Formal Analysis of Jackrabbit Software Using Java PathFinder

Master Thesis

Ing. Jantien Sessink
Thesis Number: 580

Radboud Universiteit Nijmegen

Student:	Ing. Jantien Sessink
Supervisors:	Prof. Dr. Frits Vaandrager Drs. Ing. Pepijn Vos Prof. Dr. Ben Dankbaar
External Supervisors:	Dr. Martijn Hendriks Drs. Jurriaan Souer
Date	Nijmegen, March 17, 2008

Abstract

Software is getting more and more complicated these days and software engineers need new techniques to improve the quality of their systems. Model checking might be a good technique to assist software engineers in finding problems, but is only slowly being adopted by companies. In this research we looked at the possibility of using model checking technique on business applications to reduce the number of concurrency problems. For this we used Java PathFinder (JPF), a model checking tool that checks Java byte code.

The research consisted of two parts. In the first part we looked at the use of JPF for verifying software, in our case Jackrabbit. Although we were able to use the model checker on Jackrabbit, we did not find the documented concurrency problems we hoped to find. We were able to make some small improvements on JPF concerning the memory usage. One component was improved to make the tool less memory consuming. We were also able to estimate the expected maximum memory needed to run our models.

In the second part we looked at the perceived usefulness and ease of use of JPF. We sent out a survey to Java PathFinder users to see how useful and easy to use the tool is. Java PathFinder received a neutral assessment and three such features that would improve the tool were found.

Acknowledgement

The author would like to thank all the people who made this research project possible. Thanks for all the help and support that I got in the past few months. Some people deserve my gratitude in particular.

First I would like to thank GX for making this research possible. They provided me with all I needed to carry out this research. Special thanks go to Martijn Hendriks, who offered his knowledge when I needed it and Jurriaan Souer for his guidance. But I also wish to thank everyone else at GX for making it a fun time to work there.

I would like to thank my supervisors from the university. I wish to thank Frits Vaandrager for guiding me in the right direction, and reflecting on the results. Pepijn Vos for his help and ideas on the management part of my research. And I would like to thank Ben Dankbaar for becoming my supervisor on such short notice.

Last of all I would like to thank Michael Stauder, the love of my life, for always believing in me and being there whenever I needed him.

Contents

List of Figures

1. Introduction

This document is the master thesis report for the project 'Formal Analysis of Jackrabbit Software Using Java PathFinder'. The research carried out was my master thesis project. I am a Computer Science student at the Radboud University Nijmegen with the graduation theme 'Embedded Systems' and the track 'Management and Technology'. The project was done at GX in Nijmegen.

This master thesis describes the research done during the final period of my master track. It is meant for all the involved parties that participated or supervised the project. This document is also open to everyone who is interested.

GX is interested in using new tools, like Java PathFinder, to improve their software quality. GX is a web technology specialist that specializes in content management systems. Their core product is GX WebManager, which is developed in-house. Like every software company, GX is concerned with improving the quality of their software. For that reason they are looking at using new tools in the software development process. For a more detailed description of GX, see appendix A.

Software can be defined as follows:

> *"written programs or procedures or rules and associated documentation pertaining to the operation of a computer system and that are stored in read/write memory" [Definition from WordNet 2.0, Princeton University]*

Software usually has some input and output, otherwise it would do the same operation every time. The input can be supplied by the user, hardware, file system, etc. The output depends on the software program and can be a file, some message on the screen, etc. The output is usually successful if it is in the expected form, value, etc. It is unsuccessful if the wrong output is returned or when some error occurred. An error can have many causes, one of them being concurrency problems.

Software development is the process of creating computer applications. This includes the entire process from the design to the deployment. Most errors are found in the testing phase and after deployment, this includes concurrency problems. Companies usually try to find all the problems in the testing phase, but due to the complexity of the software, this is not always possible. Model checking could be a good addition to the testing phase.

Concurrency issues are problems that occur in multi-threaded programs. Examples of concurrency issues in software are race conditions and deadlocks. When a race condition happens, the further execution of a program cannot be predicted anymore. Mutual exclusion can prevent race conditions from happening, but can lead to deadlocks. When a deadlock occurs, the system will stop running. Both situations are unwanted in software systems, so they need to be removed. To trace concurrency issues there are tools available, such as Java PathFinder.

Model checking is the process of checking whether a given structure (in our case Java code) is consistent with a logical formula. The technique is very general and, among others, used for verifying formal systems. There are model checkers available for checking software code, for Java code that is Java PathFinder.

Java PathFinder (JPF) is a model checking tool that can be used to track concurrency issues in Java programs. It has been developed by NASA, but became open source in 2005. The tool can be downloaded and adjusted freely. Java PathFinder examines the Java byte code and gives a verdict in the output. When issues are found, these are reported in the output report. At this moment JPF is not used in GX outside of this research project. For more information on JPF, see chapter 3.

In this research we used Java PathFinder on Jackrabbit, a content repository system, developed by Apache and used by GX. A content repository system is like a database and stores information provided by other applications or users. Because it is developed by Apache, it is an open source system and can be downloaded freely.

This document is split up into two different parts, since two different subjects were addressed in our research. The first part contains all the findings of the technical research, finding problems in Jackrabbit, etc. The second part offers all the results from the survey we used to see how other users of JPF experience the tool. After both parts, a combined conclusion is given of the entire project and the further research that is needed in this field.

The second chapter gives a description of the context of this project. It describes the problem, objectives and research questions. The third chapter offers a more detailed description of Java PathFinder, the tool used in this research. The fourth chapter gives a description of Jackrabbit, the repository that was analysed. The fifth chapter describes the results and findings of our research. It contains the models that were made of Jackrabbit and the extensions that were built and added Java PathFinder.

The sixth chapter contains the context description of the survey part of this project. The seventh chapter describes the conceptual model used. Chapter eight gives the research strategy and the outcome we expected before gathering the information. The ninth chapter shows the analysis and the results of the survey outcome.

After the two different parts a combined conclusion of the entire project is given in chapter ten, which gives an overview of our findings during the project. In that period we found several topics that need further research in the future. Chapter eleven gives an overview of those.

Part 1

2. Context

This chapter describes the context of the project starts with the problem description of the research. The first section describes the reason this research is relevant. The second section gives the taken approach we took. The last section describes the problem definition, containing the research questions en objectives.

2.1 Problem Description

Software is getting more complicated these days. One of the problems with large software systems is concurrency. Concurrency problems occur when software becomes multi-threaded and two threads interfere with each other in an unwanted way. Business applications usually have multiple threads running at the same time, for example when several users are working on the same system. When these threads collide in a way they should not, concurrency problems like race conditions or deadlocks might occur.

To prevent concurrency problems from entering the software, companies use stress tests to find them. But these tests can only search a small part of the software, therefore there is a large change that a concurrency bug will slip through the tests and will not be found until the software is up and running.

The academic community has introduced formal methods, such as model checking, to find and reduce concurrency problems. Model checking is a process of checking whether a given structure is consistent with a logical formula. The technique is very general and is, among others, used for verifying formal systems. These tools have not yet been widely adopted by businesses (see section 10.3) for good reasons. Usually model checkers use formal languages to model systems instead of software code. Business applications are usually too large to be searched by model checkers; they will cause the state space to explode. That is why many business applications still contain concurrency problems, one of them being Apache Jackrabbit.

Apache Jackrabbit is a fully conforming implementation of the Content Repository for the Java Technology API (JCR) and is defined by JSR-170 [JSR-170]. A content repository is a hierarchical concurrent content store with support for structured and unstructured content, full text search, versioning, transactions, observation, and more. Typical applications that use content repositories include content management (GX's Webmanager), document management, and records management systems. [Jackrabbit]

The diagram in figure 1 explains which components of the Jackrabbit repository are used when a user of the JCR API modifies data in the content repository. This is a simple and very common operation that touches a large part of the components in Jackrabbit. Examples of other possible operations are to add, delete and search for content.

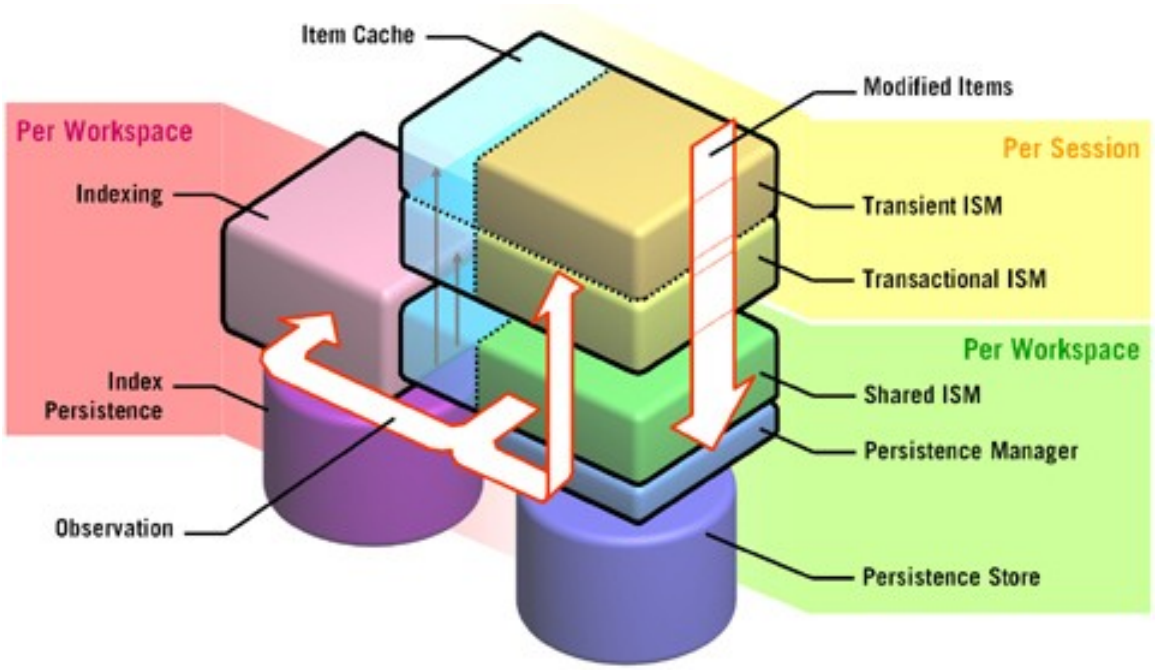

Figure 1: Jackrabbit components

An application that needs to access the repository must achieve this through a Session. There can be a large number of concurrently active Sessions, which may alter an intersecting set of repository items. Each Session therefore has a Transient Item State Manager (ISM) that manages the items locally in the Session. The local changes may be persisted by saving the Session. When this operation is invoked, the local items are pushed to the single Shared ISM which invokes the PersistenceManager (PM). The PM, in turn, takes care of storing the items in the persistence store (e.g., a database). Only the last part of the storing, the actual entering the data into the database, is done atomically, the rest of the operation is not.

