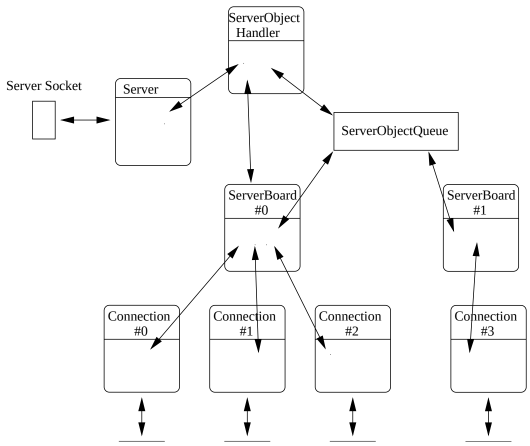

Fig. 5. A simplified illustration of the system. The boxes with rounded edges denote threads, the square boxes denote non-thread objects. For simplicity, we have not shown the vulture threads. In this example, `ServerBoard` 0 has three players associated with it, `ServerBoard` 1 has one. The arrows indicate the communication patterns between the different threads.

Commands A player has a predefined set of commands that can be sent via the network to the server. When a command arrives, it is stored in a separate FIFO queue in the associated `ServerBoard`. The `ServerBoard` thread then processes the commands one at a time.

The commands can be grouped into three classes: game commands, administration commands, and communication commands. Game commands are used to move the pieces around the board and to stop a game. Administration commands are used to create new boards, move between them, and log out. Finally, communication commands are used to communicate between the players, either to a few ("whisper") or to a larger crowd ("broadcast"). There are 12 different commands.

Each time a move is made, the other players in the game are notified by the `ServerBoard` broadcasting the information to its `Connection`s.

When a player creates boards, moves between boards, leaves the game, or sends out a global broadcast, the command is read by the `ServerBoard` and then stored in `ServerObjectQueue` for processing by `ServerObjectHandler`, which processes the commands one at a time. `ServerObjectHandler` handles global broadcasts by calling local broadcast commands in each of the `ServerBoard` threads.

3.2 Abstraction

We focussed on abstracting out a synchronization and communication "skeleton" and removed game code that was not important for detecting synchronization errors. In the process, we formed an idea of a potential deadlock in the code by studying the order in which threads obtained locks and accessed shared data. However, we were not positive that the deadlock was present until it was demonstrated by the model checker. Subsequently, we therefore targeted the abstractions at demonstrating this bug.

Focused on demonstrating the potential deadlock, we manually cut away big chunks of the program by interleaved application of static slicing and over- and under-approximations, known as the "meat-axe" technique in [11].

Static Slicing Given a marking of a statement in the program, static slicing techniques [18] will produce the subset of the program that corresponds to the *cone-of-influence*. In *forward* slicing, the output is the part of the program that is potentially influenced by the marked statement. In *backward* slicing, the output is the part of the program that potentially can influence the values of the variables at the statement of interest.

We used forward slicing to remove the part of code that was potentially irrelevant to the verification task. The relevant code was all the code that was not directly game related. We marked irrelevant variables, methods, and classes, and used this as a guidance to (manually) forward-slice the program from the unmarked variables. For example, the `Board` class contains the data structures related to the Chinese Chess board and the positions of the pieces and was removed; methods related to communication with the player process were also removed; and fields containing a player's role in the game were likewise removed. Of the 11 server code classes, two of them could be removed fully.

We did not use backward slicing in the abstraction. The deadlock scenario that we were interested in was caused by thread interaction. Marking a particular statement involved in the deadlock scenario would clearly have given a slice that was too large/imprecise for our purpose.

Over-Approximations Over-approximations are obtained from the original program by introduction of non-determinism in the control flow such that the abstracted model will exhibit more behaviors than the original program. Over-approximations are sound in the sense that when checking safety properties, if no bugs are found in the abstracted model, no bugs exist in the original program. Counter examples, on the other hand, are not necessarily true counter examples in the original model.

