Practical Application of Model Checking in Software Verification

STREAM: FOUNDATIONS AND METHODOLOGY

MINI TRACK: MODEL CHECKING

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://ic-www.arc.nasa.gov/ic/projects/amphion
[2] Computer Systems Laboratory, Stanford University, Stanford, CA 94305, USA
jus@cs.stanford.edu
http://verify.stanford.edu

Abstract. This paper presents our experiences in applying the JAVA PATHFINDER (JPF), a recently developed JAVA to SPIN translator, in the finding of 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 the abstraction and verification of the game server. Finally, we analyze the results of the effort. We argue that abstraction by under-approximation is necessary for abstracting sufficiently smaller models for verification purposes; that user guidance is crucial for effective abstraction; and that current model checkers do not conveniently support the computational models of software in general and JAVA in particular.

1 Introduction

Current trends in software and hardware development indicate that low-cost, embedded systems will soon permeate our every day living. Embedded, reactive software applications will be ubiquitous, and distributed, networked servers will provide instant information access. Software in both embedded processors and server systems will increasingly be multi-threaded, since it must respond to events from the user and environment at any time. In both cases, bugs in the software can be costly, as the software is distributed in many copies.

Finding concurrency bugs in multi-threaded software by testing is a difficult task, since it not only involves providing particular input test data but also requires the underlying operating system scheduler to schedule the different threads in a particular way, in order to expose the bug. Even worse, the thread schedulers for different platforms may vary and bugs may appear in actual deployment that were not exposed on the test platform.

* Contact: Phone: (650) 604-3366, Fax: (650) 604-3594

Model checking has increasingly gained acceptance within hardware [5, 16, 2, 1] and protocol verification [14] as an additional means to discovering bugs. In contrast to testing, it exercises the model to be verified in an exhaustive fashion. To check for concurrency bugs, it will exercise the threads in all possible interleavings, and is thus able to find the bugs independently of particular scheduling algorithms.

However, verifying programs is different from verifying hardware or protocols: The state 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. This process is currently not well understood.

In order to investigate the challenges that software poses for model checking, we have applied the JAVA PATHFINDER (JPF) [13], a recently developed JAVA to Spin translator, in the verification of a Chinese Chess game server application[1] written in JAVA [9]. We performed the abstractions by hand and translated the simplified JAVA program to Spin 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. Currently, we have not verified the whole application (it is questionable if it is at all useful) but we will try other parts of the program that show potential for bugs.

This paper provides an overview of JPF and presents our experiences and lessons learned. We argue that (1) abstraction by under-approximation is necessary for abstracting sufficiently smaller models for verification purposes, (2) user guidance is crucial for effective abstraction, and (3) that current model checkers do not conveniently support the computational models of software in general and JAVA in particular.

Related work is described in Section 2. The JPF tool is introduced in Section 3 and the application to the game server is described in Section 4. We analyze the results in Section 5 and conclude with a discussion in Section 6.

2 Related Work

Few attempts have been made to automatically verify programs written in real programming languages. The most recent attempt we can mention is the one reported in [3], 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

[1] `http://www.cchess.net`

as parameter). The work in [4] 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.

The VeriSoft tool [8] 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 state space is too large, 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. 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, incl. JAVA[19, 18]. 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. Corbett [6] 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.

3 The JAVA PATHFINDER

JPF [13] is basically a translator that automatically translates a non-trivial subset of JAVA into PROMELA, where PROMELA is the programming language of the SPIN verification system [14]. SPIN supports the design and verification of finite state asynchronous process systems. PROMELA is a simple multi-threaded programming language with non-deterministic guarded commands. Processes communicate either via shared variables or via message passing through buffered channels. Properties to be verified are stated in the linear temporal logic LTL. The SPIN *model checker* can automatically determine whether a program satisfies a property, and in case the property does not hold, it generates an error trace.

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. Temporal logic properties are specified using calls to 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, 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 overwriting, recursion, strings, floating point numbers, static variables and static methods, 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. Note that many of these features can be avoided by small modifications to the input code. In addition, the tool is currently improved.

Despite the omissions we expect the current version of JPF to be useful on a large class of software. The game server application described in Section 4 fits in the current subset 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 [12]. The operating system is one component of NASA's Remote Agent [20], 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.

3.1 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* `t` 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.