The observation mechanism has the responsibility of informing all the involved components of persistent changes to the repository. The ISMs and the Search Index (SI) are among these components. For instance, when a session deletes an item which is concurrently saved in another, then the ISMs and the SI are synchronously informed of this deletion and storing.

One of the problems that recently occurred in Jackrabbit is that of deadlocking. A large number of concurrent processes are allowed to access the repository at any given time. Thus it may occur that two Sessions are trying to change the same data at the same time. Currently Jackrabbit uses locking mechanisms to synchronize the access to shared data, but this introduces unexpected deadlocks like described in issue reports JCR-447 [JCR-447] and JCR-962 [JCR-962]. It also causes a race condition that is reported in issue JCR-1148 [JCR-1148].

A deadlock occurs when there are two or more processes, each of which is blocked waiting for a resource it will never get without some drastic action being

taken [Shub03]. In order for a deadlock to occur, the following conditions, also called the Coffman conditions [Coffman71], have to be met:

1. Tasks claim exclusive control of the resources they require ("mutual exclusion" condition).
2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no pre-emption" condition).
4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

An example of a deadlock can be found in listing 1. In this example, A and B are two resources and T1 and T2 are tasks that want to acquire the resources. T1 locks A and T2 locks B, then T1 tries to get B and T2 tries to get A. Task T1 is locked because it is waiting for the unlocking of B by T2. However T2 also needs A to finish its computation and free B. This is a deadlock, also called a deadly embrace.

```
Task T1                          Task T2
        Get A                            Get B
                Get B                            Get A
                Release B                        Release A
        Release A                        Release B
End                              End
```

Listing 1: Simple deadlock example

There are two different kinds of locking mechanisms that can be used in the standard Java programming language to obtain a resource and that are important in applications[1]. These lock mechanisms are:

1. Synchronized locks
2. Read/Write locks

Synchronized locks are a simple locking mechanism in Java. Synchronized locks work a bit like semaphores. The first task to claim a resource gets it. The next task that tries to claim the resource will have to wait until the first one releases it. A first in first out (FIFO) queue is formed with waiting tasks for a resource.

Read/Write locks only allow a single thread at a time (a *writer* thread) to modify the shared data, but allow a large number of threads to concurrently read the data (hence *reader* threads). Read/Write locks are introduced in Java 5 in the package java.util.concurrent and are described in JSR-166 [JSR-166]. Listing 2 shows a Java example of a synchronized lock with a deadlock.

```
public class Deadlock {

  public static void main(String[] args) {
```

[1] Only synchronized locks were investigated in Jackrabbit.

```
    final Object lock1 = new Object();
    final Object lock2 = new Object();

    new Thread() {
      public void run() {
        synchronized(lock1) {
          try {
            sleep(10);
          } catch (Exception e) { e.printStackTrace(); }
          synchronized (lock2) {
            try {
              sleep(10);
            } catch (Exception e) { e.printStackTrace(); }
          }
        }
      }
    }.start();

    synchronized(lock2) {
      try {
        sleep(10);
      } catch (Exception e) { e.printStackTrace(); }
      synchronized (lock1) {
        try{
          sleep(10);
        } catch (Exception e) { e.printStackTrace(); }
      }
    }
  }
}
```

Listing 2: A simple deadlock example in Java

To solve the deadlock problem in Jackrabbit as stated this section, the locking mechanism, in the case of JCR-447, is temporarily disabled. But this may lead to another problem, data corruption due to concurrent access to a data structure. When the data gets corrupted, as in JCR-1148, it is unsure what the saved data at the end of the program will be.

The current technique used to find these kind of problems is stress testing. Many threads are started and do the same instructions many times. They hope this will reveal the concurrency bugs. But the problem with this kind of testing is that it is not reliable, not every execution path is explored and bugs may slip through. Another problem is that if a bug is found, it is usually not reproducible. The stress test accidentally stepped on the bug, the next time it may not find the same bug.

2.2 Approach

Jackrabbit was analysed using formal methods to see whether it was possible to use formal methods for this purpose. There were different kinds of models and tools available to us that could have been used for analysing Jackrabbit, for example Uppaal, Java PathFinder, Spin, Bandera, NuSMV, etc.

Formal methods are mathematically based methods that are used for specification, development and verification of hardware and software systems. In this research we tried to verify Jackrabbit, therefore there were two main kinds of techniques available. The first was theorem proving and the second was model

checking. Because we tried to use formal methods on a business application, it would have been best to use it on the actual software code. Therefore we chose to use model checking. To be more specific we used Java PathFinder, because it is a model checker that works directly on the Java byte code

Java PathFinder (JPF) is a system to verify executable Java byte code programs. In its basic form, it is a Java Virtual Machine (JVM) that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. Unlike traditional debuggers, JPF reports the entire execution path that leads to a defect. JPF is especially well-suited for finding hard-to-test concurrency defects in multithreaded programs. JPF was originally developed by NASA, but became open source in 2005. [JPF]

The advantage of JPF is that one does not have to analyse the source code themselves, the tool will do that. One also does not have to model the JVM, because this is already enclosed in JPF. A disadvantage of JPF is that it is not designed to scale well for large programs (~10.000 lines of code depending on the internal structure [JPF]). This means that a great deal of abstraction was needed to be able to use JPF. A part of this abstraction was already implemented in the model checker; another part was made in the model. This was done by building test case scenarios in which only a part of Jackrabbit was tested. JPF was able to follow the path and check the calls that were made to Jackrabbit. This way, Java PathFinder was able to check only a part of Jackrabbit at the time. The test case scenarios had to be well designed to ensure that as much of the repository as possible was verified. For this we chose some examples from the Jackrabbit issue tracking system that had concurrency issues in them.

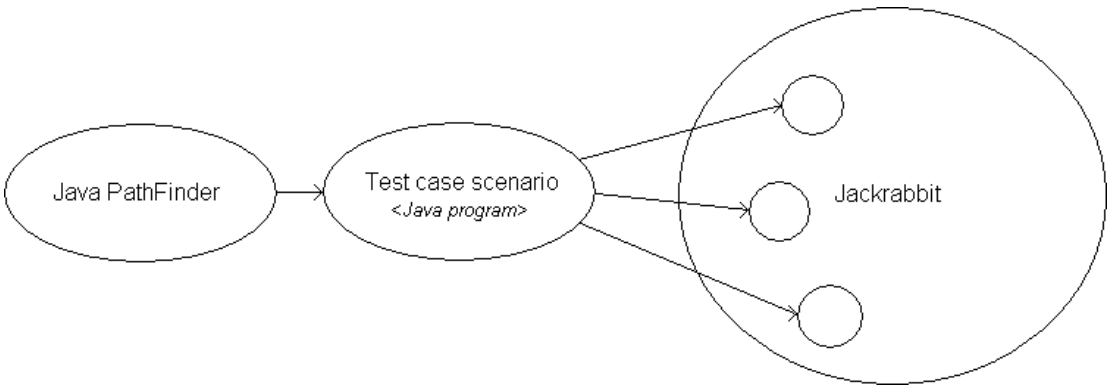

Figure 2: Test case scenarios to test Jackrabbit

It is important to notice that not the entire Jackrabbit code was verified, because it was not possible to build test cases that covered all of the code. Therefore we only tested critical parts that contained problems. And even if the entire code of Jackrabbit would have been tested, that would not have meant that the program is completely bug-free. JPF can only find concurrency bugs like deadlocks and race conditions, but there are more kinds of problems that could still be in the software. For example security issues, code does not correspond with the functional demands, graphical errors and other bugs.

2.3 Problem Definition

2.3.1 Objectives

The main objective of the research was to evaluate the usefulness of model checkers for the analysis of real production code, such as Apache Jackrabbit. We have tried to find out whether it is feasible to use model checkers by software engineers on Java applications to verify the software. We tried to build use cases of the Jackrabbit API and verify these.

Another objective of the research was to improve the quality of Jackrabbit by trying to reduce the chances for deadlocks. With the models that were created, we tried to find problem areas in the Jackrabbit source code.

2.3.2 Research Questions

There was one main research questions that was supported by sub questions.

Is it feasible to use model checking technology to analyse a real-life software system, for example Jackrabbit?

- *(a) How useful are model checkers for analysing real-life software systems?*
- *(b) Is it possible to use Java PathFinder in the build-lifecycle of a Java program, for example in the build of Apache Jackrabbit?*
- *(c) Is it possible to find the known deadlocks in Jackrabbit with model checkers?*

2.3.3 Relevance

As can be seen in appendix A, GX main product is GX Webmanager. This product leans heavily on Apaches Jackrabbit for its content repository system. When Jackrabbit enters a deadlock state, it would affect the execution of Webmanager. This made the research relevant for GX, because by improving the quality of Jackrabbit, the quality of Webmanager could have been improved.

The research was also relevant for both the Apache community as the formal methods community. If it was possible to use formal methods to identify deadlock problems, then the Apache community might have been able to solve them. The formal methods community was interested in the possibility of using model checkers on real-life software programs.

2.3.4 Products

At the end of this research project, these products were delivered:

- **Master Thesis**
  A written report describing the research period with at least a summary, the problem definitions, used methods/models, fundamentals and the conclusions.
- **Presentation**
  During the presentation all interested parties are informed of the results and course of the research.
- **Models of Jackrabbit**
  Models of Jackrabbit were made to answer the research questions. In case of JPF, the models are in the Java code of the test programs.

3. Java PathFinder

This section gives a description of Java PathFinder. Most of the information below comes from the Java PathFinder website [JPF].

3.1 What is Java PathFinder

Java PathFinder (JPF) is a tool to verify executable Java byte code. JPF can be interpreted as a Java Virtual Machine (JVM) that is used as an explicit state software model checker. It explores systematically all possible execution paths to find problems like unhandled exceptions, deadlocks and race conditions. The power of JPF lies in finding concurrency defects in multithreaded Java programs.

JPF is a model checker that runs directly on the Java byte code. This means that instead of running the Java program once, it will execute all possible execution paths. When a property is violated along one of the execution paths, JPF will report this and the entire execution that leads to the defect. JPF keeps track of every step how it got to the defect.

3.2 History

Java PathFinder was developed at NASA, at the NASA Ames Research Center. The JPF program started in 1999 as a Java-to-Promela translator so Java programs could be translated to Promela and then checked with the Spin model checker [Spin]. But this introduced language coverage problems, because not every feature of Java could be translated to Promela.

The second stage of JPF was to become a stand-alone model checker that could directly process Java byte code. This happened in 2000, JPF became a Java Virtual Machine to check Java byte code. In 2005 JPF became an open source project and is hosted by SourceForge.net. In 2006 JPF4 was introduced which was the version used in this research project. From now on, when talked about JPF, Java PathFinder version 4 is meant (the svn version).

3.3 JPF Architecture

The JPF architecture is designed in such a way that it is easily expendable with other search strategies, properties, etc. Figure 3 shows the JPF architecture. A Java byte code program is fed to JPF; JPF searches through the program for property violations and reports them in the verification report.