Over-approximations were used many times in the abstraction of the game server code. We illustrate this with two examples. First, it was useful to abstract the control conditions marked in the slicing process. For instance:

```
if (...something game related...) then {...server.broadcast(...);...}
```

was changed to

```
if (nondet.flag) then {...server.broadcast(...);...}
```

where `nondet.flag` is a flag in a new class `NonDet` that is non-deterministically set and reset by the model checker in all possible interleavings with other threads.

Second, we abstracted the types of messages that the threads pass around to a few. The messages are encoded in strings, where the first characters of the string contains the command, e.g., "`/broadcast`", "`/open`", and "`/join`". `ServerBoard` and `ServerObjectHandler` determine the type of messages by looking at this string prefix. Typically, this is done in a nested if-then-else structure:

```
if (line.startswith("/broadcast") then {...}
else if (line.startswith("/talk") then {...board.processCommand(...);...}
else if (line.startswith("/whisper") then {...board.broadcast(...);...}
else {...}
```

We abstracted the message strings into a record with a type and a value. From code inspection, we found 3 commands that were related to the concurrency behavior we were interested in pursuing and mapped the message type into something equivalent to an enumerated type with values `broadcast`, `open`, `join`, and `other`. The latter was introduced by us to capture the remaining types of commands. The nested control structure was then modified to be

```
if (line.type=broadcast) then {...}
else if (line.type=other) then {...
        if (nondet.flag) then board.processCommand(...);
        else board.broadcast(...);
        }
else {...}
```

where non-determinism was introduced to model all possible behaviors of external events associated with the commands mapping to `other`.

This last abstraction is a form of abstract interpretation [5] where the user decides which behavior to keep in the collapsed code.

Under-Approximations Under-approximations are obtained by removing code (and state that it depends on) from the program, with the effect of reducing the possible behaviors of the original program. Under-approximations may not be sound for safety properties, since the code that is removed could be causing bugs in the original program – bugs that are not caught when verifying the abstract model. However, when checking safety properties, if a bug is found in the under-approximation, it will also be present in the original model. Under-approximation is a very useful technique when narrowing the search space to a particular part of the code.

We used under-approximations many times in the game server verification. First, we ignored some exceptions of the program, including the exception handling. In the part of the server code that we were interested in, the exception handling was of little importance. Second, initially we limited the number of threads to two `ServerBoard` threads and two `Connection` threads; and the number of messages from the players to consider were limited to 3. Third, we inserted 3-4 extra synchronization points to sequentialize the thread behaviors. This limited the number of possible interleavings that SPIN had to consider and more quickly guided it towards the deadlock scenario.

3.3 Verification

We combined the abstracted JAVA classes into a file, translated it using the JPF, and tried to run it through SPIN. It took several more cycles of abstraction before the model was sufficiently manageable for SPIN to find the deadlock bug. We inserted JPF print statements in appropriate places in the JAVA code. This information made it easy to interpret SPIN's Message Sequence Chart [13] description of the counter example scenario.

The bug we confirmed was a deadlock caused by cyclic waits between a number of threads. It involved only three threads to cause the deadlock. Using SPIN we were able to find an unknown and significantly simpler deadlock scenario with two threads. We will present this simpler example below.

The deadlock may occur when the `ServerObjectQueue` becomes full. It happens when a `ServerBoard` processes incoming messages from its `Connections`, while the `ServerObjectHandler` wants to broadcast a message to the same connections.

The deadlock, illustrated in Figure 6, may arise as follows: The `ServerBoard` thread has obtained exclusive access to its connections by locking the `connections` vector, which stores references to the connections. Next, it goes through a loop where it processes the incoming connection messages, one at a time. Some of the messages must be processed by `ServerObjectHandler` and the `ServerBoard` thread will therefore want to put these messages in the `ServerObjectQueue`. However, if the queue runs full in this process it will busy-wait on the queue, while holding the lock on `connections`.