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 exception construct, the general form of which is:

```
try S1 catch(E x) S2 finally S3
```

where each `S1,S2,S3` is a block (a statement or a sequence of statements enclosed by {...}) and `E` is an exception type (class), and `x` is a variable. The body `S1` of the `try` statement is executed until either an exception is thrown or it finishes successfully. If an exception is thrown, the `catch` clause is examined in order to

```
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);};
  }
}
```

Fig. 1. Task Execution

find out whether the thrown exception is of the corresponding class `E` or of a subclass thereof. If this is the case the corresponding block `S2` is executed with `x` being bound to the thrown exception. If the exception is not of type `E`, the exception "flows out" of the `try` statement into an outer `try` that might handle it. In any case, the `finally` clause `S3` is always executed. This happens no matter how the completion was achieved, whether normally, through an exception, or through a control flow statement like `return`. The code in Figure 1 shows how the lock is set, the activity is executed, and then finally, the lock is released.

Figure 2 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 Normally an exception is thrown explicitly within a thread using the `throw(e)` statement, where `e` is an exception object (a normal object of an exception class which may include data and methods). However, 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 3 does. In fact, the daemon together with the `Main` class with the `main` method representing the main program constitutes 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 having joined the task, when the task terminates that is. The assertion states that the property is no longer locked.

```
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. 2. The Lock Table (`LockException` is defined elsewhere)

The assert statement is expressed as a call to the static method `assert` in the `Verify` class shown in Figure 4. The fact that this method is static means that we can call it directly on the class without making an object instance first. It takes a string argument being printed if the assertion, given as the second Boolean argument, is violated. 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.

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 problem is that 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, for example just before the lock releasing, the releasing never gets executed. The generated error trace shows exactly this behavior. This is in folklore called the "unwind-protect" problem in LISP, obviously also present in JAVA, and causing real bugs as illustrated here.

```
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("Released",table.table[1] == null);
  }
}
```

Fig. 3. Environment

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

Fig. 4. The `Verify` Class

3.2 Translation to PROMELA

This section shortly describes the translation of JAVA into PROMELA. A more detailed description of the translation can be found in [13].

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 [12]), 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, in this case most interestingly in the `LockTable` array. 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 modeling locking, unlocking and synchronization are shown in Figure 5.

```
#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. 5. Synchronization Macros