JPF is a flexible system; the pluggable objects that can be seen in figure 3 can be set in the configuration files. This file, named default.property, contains all the options for JPF, which search class to use, which backtracker, properties, etc.

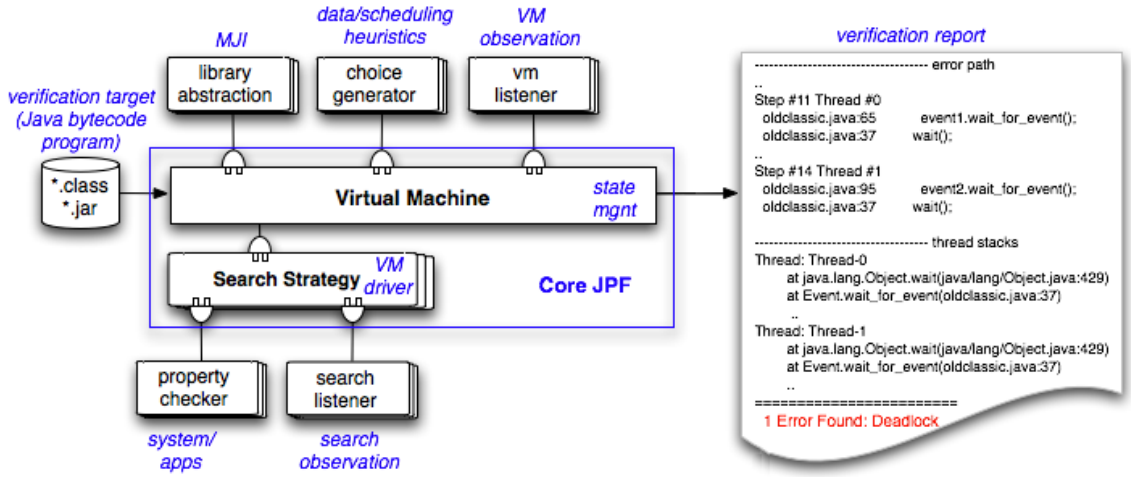

Figure 3: JPF architecture

The two main objects of JPF are the JVM and the Search object. The JVM is a Java specific state generator. By executing Java byte code instructions, the JVM generates state representations. The main JVM parameterizations are classes that implement the state management (matching, storing, backtracking).

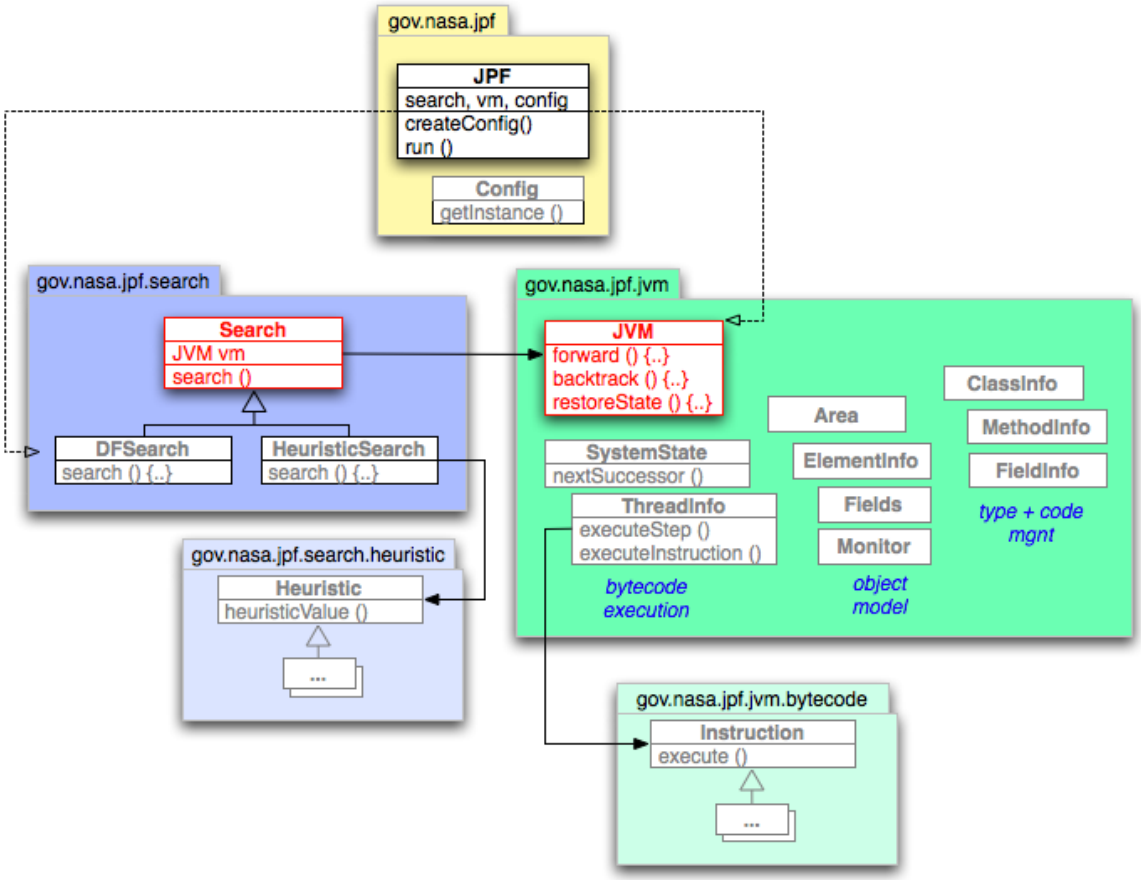

Figure 4: JPF class architecture

Most of the execution scheme is delegated to the SystemState, which in turn uses a SchedulerFactory (a factory object for ThreadChoiceGenerators) to generate scheduling sequences of interest. Figure 4 shows what the JVM looks like.

The Search object is responsible for selecting the state from which the JVM should proceed, either by directing the JVM to generate the next state (forward), or by telling it to backtrack to a previously generated one. Search objects can be thought of as drivers for JVM objects.

Search objects also configure and evaluate property objects (e.g. NotDeadlockedProperty, NoUncaughtExceptionsProperty). The main Search implementations include a simple depth-first search (DFSearch). A Search implementation mainly provides a single search method, which includes the main loop that iterates through the relevant state space until it has been completely explored, or the search found a property violation.

3.4 Partial Order Reduction

Partial Order Reduction (POR) is a technique to reduce the size of the state space to be searched by a model-checking algorithm. The main principle or partial order reduction is to find a subset of the enabled transitions $ample(s) \subseteq enables(s)$ that are used to generate the successor of a state s. By choosing the subset of enabled transitions carefully, the correctness of the checked property (or the existence of a counterexample) is preserved between the full state space and the reduced one [Peled97]. A property holds in the reduced state space iff it holds in the full state space.

The number of different scheduling combinations can increase rapidly when dealing with concurrent programs, causing the state-space to explode. But not all possible instruction interleavings for all treads are necessary to explore, since some instructions are not relevant and do not have effect outside the thread itself. By putting these instructions into one transition, the number of scheduling states can be significantly reduced.

JPF uses on-the-fly partial order reduction, which means that JPF determines at runtime which instructions are relevant for scheduling. This is done when the next byte-code instruction is evaluated. When this instruction is relevant for scheduling, a new thread-choice-generator is created and the instructions are scheduled for re-execution. When an instruction is not relevant, the next instruction is evaluated by the JVM. The JVM executes all instructions in the current thread until one of the following conditions is met:

1. The next instruction is scheduling relevant (mostly the get and set instructions)
2. The next instruction yields a non-deterministic result (i.e. simulates random value data acquisition).

3.5 Choice Generators

Usually model checking explores every path and state in the state-space, but not very state is relevant. JPF only stores states that could affect multiple threads. Choice Generators (CG's) in JPF are created when there is a choice to be made. This can be either a data choice or a scheduling choice. In this research, only the scheduling choices have been used, so in this section we will only elaborate on these.

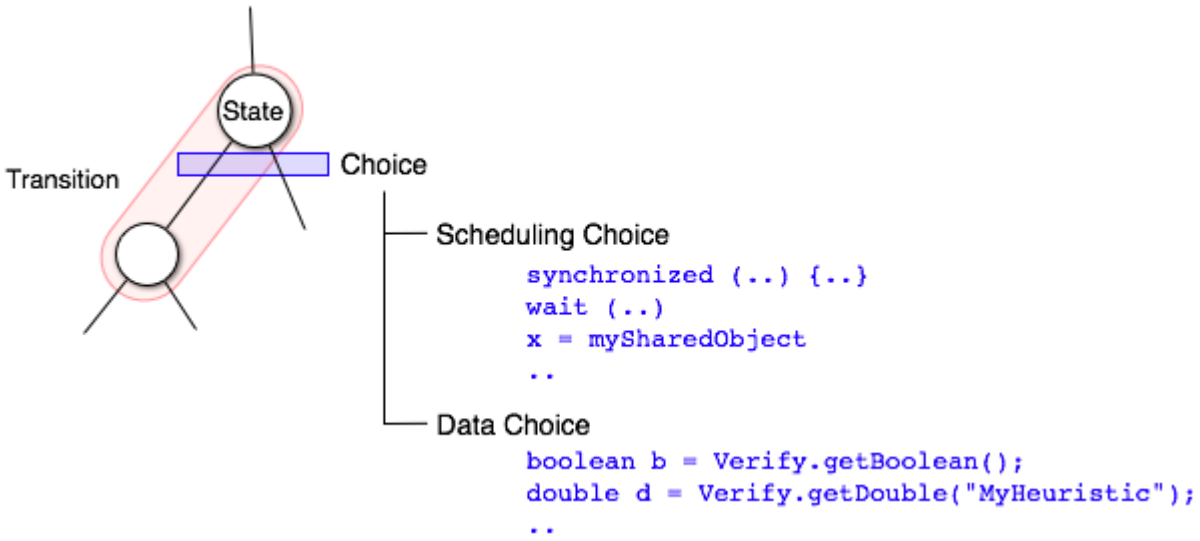

Figure 5: States, Choices and Transitions

Figure 5 shows what choices, states and transitions there are in JPF. A state is a snapshot of the current execution status of the application, plus the execution history that leads to that state. The three most import components of a state are the Kernel State (threads, heap), Tail (execution history) and current and next CG's (the objects encapsulating the choice enumeration that produces different transitions). A transition is a sequence of instructions that leads from one state to the next. There can be multiple transitions that lead out of a state. A choice starts a new transition.

When a choice is generated, JPF stores the current state in the backtracker. Then it takes the first choice and executes the transaction that belongs with that choice. If there are no more transitions going out of a state, this state is an end-state. When JPF reaches an end-state, it will backtrack with the backtracker to the state that came before the current one. It will then take the next transition of that state and go on searching.