If the `ServerObjectHandler` is simultaneously processing a command that causes a broadcast of a message to the connections of the `ServerBoard` thread, it

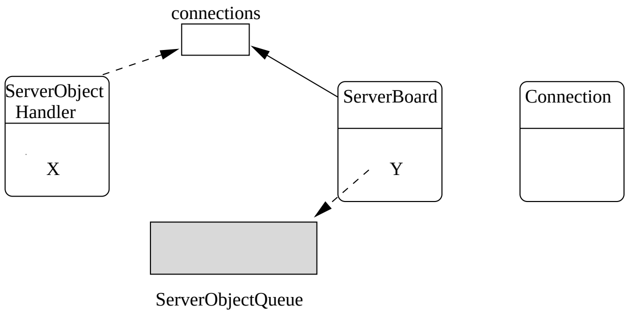

Fig. 6. Deadlock scenario: `ServerBoard` waits for a free slot in the queue, and `ServerObjectHandler` waits for the lock on `connections` to be released. Dashed lines indicates wait, the solid line indicates the lock that has been obtained. X and Y are the messages waiting to be processed.

will try to obtain the lock on `connections` by a `synchronize`. However, the lock will never be released by `ServerBoard` and a cyclic wait has been established.

Note that the deadlock is caused by a simultaneous wait on a Java lock, using the `synchronized` statement, and a busy wait on the queue. The code for enqueueing and dequeueing was written by the developer himself and is not part of the Java Virtual Machine (JVM).

4 Evaluation

The analysis itself took a month till the bug was confirmed. However, during this period much time was used on understanding the code and improving Jpf to translate the subset of Java that the game server covers. In future applications of the Jpf, this will be reduced significantly. At present, we have not yet completed the verification of all parts of the code. More effort would be required to fully verify the code.

Static slicing and abstract interpretation [5] turned out to be useful, but did not produce sufficiently small models. Under-approximations were necessary to narrow down the search space to focus on the bugs that we were interested in finding. This confirms the experiences from the Remote Agent analysis [11]. In general, user guidance was crucial for identifying parts of the code were the techniques could be applied.

Concerning the Jpf technology itself, current model checkers, including Spin, have been constructed with hardware and protocols in mind and require static memory allocation. That is, they do not support the dynamic nature of object oriented software, more specifically the `new C(...)` construct which generates an object from a class `C`. Jpf currently generates statically sized global Promela arrays that hold the state of the objects, causing wasted memory when only few objects are allocated while the arrays are initialized to hold a larger number. In addition to wasted memory, a second problem with this array solution is the intricate name resolution machinery required to search for the right array when looking up variables. This is caused by the class subtyping in JAVA (*polymorphism*), where the class of an object cannot be statically decided. Consider for example a method taking a `C` object as parameter. It can be applied to any object of any subclass of `C`. As a consequence, variable lookups consist of conditional expressions searching through the subclasses. In an early version of JPF this in fact caused a code blow up that made the C compiler gcc fail on the C code that SPIN generates from the resulting PROMELA program. A related issue is the lack of variables local to "methods" in PROMELA. Macros have no concept of locality, leading to further machinery in the translation of variables local to JAVA methods.

JPF does not handle all of JAVA, major substantial omissions being recursion, strings and floating point numbers, and perhaps most importantly: the JAVA libraries. Recursion requires more elaborated modeling of the Java Virtual Machine, for example in terms of a call stack, which however will be costly. Alternatively PROMELA's process concept can be used to model recursive methods, but this solution appears to be time inefficient. It was tried, but undocumented, in the work described in [11]. The translation of JAVA exceptions is quite sophisticated, handling all the special circumstances possible. Gerard Holzmann helped in providing a new semantics of PROMELA's `unless` construct, but even in that case the translation is "clever". JPF is currently being improved to translate a bigger subset of JAVA, and to handle an extension of the `Verify` class with linear temporal logic operators.

The translation does not cater for garbage collection. Normally garbage collection (or the lack thereof) is hidden from the programmer, and should not effect the functionality of a program. However, the effectiveness of the verification may be improved by regarding states with garbage equivalent to states without garbage. Garbage collection seems absolutely non-trivial to handle.