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 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 6.

```
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. 6. 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 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, an object that is, is now modeled by setting the `EXN` variable to contain the exception object reference, and this will then trigger the `unless` statements. Even with this modification, the translation is quite elaborate.

4 A Game Server Application

4.1 Background

We have applied JPF to the verification of a game server for Chinese Chess, developed by An Nguyen, a Stanford student. 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. However, as we shall describe below, the size of the code does not give an accurate indication of the complexity of thread interaction.

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 [8]. 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.

4.2 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 multi-threaded user-interface code to future work.

Threads The thread structure is illustrated in Figure 7. 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`.

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 FIFO queue in the associated `ServerBoard`. The `ServerBoard` thread then processes the commands one at a time.

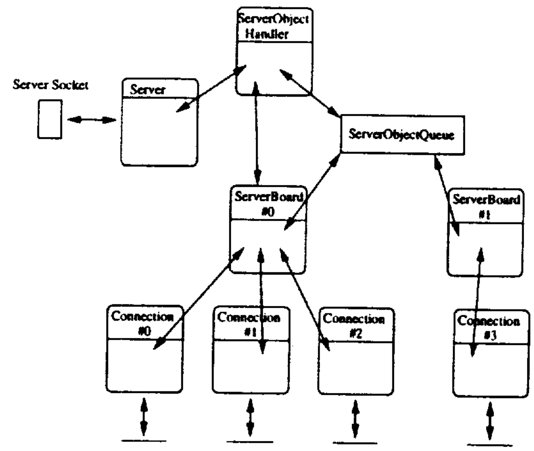

Fig. 7. A simplified illustration of the system. The boxes with rounded edges denote threads, the square boxes denote non-thread objects. For simplicity, we have not shown the vulture threads. In this example, `ServerBoard 0` has three players associated with it, `ServerBoard 1` has one. The arrows indicate the communication patterns between the different threads.

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 `Connections`.

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 sending local broadcast commands to each of the `ServerBoard` threads.

4.3 Abstraction

Since it was obvious that the code was too large to be verified by JPF directly, we started a manual abstraction process. The obvious challenge was to decide what parts of the software to abstract away. We did it in a two stage process. First, since we were interested in the synchronization bugs, we decided to remove all the code that was related to the Chinese Chess game itself. As long as the synchronization and communication "skeleton" was left intact, the particular details of the game were not important for detecting synchronization errors.

Second, in the process of verifying the game server, 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. The suspicion of a deadlock was extremely helpful in guiding our abstraction process to find a deadlock quickly. Without the guidance, we would have been hesitant to abstract away too much detail, in fear of abstracting away potential error scenarios.

The manual abstraction process was focused on demonstrating the potential deadlock and brutally cut away big chunks of the program, using the "meat-axe" technique as described in [12]. We applied a number of techniques simultaneously. In retrospect, they can be divided in a number categories, as described in the following.

Static Slicing Given a marking of a statement in the program, static slicing techniques [22] will produce the subset of the program that corresponds to the *cone-of-influence.* 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. In *forward* slicing, the output is the part of the program that is potentially influenced by the marked statement.

Static slicing is traditionally motivated by classical compiler theory such as dead-code elimination and debugging. For verification purposes, it can be used to remove the part of code that is potentially irrelevant to the verification task. We used forward slicing in our verification. 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 rôle in the game were likewise removed. Of the 11 server code classes, two of them could be removed fully.

Backward slicing could also potentially be of use in verification. However, as we shall describe later, the deadlock scenario that were were interested in were a combination of thread interaction. Marking a particular statement involved in the deadlock scenario would have given a slice that was too large/imprecise for our purpose.

Forward slicing in isolation turned out not to be of much use, however. It was too conservative and thus included irrelevant code in the potential slice. We combined it with more approximate methods to achieve smaller models.

Approximations of a program are simpler models of the program, where the set of possible behaviors have been altered. In the verification we used both over-approximations and under-approximations.

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. For instance, when variables used in an if-then-else condition have been abstracted away, the whole condition can be replaced by a non-deterministic choice. 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[2] 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(...);
   ...
   if (nondet.flag) then 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 [7] where the user decides which behavior to keep in the collapsed code. Having an abstract interpretation tool available would be a significant help. However, this an example where user interaction is crucial.

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, an obvious under-approximation was to initially ignore all exceptions of the program, including all 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.

[2] We implemented this using the naturals and constants representing the elements.

4.4 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 [14] 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 8, 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 `ServerObjectHandler` is simultaneously processing a command that causes a broadcast of a message to the connections of the `ServerBoard` thread, it 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.

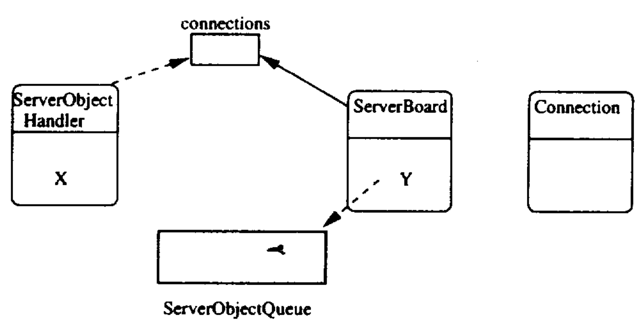

Fig. 8. Deadlock scario: `ServerBoard` waits for a free slot in the queue, and `ServerObjectHandler` waits for the lock on `connections` to be released. Dashed lines indicates wait, the solid line indicates the lock that has been obtained. X and Y are the messages waiting to be processed.

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 JVM.

The verification itself took a month till the bug was confirmed. However, during this period much time was used on understand the code and improving the JPF to handle the subset that the game server uses. In future applications of the JPF, this will be reduced significantly.

Currently, we have not yet completed the verification of all parts of the code. By studying the dependencies, however, we are not convinced that there are more deadlocks to be found, although this is the opinion of the developer.

5 Analysis

Formal Verification as a Debugging Process As illustrated above, 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 original program 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.

User-interaction, potentially aided by heuristics, is crucial for effective application of the abstraction techniques. Without restrictions on the way the code is written, it is not likely that abstractions can be fully automated. For instance, it is not likely that a tool could automatically recognize that the strings that encode communication messages have a certain pattern and can be abstracted into a record with a "type" and "data" field, where the type itself is an enumeration type. It is possible, however, that automatic abstraction techniques suggested by Graf and Saidi [10], extended with string pattern matching, could be applied to determine the enumeration type.

Alternatively, code annotations could provide hints to the abstraction tool. This is a way of capturing the higher-level understanding of the programmer at the time of development. 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.