3.6 Exploring the State-Space

The main function of JPF is to explore the state-space, searching for property violations. To explore the state-space of a Java program, JPF uses a search strategy. There are multiple search strategies implemented in JPF, such as a depth first search and a breath first search. In either case, JPF uses a waiting-list and a past-list to know which states still have to be explored and which already have been.

All the states that still need to be searched (waiting list) will be stored in the backtracker. The backtracker stores the entire kernel state of the JVM so JPF can load the data again and continue from that point on. The past list in JPF is implemented in the state set. The state set calculates a hash code for that state and stores it. With this set, JPF can see if a state has already been visited. This does mean that JPF is inherent incomplete, since the passed set is hashed. It is possible that two states produce the same hash code and JPF will think that the state has already been visited. When JPF thinks this state is already visited while it has not been, it will not continue exploring that state. Therefore it is possible that at some point, JPF misses states. This can be avoided by not using a state set to store the pass list. But not using a state set can only be done when the state-space is acyclical. When a state-space is cyclical, JPF will not stop searching because it cannot detect whether a state has already been visited and it may continue in an endless loop. When a state-space is acyclicalit is not necessary to store the pass list [Behrmann03].

Listing 3 shows a standard depth first search algorithm [Biggs03]. A depth first search (DFS) algorithm is a way of traversing or searching a tree or graph. DFS is an search strategy that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation (like in JPF and in the example of listing 3), all freshly expanded nodes are added to a last in first out (LIFO) stack for exploration.

One of the search algorithms in JPF is also a depth first search algorithm, shown in listing 4. This listing is stripped of all the Java code that is not part of the depth first search algorithm. Both listings are similar to each other. Listing 3 starts initialising the stack with only the initial state in line 1. In JPF the stack is saved in the backtracker class, which is initially empty. The first state is entered to the backtracker when the `forward()` function is called for the first time. At that point the JVM takes his first step and stores that state.

Then the standard algorithm starts its loop in line 2, it will continue this until the stack is empty. The same loop in JPF can be found in line 3 of listing 4, there the program loops until the variable `done` is set, which is only set when there are no more states left. The JPF algorithm can stop on two more places, when it is not possible to backtrack anymore (line 6) or when the current state satisfies one of the properties searched for (line 12). The lines 5 and 6 of listing 3 check if there is a new state and go to that state, in JPF this is done in line 10, the `forward()` function. Line 7 of listing 3 goes one state back, JPF does this in line 5, the `backtrack()` function.

The most important part of the search algorithm is done in the `forward()` function. This function saves the current state in the backtracker class and calculates the next state from the byte code. The `backtrack()` function only pops the last saved state in the backtracker class and goes one state back.

```
1  let stack = {v};
2  while stack is not empty do
3    begin
4      x := top(stack)
5      if x is adjacent to a new vertex y
6        then add y at the top of the stack
7        else remove x from the stack
8    end
```

Listing 3: Depth first search algorithm

```
1  public void search () {
2    depth = 0;
3    while (!done) {
       // check if backtracking is needed, this is the case
       // when the current state has already been visited or
       // it has no successors.
4      if ( !isNewState || isEndState) {
         // go back to the previous state on the stack
5        if (!backtrack()) {
           // backtrack not possible, terminate search
6          break;
7        }
8        depth--;
9      }
       // save current state and step forward to the next state
10     if (forward()) {
         // check if this state satisfies a given property,
         // when satisfied, terminate search.
11       if (hasPropertyTermination()) {
12         break;
13       }
14       depth++;
15       }
16     }
17   }
18 }
```

Listing 4: Depth first search algorithm in JPF

3.7 Running JPF

JPF can be run in two different ways. One is by adding JPF in the source code of your software, the other is by running JPF from the command line. The easiest way is by running JPF from the command line, because then nothing has to be changed in the source code of one's program.

Before running JPF, it is important to have the right configuration. JPF is very flexible because it uses dynamic class loading to load part of its kernel. Which classes have to be loaded can be set in a configuration file named *default.properties*. Examples of classes that are configurable are the JVM, backtracker, state set, search algorithm, and many more. Besides the classes that have to be loaded, some variables can be set like the maximum depth of the search, whether to search for multiple errors, which properties to check for, etc.

After running JPF, a report is shown with the results of the run. This report contains the errors that were found, with a snapshot of the threads. It also contains a statistical report that shows how many states were passed, how much memory was used, how many states where backtracked, etc. An example run of JPF is shown

below. Listing 5 shows the output of JPF when given the input of listing 2 from chapter 2.

```
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center


====================================================== system under test
application: nl.gx.firsthop.Deadlock.class

====================================================== search started: 12/5/07
11:03 AM

====================================================== error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
  thread
index=0,name=main,status=TERMINATED,this=null,target=null,priority=5,lockCount=0
  thread index=1,name=Thread-
0,status=BLOCKED,this=nl.gx.firsthop.Deadlock$1@225,priority=5,lockCount=0
  thread index=2,name=Thread-
1,status=BLOCKED,this=nl.gx.firsthop.Deadlock$2@127,priority=5,lockCount=0


====================================================== snapshot #1
thread index=1,name=Thread-
0,status=BLOCKED,this=nl.gx.firsthop.Deadlock$1@225,priority=5,lockCount=0
  owned locks:java.lang.Object@220
  blocked on: java.lang.Object@221
  call stack:
        at nl.gx.firsthop.Deadlock$1.run(Deadlock.java:21)

thread index=2,name=Thread-
1,status=BLOCKED,this=nl.gx.firsthop.Deadlock$2@127,priority=5,lockCount=0
  owned locks:java.lang.Object@221
  blocked on: java.lang.Object@220
  call stack:
        at nl.gx.firsthop.Deadlock$2.run(Deadlock.java:39)


====================================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:   thread
index=0,name=main,s..."

====================================================== statistics
elapsed time:       0:00:01
states:             new=46, visited=0, backtracked=10, end=7
search:             maxDepth=35, constraints=0
choice generators:  thread=39, data=0
heap:               gc=49, new=244, free=159
instructions:       2938
max memory:         4MB
loaded code:        classes=54, methods=867

====================================================== search finished: 12/5/07
11:03 AM
```

Listing 5: An example of a JPF run with code from listing 2

4. Jackrabbit

Jackrabbit is the Java content repository implementation of the Apache Software Foundation [Apache]. It offers a graph-based architecture consisting of nodes and properties to store content. This chapter first gives a description of the Java Content Repository API and then a more detailed description of Jackrabbit.

4.1 Java Content Repository API

The amount of content management applications is growing rapidly (CMS Matrix reports 875 systems[2] [CMS]), which calls for a common, standardized API for content repositories. The Java Content Repository API (JCR-170) provides such an interface. The goal of a content repository API is to abstract the details of application data storage and retrieval such that many different applications can use the same interface, for multiple purposes, without significant performance degradation. Content services can then be layered on top of that abstraction to enable software reuse and reduce application development time. [Fielding05]

The Repository model is a simple hierarchy and looks much like an n-ary tree. It consists of a single content repository, with one or more workspaces. Each workspace contains a tree of items; an item can be either a node or a property. A node can have zero or more children, and zero or more associated properties, where the actual content is stored. Figure 6 shows an example of a repository hierarchy. Circles represent nodes, while rectangles represent properties. Of interest are nodes A, B, and C, descending from the singular root node. Node A has two properties: a string, "John," and an integer, 22. [Barik06]

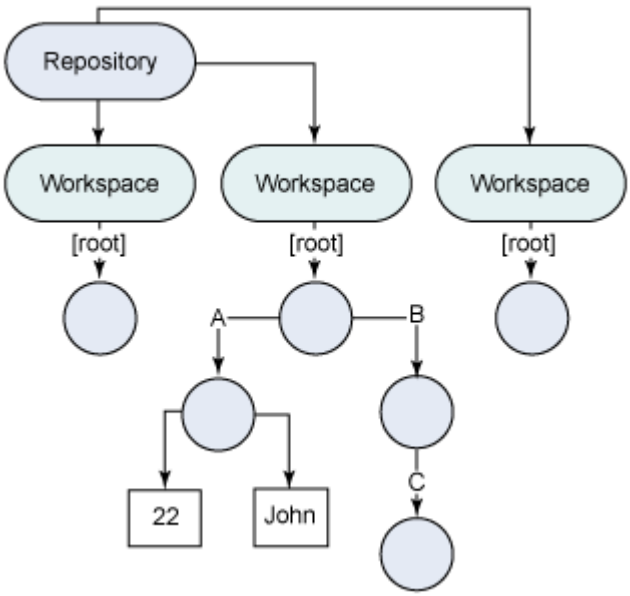

Figure 6: A repository model with multiple workspaces

[2] Number of registered content management systems at www.cmsmatrix.org at March 13, 2008

There are different features and operations that should be supported by a JSR-170 compliant repository. To simplify transfer from a non-JSR-170 compliant repository to a JSR-170 compliant repository, the functionalities are divided into two levels. Vendors should first implement the first level functionalities before the second level functionalities. Vendors do not have to implement the second level, but then they are not fully in compliance with JCR-170. There are also some advanced options that the vendor can choose to implement or not. The following list describes these levels [Patil06], figure 7 gives a visual representation of all functionalities in the different levels [Jackrabbit]:

- Level 1 (read-only repository):
  This level includes basic functionalities for the reading of the repository, search options and export to XML.
- Level 2 (writable repository):
  A level 2 repository is a superset of level 1. In addition to level 1's functionalities, it defines writing and importing content into the repository.
- Advanced options:
  The extra options include versioning, (JTA) transactions, query using SQL, explicit locking and content observation. In addition to being either level 1 or level 2 compliant, any repository can decide to implement one or more of these functional blocks

Figure 7: JCR-170 functionalities

JCR-170 interface only defines what is needed to communicate between systems. It does not define how vendors should implement these options. It is only an interface, not a design architecture.

4.2 Jackrabbit Example

Jackrabbit is an implementation of the JCR-170 that supports all the functionalities that are mentioned in the previous sections. This section gives an example of basic repository call to show how Jackrabbit works. The example below is taken from the Jackrabbit website [Jackrabbit].

```
1  import javax.jcr.Repository;
2  import javax.jcr.Session;
3  import javax.jcr.SimpleCredentials;
4  import javax.jcr.Node;
5  import org.apache.jackrabbit.core.TransientRepository;
6
7  /**
8    * Second hop example. Stores, retrieves, and removes example
9      content.
10 */
11 public class SecondHop {
12
13   /**
14     * The main entry point of the example application.
15     *
16     * @param args command line arguments (ignored)
17     * @throws Exception if an error occurs
18     */
19   public static void main(String[] args) throws Exception {
20     Repository repository = new TransientRepository();
21     Session session = repository.login(
22       new SimpleCredentials("user", "password".toCharArray()));
23     try {
24       Node root = session.getRootNode();
25
26       // Store content
27       Node hello = root.addNode("hello");
28       Node world = hello.addNode("world");
29       world.setProperty("message", "Hello, World!");
30       session.save();
31
32       // Retrieve content
33       Node node = root.getNode("hello/world");
34       System.out.println(node.getPath());
35       System.out.println(node.getProperty("message").getString());
36
37       // Remove content
38       root.getNode("hello").remove();
39       session.save();
40     } finally {
41       session.logout();
42     }
43   }
44 }
```

Listing 6: Second hop example from the Jackrabbit website

The following output is created when running the example:

```
/hello/world
Hello, World!
```

The first few lines of the code (lines 1 till 5) import the packages needed for running this code. The first four packages are from the javax.jcr package which contains the JCR API. By only using this API, the application remains independent from the underlying repository. The fifth package that is imported is the jackrabbit repository package. Normally it is good practice to keep the application code free from direct Jackrabbit dependencies, but since this is a simple example program, this shortcut is used.