5 Discussion and Future Work

Successful application of model checking in program verification will involve an iterative process of abstraction and verification and will draw on multiple techniques for abstracting manageable models from the original program. The sheer size and complexity of the software-under-verification pushes current verification technology to its limits and the goal of the abstraction process is to fit parts of the verification problem within boundaries of feasible verification. In order to make model checking of programs scalable, program abstraction needs to be better understood, and supported with automated tools. Below we will briefly discuss issues relevant to a future verification environment.

Static Analysis Static analysis such as data flow analysis [16] can be used to identify potential problem areas. The techniques must be able to recognize complex dependencies such as deadlocks involving cyclic wait on data as well as locks (as our example illustrated). In general, automated guidance is crucial to guide the effort to certain parts of the software that is potentially buggy. Under-approximations (such as partial evaluation [14] and other program specialization techniques [10]) remove behaviors from the original model. The verification engineer must apply these cautiously in order not to remove behavior that could lead to bug scenarios.

User Guided Abstractions by Code Annotation User-interaction, potentially aided by heuristics, is crucial for effective application of the abstraction techniques. Code annotations are a potential means of capturing the higher-level understanding of the programmer at the time of development and could provide hints to the abstraction tool. For instance, the game server code could contain information that would guide an abstraction tool to understand the strings that are being passed around as actual message types. Alternatively, predefined coding styles in the form of design patterns could be recognized by an abstraction tool. It is also possible that automatic abstraction techniques suggested by Graf and Saidi [9], extended with string pattern matching, could be applied to determine the enumeration type.

Libraries One of the decisions to make in the abstraction process is deciding the boundaries of the program to be verified and how to model its environment. For the game server, the obvious boundaries were the network interface. However, in other cases the correctness properties may be specified in terms of the client applications. In this case, the network needs to be modeled also. An effective verification environment needs pre-written environment models of commonly used libraries. The game server code, for instance, uses the JAVA `Vector` class. By studying the JVM specification we were able to write simple stubs that modeled the class sufficiently for our use. In general, such environment modules must be predefined to save the verification engineer time and effort.

Compositional Approaches An alternative solution to dealing with scalability is compositional model checking [6], where only smaller portions of code are verified at a time, assuming properties about the "rest of the code", and where the results are then composed to deduce the correctness of larger portions of code. This approach is not problem free though since composing proofs is non-trivial and often requires iteration as does induction proofs (no silver bullet around). A practical solution is so-called *unit testing* where a class or a small collection of classes are tested by putting them in parallel with an aggressive *environment*. The Remote Agent analysis presented in Section 3 is an example of this. It's likely that model checking at least can play an important role in program verification at this level.

Acknowledgments The authors wish to acknowledge the following persons at NASA Ames for their contributions in terms of ideas and support: Tom Pressburger, Mike Lowry, John Penix and Willem Visser. At Stanford we would like to thank David Dill for discussions and comments. The second author is sponsored by NASA contract number NAG2-891: An Integrated Environment for Efficient Formal Design and Verification.

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.
2. R. Iosif C. Demartini and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998.
3. T. Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, LNCS 1384*, April 1998.
4. J. C. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis, Clearwater Beach, Florida.*, March 1998.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
6. W. de Roever, H. Langmaack, and A. Pnueli (Eds.). *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany. LNCS 1536*. Springer-Verlag, September 1997.
7. P. Godefroid. Model checking for programming languages using Verisoft. In *ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.
8. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. A-W, 1996.
9. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV*. Springer-Verlag, June 1997.
10. J. Hatcliff, M. Dwyer, S. Laubach, and D. Schmidt. Stating static analysis using abstraction-based program specialization, 1998.
11. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998.
12. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. NASA Ames Research Center. To appear in the *International Journal on Software Tools for Technology Transfer* (STTT), 1999.
13. Gerard Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.
14. N. D. Jones, editor. *Special Issue on Partial Evaluation, 1993 (Journal of Functional Programming, vol. 3, no. 4)*. Cambridge University Press, 1993.
15. K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
16. G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent java programs. Technical Report 98-22, Computer Science Department, University of Massachusetts at Amherst, April 1998.
17. B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.
18. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.