We imagine an abstraction framework that supports the abstraction techniques, much in the form that theorem provers today support the verification engineer. Effective debugging using such an environment will be guided by user experience, heuristics, and automated static analysis techniques. These will be used to focus on different classes of bugs separately. For each class of bugs, an effective way of controlling verification complexity is to initially narrow the model behaviors to a manageable size, followed by gradual expansion of the set of behaviors.

In the process of verifying the game server, it quickly became obvious that automated guidance is essential for the abstraction/verification iteration to succeed. Under-approximations remove behaviors from the original model and without a focus, the verification engineer might be hesitant to perform such abstractions in fear of removing potential bug scenarios. Automated guidance could guide the effort to certain parts of the software that is potentially buggy.

Static analysis such as data flow analysis [19, 18], etc. could be useful here. However, as we have seen above, deadlock scenarios can occur not only with cyclic waits on locks, but also cyclic waits on data, such as buffers. Static techniques must be able to recognize such situations.

In contrast to finite specifications, criteria for deciding when to stop the debugging process are important. In many applications, the number of threads that can be started and objects that can be created is in principle only bound by the the word size and the size of the memory on the server computer. Finding and proving the minimum number of processes necessary for a "full" verification is a difficult task in the presence of complex software architectures. It is questionable if this is a at all a feasible approach when verifying big software architectures. It might even be that the size and speed of the verification hardware will limit the number of processes before the minimum is reached.

Abstraction Techniques As described above, 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 verification [12], that over-approximation techniques (such as abstract interpretation [7]) do not produce sufficiently small models. Under-approximations can be obtained by program specialization such as partial evaluation [15] and other program specialization techniques [11], as well as by taking advantage of domain specific information provided by the user. Other techniques we used were limiting the number of tasks and inserting extra synchronization to sequentialize task execution. This cut down the search space significantly.

Since we are approaching the verification as a debugging process, abstraction algorithms that cover only subsets of the programming language are also potentially useful. It is noteworthy that the code of each thread and method in the game server is not very complicated. There is, for instance, no recursion or complicated data structures. The complexity instead stems from the use of multiple levels of threads that communicate and interact in complicated communication patterns.

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. Other environment modeling comes in when the software uses pre-defined libraries. The game server code, for instance, uses the JAVA `Vector` class. By studying the JVM spec 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.

Verification Once the program abstraction activity ends, JPF is applied, and the resulting PROMELA program is model checked by SPIN. Verifying the game server using the JPF translator pushed the limits of SPIN, especially in terms of the size of the C code that SPIN generates from the resulting PROMELA program, sometimes causing gcc to fail. The reason for this is essentially the mismatch in concepts between the model checker and the programming language concepts. This is in spite the fact that SPIN appears to be one of the better suited target model checkers due to its dynamic process creation construct and asynchronous (interleaved) process model in particular – and flexible C–like programming language like notation in general. Generally, one has to model the Java Virtual Machine in the model checker, and the translator has resemblance to a compiler.

First of all, current model checkers 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. 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.

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 does not handle all of JAVA, major substantial omissions being recursion, strings and floating point numbers. 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 [12].

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 though.

A general question is whether translating from a high level source language like JAVA to a high level target language like PROMELA is an advantage when the two languages do not fit exactly. An example is the "clever" translation of exceptions. Alternatively one can consider translating JAVA byte code instead. This would have the merit that possibly the translation would be simpler, and potentially more reliable since each translation rule would be "simple" and local. The disadvantage might be that verification at the byte code level will be inefficient due to the finer grain atomicity of byte code.

SPIN was chosen as target system due to its high level programming language like notation, its dynamic process creation construct, and its focus on an interleaved process model. Since the source and target languages do not fit exactly, the advantage of a high level target language is slightly smaller than expected. Other model checkers may be applicable, such as SMV [16] and MURPHI [17]. However, SPIN's dynamic process creation and interleaved process model, with efficient verification algorithms for this model, seems to be of value.

Compositional Approaches An alternative solution to deal with scalability is compositional model checking [21], 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.

6 Discussion

In order to make model checking of programs scalable, program abstraction needs to be better understood, and supported with automated tools. It's likely that such abstraction environments will be user guided. One can imagine special programming styles, design patterns, and program annotations that would support the abstraction activity. Especially object orientation could become useful in that data (to be abstracted) are defined together with their methods (to be changed as a result of the abstractions). Furthermore, model checkers need to deal explicitly with such issues as dynamic memory allocation, object references, and garbage collection.

Even in the current situation, where abstraction tools are not available, we believe that model checking is ready for unit testing of programs, giving the programmer a chance to put his sub programs (classes) under more "stress" than otherwise possible with testing. The Remote Agent example in Section 3 illustrates this approach.

JPF is a prototype, and future work at NASA Ames as well as at Stanford will consist of building more efficient model checking technology for JAVA. At this end, we study byte code verification as well as source code verification. Furthermore, work is focusing on program abstraction with the purpose of building an integrated program verification environment.

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. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*, pages 369-386, CA, November 1998. Springer-Verlag.
2. 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.
3. R. Iosif C. Demartini and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998.
4. 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.
5. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, 1986.
6. 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.
7. 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.
8. P. Godefroid. Model checking for programming languages using verisoft. In *ACM Symposium on Principles of Programming Languages*, pages 174-186, Paris, January 1997.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. A-W, 1996.
10. S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *CAV*. Springer-Verlag, June 1997.
11. J. Hatcliff, M. Dwyer, S. Laubach, and D. Schmidt. Stating static analysis using abstraction-based program specialization, 1998.