In line 20 the program starts with creating a repository. In this example a default repository is created, but it is also possible to create a repository with a different configuration file and home directory. An example of that can be seen in the models that where build in chapter 5.

Line 21 starts a new session to the repository using the default workspace and simple credentials. When no credentials are used, the session is a read-only session. Jackrabbit accepts any username and password as valid credentials, so a SimpleCredentials class is used. The SimpleCredentials constructor follows the JAAS (Java Authentication and Authorization Service) convention of representing the username as a normal String, but the password as a character array, so we need to use the String.toCharArray() method to satisfy the constructor.

Line 24 gets the root node from the repository, this is the start of the tree. In line 27 till 30, new content is saved in the repository. The first 2 lines add two nodes, line 27 add a node called *hello* to the root node and line 28 adds a node called *world* to the *hello* node. Then line 29 adds a property to the world node containing the string message “Hello, World!”. In line 30 all the changes to the session are being saved in the transient storage. The data tree of this example is shown in figure 8.

After the content is added to the repository, it can be read by different sessions. Lines 33 till 35 show how data can be read. Line 33 gets the node ‘*hello/world*’, that is the node called *world* that is located under the node called *hello*. Line 34 prints the first line to the output, which is the path of that node (in this case ‘*hello/world*’). In line 35 the string that is saved in the property *message* is printed (in this case ‘Hello, World!’).

Line 38 removes the node hello. This means that the node world is also gone, since the remove call removes the node and its entire subtree. This is first done only locally (in the session) and when the session is saved in line 39, it is removed from the repository.

Line 41 logs the session out from the repository. It is good practice to do so, to make sure the repository is properly closed. When the last session is logged out (which is the case here, since we only have one session) the repository is automatically shut down.

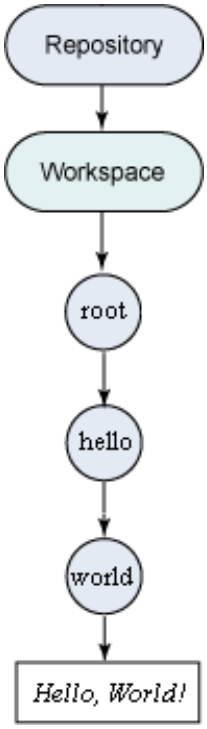

Figure 8: Hello world example tree

4.3 Issues in Jackrabbit

In this research we used some issues that were reported in the issue tracking database of Jackrabbit. When one knows that there might be a problem at some place in the repository, it is easier to search for it.

4.3.1 JCR-447

This issue is a deadlock on a concurrent commit of transactions to the repository. The lock happens when two different sessions both enter the shared item state manager. This is fixed in the current version of Jackrabbit by disabling the write-locking mechanism in the version specific item state manager when a transaction is being committed. This does fix the deadlock problem, but by disabling the locking mechanism, it may occur that two sessions write data at the same time which makes the data inconsistent.

4.3.2 JCR-962

The JCR-962 issue concerns a deadlock in the concurrent committing of transactions that use versioning. It is similar to the issue of JCR-447, but with the difference that these transactions use versioning. The deadlock happens when one session is saving a session and the other is committing a transaction. This issue has been fixed in the current version of Jackrabbit.

4.3.3 JCR-1148

This issue is a NullPointerException that occurs when multiple threads are concurrently adding, retrieving and removing nodes from the repository. At some point a NullPointerException is thrown from the item state class while saving the session.

5. Models and Results

This chapter describes the models that were built to trigger the issues in Jackrabbit, the problems encountered when running these models, the extensions that were made to JPF to solve these problems, and the final results.

5.1 Changes to Jackrabbit

In order to let JPF accept Jackrabbit, some changed had to be made to the repository. There are two main functionalities that are not implemented in JPF and that Jackrabbit needs. These functionalities are File I/O and dynamic class loading [Visser03].

Jackrabbit uses several XML files to get the configuration settings from the user. These configuration files contain information about what kind of file system to use, security settings, workspace settings, versioning, etc. During the configuration of the repository these files are parsed and the information is set into variables. Some of the values that are loaded are classes that have to be used by Jackrabbit.

To load the classes that are configured in the configuration files, Jackrabbit uses dynamic class loading. Examples of the classes that can be configured are what kind of persistence manager to use, what backtracker, state set, etc. There are different persistence manager classes available in Jackrabbit, such as an ImMemPersistenceManager, DerbyPersistenceManager, DatabasePersistenceManager, etc. The classes to use are set dynamically during the configuration of the repository. The same goes for all the other configurable classes.

In order to avoid XML parsing and dynamic class loading, the source code of Jackrabbit had to be adjusted. It is very well possible to program around the XML parsing by putting the configuration information directly in the source code. Although by doing so the repository is not configurable anymore, but it is the only way to make JPF accept Jackrabbit. We removed the code that loads a class, which was set in the configuration files, and replaced it by a normal way of creating a new instance of a class. In total 18 files were adjusted.

It took some time approximately two weeks to program around the XML parsing and dynamic class loading, but after that was done, Jackrabbit was accepted by JPF and no more programming was needed. Our approach may become a bit problematic when one wants to use JPF in the build lifecycle of a software system.

In February of this year, a patch was added in the issue tracking system [JCR-1412] of Jackrabbit to add the possibility of dynamically configure the repository. This means that no configuration file is needed any more, but the repository can now be configured in the source code of the program that calls it. This functionality was added so JPF could be tested with different configurations without having to create

hundreds of configuration files. In the future, JPF could also use this feature of Jackrabbit

All the models in the next sections were run with the same configuration of JPF. We used JantiensBacktracker2 (see section 5.7.2) as a backtracker. The depth first search algorithm was used which is standard in JPF. In the runs, we did not use a state set to store all the states that were passed. All our models are acyclical and therefore need no pass set [Behrmann03].

5.2 Model 1

The first model represents a basic Jackrabbit operation (for source code see listing 7). The model starts as a single thread. It starts up two different sessions to the same repository and adds the same node in both sessions. Then it starts up a new thread and saves both sessions simultaneously.

```
public class Model1 extends Model {

  public static void main(String[] args) throws Exception {
    Session session = getSession("jantien", "password");
    final Session session2 = getSession("jantien2", "password2");

    try {
      Node root = session.getRootNode();
      Node session1Node = root.addNode("session1");
      root.addNode("session2");
      session.save();

      Node root2 = session2.getRootNode();
      Node session2Node = root2.getNode("session2");

      session1Node.setProperty("prop", "value");
      session2Node.setProperty("prop", "value");

      new Thread() {
        public void run() {
          try {
            session2.save();
          } catch (RepositoryException e) {
            throw new RuntimeException(e);
          }
        }
      }.start();

      session.save();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 7: Model 1

The reason that only the save operation is done multi-threaded is because that is where the two threads come together. Adding a node is done within each session, so the sessions do not affect each other. When a save is done, the session

sends its changes to the repository to the shared item state manager, which starts the storing in the common repository. Since this is the only action that may affect the other session, this is done in multiple threads. The more instructions are done multi-threaded, the more rapidly the state-space grows. So it is important to reduce the amount of multi-threaded instructions as much as possible.

When running this model[3], JPF does not find any problems when searching for uncaught exceptions, deadlocks and race-conditions. This is expected, since this model represents a basic Jackrabbit interaction and not a known error. It takes JPF 3½ hours to complete, using 226 MB of memory and searching 27,433 different states.

5.3 Model 2

Model 2 tried to find the issue JCR-447 (for source code see listing 8). This model also starts as a single thread to create two sessions and fill them both with the same node using versioning and Java transactions (JTA). Then the model starts a new thread and commits the sessions simultaneously.

```
public class Model2 extends Model {

  public static void main(String[] args) throws Exception {
    XASession session1 = (XASession) getSession("jantien",
"password");
    XAResource xares1 = session1.getXAResource();

    final XASession session2 = (XASession) getSession("jantien",
"password");
    final XAResource xares2 = session2.getXAResource();

    Xid xid1 = getXid();
    final Xid xid2 = getXid();

    try{
      Node root1 = session1.getRootNode();
      Node session1Node = root1.addNode("session1");
      session1Node.addMixin("mix:versionable");
      session1.save();

      Node root2 = session2.getRootNode();
      Node session2Node = root2.addNode("session2");
      session2Node.addMixin("mix:versionable");
      session2.save();

      xares1.start(xid1, XAResource.TMNOFLAGS);
      session1Node.checkout();
      session1Node.setProperty("bla1", "1");
      session1Node.save();
      session1Node.checkin();
      xares1.end(xid1, XAResource.TMSUCCESS);
```

[3] All models were run on a Linux server, with 2 GB memory, and 2 Intel Pentium 4 (3.4 GHz) processors. JPF was configured with JantiensBacktracker2 (see section 5.7.1), no pass list (state space is acyclical) and the depth first search algorithm.

```
      xares2.start(xid2, XAResource.TMNOFLAGS);
      session2Node.checkout();
      session2Node.setProperty("bla1", "1");
      session2Node.save();
      session2Node.checkin();
      xares2.end(xid2, XAResource.TMSUCCESS);

      new Thread() {
        public void run() {
          try {
            xares2.commit(xid2, true);
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }.start();

      xares1.commit(xid1, true);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 8: Model 2

We expected this model to find the bug described in JCR-447, because, as reported, the deadlock occurs in the version specific item state manager during the committing of a transaction. A transaction can be defined as an indivisible unit of work comprised of several operations, all or none of which must be performed in order to preserve data integrity [Mahapartra00]. JCR-447 has been fixed with a no-lock-hack which temporarily disables the locking mechanism that causes the deadlock. This no-lock-hack has been removed before running this model, otherwise the deadlock could never be found.

In this model only the commit statements are executed multi-threaded to reduce the state-space. But it did not reduce the state-space enough, because after ca. 30 minutes JPF[3] ran out of memory (it used a maximum of 1722 MB) after going through 24,777 different states. Since the state-space is much deeper, the program ran out of memory. In that time, it did not find the deadlock that should have been in this model. We tried altering the model so that it would find the deadlock, but with no success.

5.4 Model 3

Model 3 tried to find the issue JCR-962 (for source code see listing 9). This model starts by creating two sessions in a single thread. First it adds a node, and then it makes that node versionable and saves it. After that it starts a new thread and creates a user transaction, checks the node in and out and then commits the transaction. The model does this in two threads at the same time.

```
public class Model3 extends Model{

  public static void main(String[] args) throws Exception {
```

```
    Session session1 = getSession("jantien", "password");
    final Session session2 = getSession("jantien", "password");

    Node node1 =
session1.getRootNode().addNode(Thread.currentThread().getName());
    node1.addMixin("mix:versionable");
    session1.save();

    final Node node2 =
session2.getRootNode().addNode(Thread.currentThread().getName());
    node2.addMixin("mix:versionable");
    session2.save();

    new Thread() {
      public void run(){
        try {
          UserTransaction utx = new UserTransactionImpl(session2);
          utx.begin();
          node2.checkout();
          node2.checkin();
          utx.commit();
        } catch (Exception e) {
          System.out.println(Thread.currentThread().getName() + ":");
          e.printStackTrace();
        }
      }
    }.start();

    try {
      UserTransaction utx = new UserTransactionImpl(session1);
      utx.begin();
      node1.checkout();
      node1.checkin();
      utx.commit();
    } catch (Exception e) {
      System.out.println(Thread.currentThread().getName() + ":");
      e.printStackTrace();
    }
  }
}
```

Listing 9: Model 3

Model 3 should have found the bug from JCR-962 that occurs in the commit of a transaction that uses versioning. This model checks this by committing two transactions from two different threads, both using versioning. It is derived from the example code from the website that triggers the deadlock. The original example contains 100 threads that try to do a commit, 100 times in every thread. This procedure is used because JUnit only runs one execution path of the program. By using a lot of threads doing something many times, they hope to reveal the concurrency bugs. This is simplified in the model because JPF explores every execution path. More than one thread would make the state-space explode exponentially. This model reduces the amount of threads to two, doing the operations once. Since this is run with a model checker that checks every state execution, it should find the deadlock even with fewer threads. The code that fixes this issue has been removed before running the model.

When running this model[3], no deadlocks have been found. After 24 minutes JPF ran out of memory (1722 MB), in that time it searched 29,927 states. At that point, it has not reached the maximum depth of the state-space yet. This can be concluded from two facts, one being that no states were backtracked, which happens after having reached the deepest point. And two, because in section 5.6 a test-run is done reaching at least a depth of 96,606 states.

5.5 Model 4

Model 4 tried to find the issue JCR-1148 (for source code see listing 10). This model first creates two sessions to the repository. Then it adds a node to one of the sessions. After that is starts a new thread and in one thread it removes the node that was just added and in the other thread it adds that same node again.

When we ran this model, we expected a null pointer exception in the ItemState when two threads come together. No code had to be removed from the model, since this is still an open issue and has not been fixed yet.

When running Model 4[3], there were no null pointer exceptions found. This model did finish in approximately 24 hours after having searched 78,905 new states.

```
public class Model4 extends Model {

  private static final int MAX_IT = 1;

  static void main(String[] args) throws Exception {
    Session session1 = getSession("jantien", "password");
    final Session session2 = getSession("jantien", "password");

    try {
      Node root2 = session2.getRootNode();
      final Node session2Node = root2.addNode("session2");

      Node root1 = session1.getRootNode();
      Node session1Node = root1.addNode("session1");

      for (int i = 0; i < MAX_IT; i++){
        session2Node.addNode("session2_" + i);
        session2.save();
      }

      new Thread() {
        public void run() {
          try {
            for (int i = 0; i < MAX_IT; i++){
              session2Node.getNode("session2_" + i).remove();
              session2.save();
            }
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }.start();

      for (int i = 0; i < MAX_IT; i++){
```

```
            session1Node.addNode("session2_" + i);
            session1.save();
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
```

Listing10: Model 4

5.6 JPF and Memory Usage

Like for many model checkers, one of the main problems of JPF is its memory usage. When the state space increases, the amount of memory needed increases rapidly. The main cause of this is the backtracker, because it stores the entire kernel state which can get quite large, depending on the size of VM heap, i.e. the number of objects that are alive. A kernel state in JPF is the entire content of the simulated JVM.

For our models, the maximum size of the backtracker is when the system is at its deepest point in the state-space. All objects are already allocated before the model becomes multi-threaded and JPF starts saving kernel states. Therefore all kernel states in JPF are approximately of the same size. With this knowledge, it is possible to estimate how much memory is needed to make a full JPF run for our models. Since it is hard to guess how deep the state-space is, this can be estimated by running JPF once without saving any states in the backtracker. That way, JPF only runs through the state-space one time and does not backtrack. Once it has reached the deepest state (a state that does not have any paths that will lead to a new state) it will stop. The standard report will be printed, showing what the maximum depth has been. This is only possible, because the state spaces of our models are acyclical.

We ran four different models, the same models that were used to analyse Jackrabbit. The models were run with JantiensBacktracker1, a dummy backtracker that does not store any states (see section 5.7.1 for more details). The following list shows the maximum depth used, which was printed in the report:

- Model 1: estimated max depth: 2,556 states
- Model 2: estimated max depth: 58,091 states
- Model 3: estimated max depth: 96,606 states
- Model 4: estimated max depth: 7,924 states

Model 1 and model 4 both ran completely with JantiensBacktracker2, which stores only the useful states. This backtracker is further explained in section 5.7.2. Models 2 and 3 ran out of memory with JantiensBacktracker2 and could not finish. Model 1 used 226 MB of memory and model 4 used 669 MB of memory. See the sections 5.2 and 5.5.

To make a formula to estimate the memory that is needed to run a model with JPF, we need to know the amount of memory used and the number of states

that are stored at that moment. Table 1 shows some information of different models that were run with JPF[4], showing different snapshot of the memory at certain points in time. This data has been retrieved with YourKit Java Profiler. This is a program that allows one to look at the objects that run in the JVM, their size, number, etc.

The first column shows the model, the second shows the number of states that were in the backtracker at that point in time. The third column shows the memory size that those states used. The fourth column shows the other memory that JPF needed besides the kernel states. The fifth column shows the total amount of used memory and the sixth column shows the average amount of memory per kernel state.

Model	Number of states	Memory size kernel states	Memory other	Used memory	Average memory per state
Model1	636	35,464,008B	~24MB	58MB	~55,761B
Model1	1798	100,359,384B	~ 27MB	123MB	~55,817B
Model1	3210	179,289,480B	~42MB	213MB	~55,853B
Model2	1886	128,391,648B	~34MB	156MB	~68,076B
Model2	5596	381,087,168B	~46MB	409MB	~68,100B
Model2	7059	480,735,024B	~49MB	507MB	~68,102B
Model3	1085	70,937,488B	~29MB	97MB	~65,380B
Model3	3918	256,329,008B	~37MB	281MB	~65,423B
Model3	7353	481,115,408B	~48MB	507MB	~65,431B
Model4	689	38,693,648B	~24MB	61MB	~56,159B
Model4	3683	207,076,208B	~35MB	232MB	~56,225B
Model4	8554	481,021,840B	~48MB	507MB	~56,234B

Table 1: Data used to estimate the memory usage by JPF

As the table above shows, the average amount of memory per kernel state stays almost the same for each model. This can be explained because every model we used had the entire configuration (where all the classes are loaded) in the beginning of the model. After the initialization a second thread is started. Since JPF does not store any states when a model is single threaded, these kernel states are not stored in the backtracker. Therefore the states that are stored already have all the classes in them and do not grow significantly; the number of objects in the JVM stayed unaltered.

We want to know the total amount of memory that is needed to run the models completely. To compute this, we set out the number of states to the used memory at that moment for every model. These graphs are shown in figure 9. It shows the three points of measurement of every model in a colour. We assume that the points have a linear relationship, as the graph shows. Because they have a linear relationship a line can be drawn between them. The thin line that runs through the three measurement points is the trend line for that model and shows how the graph

[4] JPF configuration: JantiensBacktracker2, no state set, DFSearch, FilteringSerializer, CollapsingRestorer.

will continue in both directions. For every line we made a formula that describes the memory usages of that model. These formulas are:

Equation 1: Formula model 1

$$mem\_size\_1 = 0.0602 * states + 20$$

Equation 2: Formula model 2

$$mem\_size\_2 = 0.0678 * states + 28$$

Equation 3: Formula model 3

$$mem\_size\_3 = 0.0654 * states + 26$$

Equation 4: Formula model 4

$$mem\_size\_4 = 0.0567 * states + 22$$

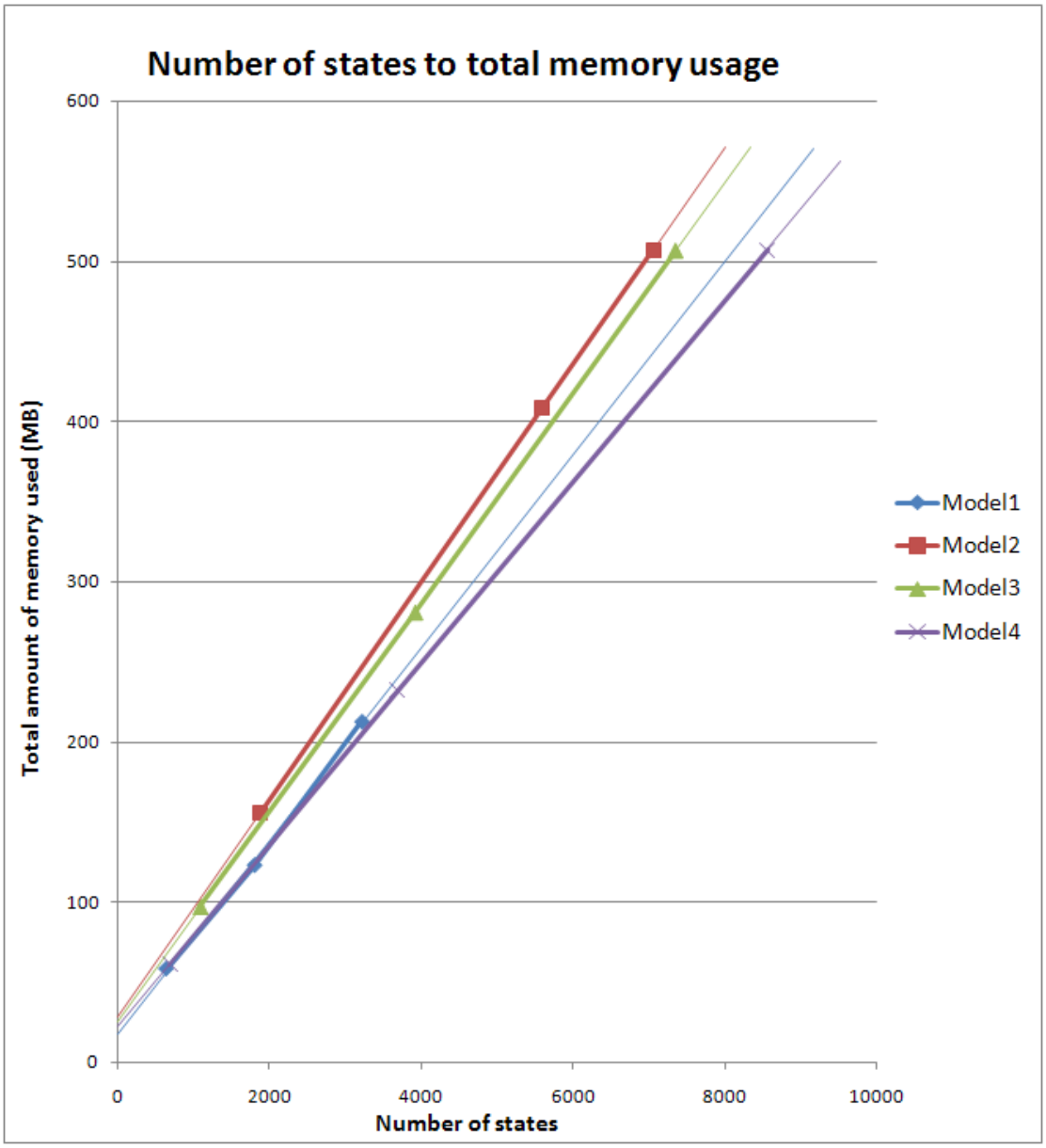

Figure 9: The memory usage for every model

With this information one can see what the total amount of memory usage is when the maximum depth of the state space is known. But the problem is that this is not known in advance. The run with JantiensBacktracker1, the dummy backtracker, gives an estimate of its depth, but is far from accurate. Because we know the amount of memory the models 1 and 4, we can look at the maximum number of states they used.

We know that model 1 used a total amount of memory of 226 MB. When using our formula for model 1, we know that the maximum number of states in the backtracker was approximately 3,421 states. The dummy backtracker found a depth of 2,556 states. The states that the dummy backtracker found needed to be multiplied by 1.34 to get the actual depth of the state space.

When we ran model 4, the model used a total amount of memory of 669 MB. When the formula for model 4 is used, we found that the actual depth of the state space must have been 11,409 states. The dummy backtracker only found 7,924 states. To get the actual number of states, the amount found with the dummy backtracker needed to be multiplied by 1.44.

The models 2 and 3 ran out of memory, so it is unknown how much memory they will actually need. To make an estimate we used equation 2 and 3. The problem is that we did not know the maximum depth of the state space. Therefore we used the amount found by the dummy backtracker and multiplied that by 1.5. We use 1.5, because model 4 has the largest multiply factor which is 1.44. Because we rather calculate too much memory, we round that number up to 1.5. When we did this, we came to the following approximation of the maximum memory of models 2 and 3:

- Model 2: 5,940 MB
- Model 3: 9,505 MB

We could not see if these numbers are actually correct, since we did not have a computer with enough memory. But since we have an exact formula that describes the memory usage and a rough estimation of the maximum depth of the backtracker, we hope these numbers will give a good approximation.

5.7 Extensions to JPF

In order to make JPF less memory consuming, we wrote several new backtrackers. Whereas for most model checkers, the passed list is the problem since it can grow exponentially. But in JPF, the waiting list that is implemented in the backtracker is the problem. The states in the backtracker are much larger than the states in the pass-list. The states in the backtracker represent the entire JVM heap while the states in the pass-list are hash codes that represent the states. For that reason four new backtrackers were built.

5.7.1 JantiensBacktracker1

JantiensBacktracker1 is a dummy backtracker that only simulates a backtracker. It is based on the default backtracker with one small change. It will only store the current state of JPF when the pushKernelState function is called.

In the default backtracker the pushKernelState function puts the current kernel state on the top of the stack. The stack represents the waiting list and the new state is pushed on to the top of that stack every time the pushKernelState is called. When JPF needs to backtrack the top state is popped of the stack by the popKernelState or the backtrackKernelState function.

This backtracker, JantiensBacktracker1, only stores the current state when it is called. When the store procedure is called again, the previous state is overwritten by the new current state so the waiting list is never be longer than one. The reason for storing one state instead of storing no states is because without saving any states, JPF will raise an exception.

JantiensBacktracker1 can be used for checking how deep the state space is. This backtracker does not take the deepest state per se, but gives you an idea of how deep the state space. This backtracker should not be used for actual model checking, because it only runs through the state-space once, therefore it only checks a single execution path.

5.7.2 JantiensBacktracker2

This second backtracker is also based on the default backtracker and only saves states that might be of importance in the future. These states can be described as states that have more than one choice left. This can be described as:

Equation 5: What makes a state interesting to save

$$Saved = (NumberOfChoices > 1)\ AND\ (NumberOfChoices - NumberOfVisitedChoices > 1)$$

When the Boolean is true, the state should be saved, otherwise not. These figures below will show some examples of states that should en should not be saved. Even though this backtracker does not save every state it passes, it does search the entire state-space.

When a state only has one unvisited choice left, it will not be saved. But that does not mean that its final transition is not executed. It will just not be saved, but the JVM will execute the instructions that belong to that last transition. When it needs to backtrack, it will go back to the last state that was saved and that has at least one more choices left. It is pointless to go back to a state that has no more choices left (since you just came from the last available), so that is why you can jump directly to the last state that had.

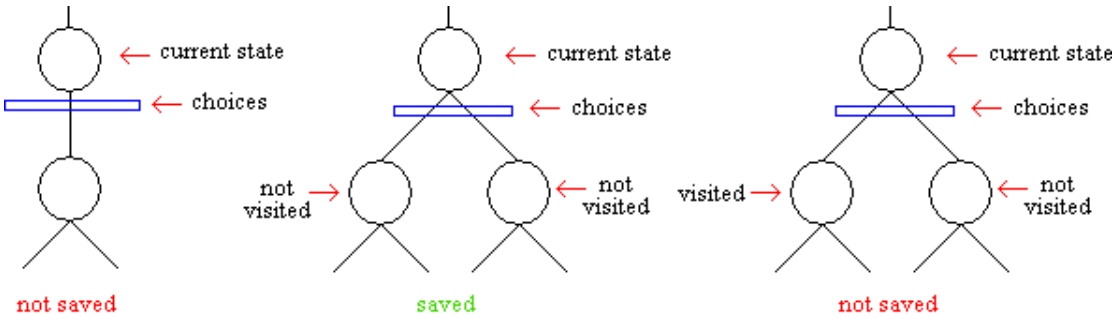

Figure 10: States that should be saved or not

Model 1 did not terminate with the default backtracker, but stays at a maximum memory usage of 601 MB. When using the same configurations of JPF except changing the default backtracker to JantienBacktracker2, the model only used 226 MB of memory. This is a reduction of at least 62%, which is significant. Model 4 ran out of memory when it had 1.8 GB at its disposal. When model 4 is ran with JantiensBacktracker2, it only used 669 MB of memory. This is a reduction of at least 63%, which is also significantly less.

5.7.3 JantiensBacktracker3

The third new backtracker is based on the default backtracker and JantiensBacktracker2. This backtracker does look at a state to see if it is worth saving and saves every x number of states. The x can be set in the default.properties file. For example if that number is two, then every other state will be saved if it is worth saving.

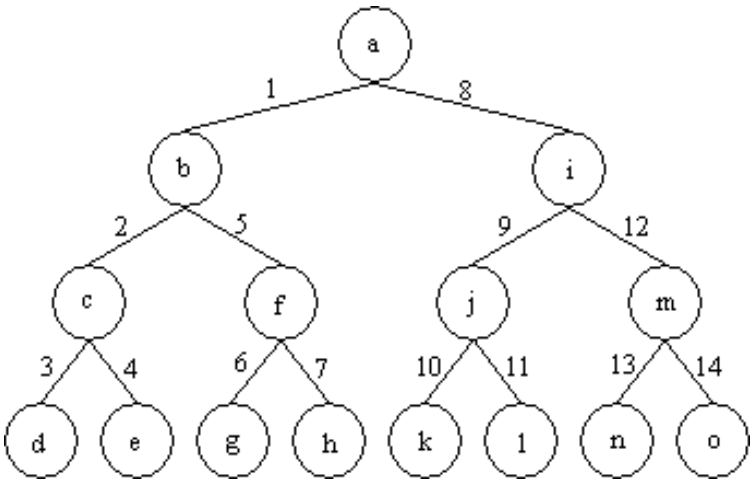

Figure 11: Example of a small tree like state-space

Figure 11 shows an example of a very small state-space to illustrate how JantiensBacktracker3 walks through it. Table 2 shows the transitions that will be taken when using the third backtracker. As can be seen in the column 'visited', 9 out of the 15 states have been visited (60%). The size of the stack has reached a maximum of 2 instead of 3, which would be the case if the default backtracker was used.

Transition	From	To	Visited	On Stack
1	a	b	a,b	a
2	b	c	a,b,c	a
3	c	d	a,b,c,d	a,c
4	d	e	a,b,c,d,e	a
8	a	i	a,b,c,d,e,i	-
9	i	j	a,b,c,d,e,i,j	-
10	j	k	a,b,c,d,e,i,j,k	j
11	k	l	a,b,c,d,e,i,j,k,l	-

Table 2: Transitions in the state-space with JantiensBacktracker3

This backtracker does not search the entire state-space, so it is very well possible that it will miss some problems. But it will save a significant amount of memory and still search a large part of the state-space. When using this backtracker, no guaranty can be given about problems in the code, but it might give a rough idea about possible problems.

It is difficult to say anything about the coverage of this backtracker. In the example 60% of the state space was covered. But this state-space was very small, three levels deep, and is a tree. Since this is not the case in every state-space, determining the coverage will be nearly impossible.

5.7.4 JantiensBacktracker4

The fourth backtracker is almost the same as the default backtracker. But instead of starting at the beginning of the state-space, it does not start until it reached a given depth and it would go until it reached another given depth.

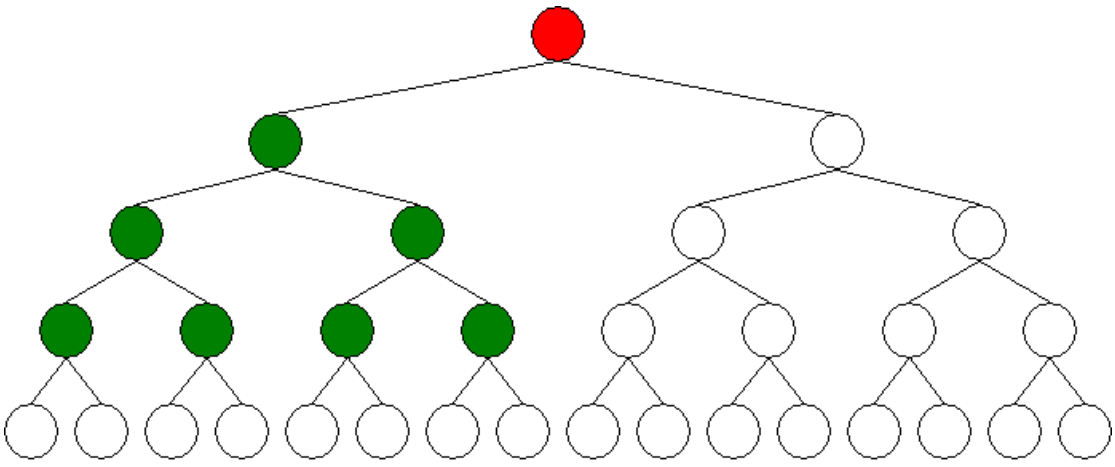

Figure 12: JantiensBacktracker4 in a state-space

Figure 12 shows an example of what part of the state-space will be searched with this backtracker. This example demonstrates what will be saved in the backtracker when JantiensBacktracker4 is used with only saving from depth 1 to depth 3. All the states before starting to save are also executed, but only once as a

linear execution path. In the example this is only the red circle. When the set depth is reached, the backtracker starts saving the states.

The first state that is saved in this example is the second state the backtracker passes, the first green circle. From there it saves all states it passes until it reached the depth of 3. Once it has reached that depth it will backtrack to the previous state.

Backtracker4 does not search the entire state-space, but only a part of it. If there is a problem in this part, this backtracker will find it. But when it does not find any problems, it does not mean there are none.

5.8 Results

It took some time before Jackrabbit could be run through because JPF did not support all functionalities needed. Jackrabbit used dynamic class loading and XML parsing, which are functionalities that are not supported by JPF. This made the start up of the project difficult and time consuming, but once the changes to the Jackrabbit source code were made, JPF had no more trouble accepting the code.

Jackrabbit could not be searched by JPF directly, because Jackrabbit is a library that offers access to a repository. JPF needs a program or model to search for problems. Therefore, four models were created that each touched a part of the Jackrabbit code where problems might exist. These models were run by JPF. By using models it was not possible to touch every part of the repository, so not the entire Jackrabbit code was tested for concurrency issues.

During the search for problems in Jackrabbit, none were found. None of the models finished when using the default backtracker, the state-space was too large to run through. For that reason, four new backtrackers were created that introduced a new way of searching the state-space.

JantiensBacktracker2 is the only backtracker that searches the entire state-space, but with some small optimisations it used less memory then the default backtracker, respectively 62% and 63% for the models 1 and 4. With this backtracker it was possible to run through the models 1 and 4. But the models 2 and 3 were still too large for this backtracker and could not finish searching the state-space due to a shortage of memory. For JantiensBacktracker2 it is possible to calculate the maximum amount of memory needed to finish the search.

For calculating this maximum, JantiensBacktracker1 is needed. This backtracker gives information that is needed to estimate the maximum amount of memory that JantiensBacktracker2 needs to run the same model. This information is the depth of the state-space. When the maximum amount of memory needed is estimated, it can be determined if the model will finish or not when run with the amount of memory at hand.

Memory issues were the biggest problem during this research. When searching for problems in Jackrabbit, the models that were created could not run because JPF ran out of memory every time. Especially when using transactions in the Jackrabbit models, the state-space exploded drastically. For that reason the models 2 and 3 could not be run completely.

It was possible to search the state-space of the models 2 and 4 partly with JantiensBacktracker3 and JantiensBacktracker4. During that search, no problems occurred. But not the entire state-space was searched, so the fact that no problems were found does not mean there are not any. Problems could still exist in the part of that state-space that was not searched.

The overall results of this project were both satisfying and unsatisfying. The satisfying part is that a lot of progress was made with improving JPF. The memory usage was reduced by creating extensions to JPF, formulas were created to estimate the maximum memory needed for a model and Jackrabbit was adjusted to be accepted by JPF. The unsatisfying part was that no problems were found in Jackrabbit. The initial goal of this project was to find the documented issues in Jackrabbit using formal methods (JPF). It is unfortunate that this goal was not accomplished. That no problems were found does not mean there are not any because not all the Jackrabbit code was checked.

Part 2

6. Context

This section describes the context of the second part of this research. It describes the project, its context and the research questions.

6.1 Problem Description

This part looked at the usability of Java PathFinder (JPF). Currently, JPF is mostly used in the academic community, but could also be used for commercial purposes in the future. This project investigated if JPF has the possibility to be used widely in the future. We looked at the tool with a user perspective, since the users have to use JPF. Until now, no one looked at how users see JPF. That knowledge is necessary to guide future development of the tool.

JPF claims to be a useful tool for both the academic community and industrial purposes. But at this moment, not many people are using it. According to the Technology Acceptance Model (TAM) [Davis89], two factors affect the usage of a tool, namely the usefulness and the ease of use. That is why this research will look at those points to see if JPF will be used in the future.

6.2 Approach

To find out how useful JPF and what affect the usability of the tool, we uses as a conceptual model the Technology Acceptance Model. With the conceptual model we created a questionnaire to gather data from the JPF users. This data was essential to determine the users' perspective on JPF that was needed to find the useful and less useful features. The questionnaire was send out to JPF users and the returned data were analysed to find the usefulness, ease of use and what features affect these subjects. A more detailed approach of this research can be found in chapter 8.

6.3 Problem Definition

6.3.1 Objectives

The objective of this research was to find out what the ease of use and usefulness of JPF are and what affects them. To know the features that affect the system usability means that one knows what to change in order to increase it. With this research we tried to see what needs to be changed to JPF to make it more useful for the users.

6.3.2 Research Questions

The following research questions will be answered in this research:

Q1: To what extent is Java PathFinder useful for finding concurrency issues?

Q2: *To what extent is Java PathFinder easy to use when searching for concurrency issues?*

Q3: *What features of Java PathFinder affect its usefulness and ease of use?*

6.3.3 Relevance

This research was relevant for both GX and the JPF community. GX is now able to better assess if JPF is useful in their software production process. For the JPF community it was relevant because it created an insight into what needs to be changed to JPF in order to make it more useful. The more JPF improves, the more people will start using it and more software can be model checked.

6.3.4 Products

At the end of this research project, these products were delivered:

- **Master Thesis**
  A written report describing the research period with at least a summary, the problem definitions, used methods/models, fundamentals and the conclusions.
- **Presentation**
  During the presentation all interested parties will be informed of the results and course of the research.
- **Summary Report**
  The users who filled in the questionnaire were promised a report with the results of this research. We have sent them a summary of the research results

7. Conceptual Model

The conceptual model that was used is the Technology Acceptance Model (TAM). There were several reasons why TAM was chosen as a model of this research. One reason is that TAM is a widely used model in the technology and ICT community. It is recognized as a good model and there are many articles written on the subject [Legris03]. Another reason is that the TAM model describes exactly what we want to do in this research. TAM measures the system usage. The perceived ease of use and the perceived usefulness are measured to determine the actual system usage. To know the usability of a new tool and how to increase it, one needs to know what features of the tool affect the usefulness and ease of use. Therefore we use the Technology Acceptance Model as our conceptual model, but included the features of JPF that affect the usability.

There were other models available besides TAM for this research. One example is TAM2 [Venkatesh00], which is an expansion of TAM. TAM2 looks at other aspects that affect the actual system usage besides the usefulness and ease of use, like experience, voluntariness, job relevance, output quality, result demonstrability, image and subjective norm. The reason we did not choose this model is because it is much more complicated than the original TAM model. Considering the lack of time for this research we did not have the time to look at all the aspects TAM2 covers.

Another model that could have been used was the Task-Technology Fit model (TTF) [Goodhue95]. TTF measures eight factors to determine how well a technology fits the users' tasks. These factors are quality, locatability, authorization, compatibility, ease of use/training, productions timeliness, systems reliability and relationship with users. But because this model looks at the individual performance of a tool, it was too specific and could not be used in our research. TTF does not look at the usability, but only to the individual performance. We want to look at the usability of JPF and what features increase the usability. This makes TTF not suitable for this research.

TAM suits our research best, because it looks at the usability of a technology. But because we also wanted to look at what affects the usability of JPF, we included this in our research. First we explain what TAM is; second we explain what part of TAM was used.

7.1 Technology Acceptance Model

The Technology Acceptance Model (TAM) is an information systems theory that models how users come to accept and use a new technology. TAM suggests that when users are presented with new software, there are several factors that influence their decision on whether to use it. These factors are 'perceived ease of use' and 'perceived usefulness'. Perceived ease of use (PEOU) is defined by Davis as "the degree to which a person believes that using a particular system would be free

from effort" [Davis89, p. 320]. Perceived usefulness (PU) is defined by Davis as "the degree to which a person believes that using a particular system would enhance his or her job performance" [Davis89, p. 320].

7.1.1 History

Since the seventies, researchers have been looking at factors that influence the decision making of the user on new software products and the integration and adoption of these products in a business. The different investigations have produced a long list of 39 factors that influence users in their decision [Bailey83]. Since the mid-eighties, this research has shifted more towards predicting the system use. To do so, management scientists turned to psychologists who already studied satisfaction in a larger sense. One of the models that was produced was the Technology Acceptance Model.

The Technology Acceptance Model was first proposed by Davis in his doctoral thesis in 1986 [Davis86]. TAM is one of the most influential extensions of Ajzen and Fishbein's Theory of Reasoned Action (TRA) [Ajzen77]. Since its development, it has been used, tested and expanded by many researchers. Some extensions to this model that were developed over the years are TAM2 [Venkatesh00] and Unified Theory of Acceptance and Use of Technology (UTAUT) [Venkatesh03]. Overall, TAM was empirically proven successful in predicting about 40% of a system's use [Legris03].

7.1.2 The Model

The Technology Acceptance Model as originally designed by Davis is shown in figure 13 [Davis86] and was an adaption of the Theory of Reasoned Action of Ajzen and Fishbein. The goal of TAM is to explain and predict the behaviours of people in a specific situation. Davis argued that Perceived Usefulness (PU) and Perceived Ease of Use (PEOU) are the most import factors that influence the users' attitude towards using the new software system. The attitude towards the system is a major determinant on whether the user will actually use the system or not [Davis86].

In figure 13, the boxes represent the factors that influence the usage of the system, the arrows represent the relationships between these factors. Design features directly influence PU and PEOU since they are external factors. This also means that changing the design features of an application will not directly influence the attitude towards the system, but they will change the PU and PEOU.

7.1.3 Usage

TAM is a widely used model in the information technology field. Since Davis introduced TAM in 1989, it has been used many times by many researchers to look at the rise of new information technologies. His article 'Perceived usefulness, ease of use, and user acceptance of information technology' [Davis89], which introduced

TAM, has been cited 978 times[5]. The other article that introduced TAM 'User acceptance of computer technology: A comparison of 2 theoretical models' by Davis et al [Davis.et.al89] has been cited 788 times[5].

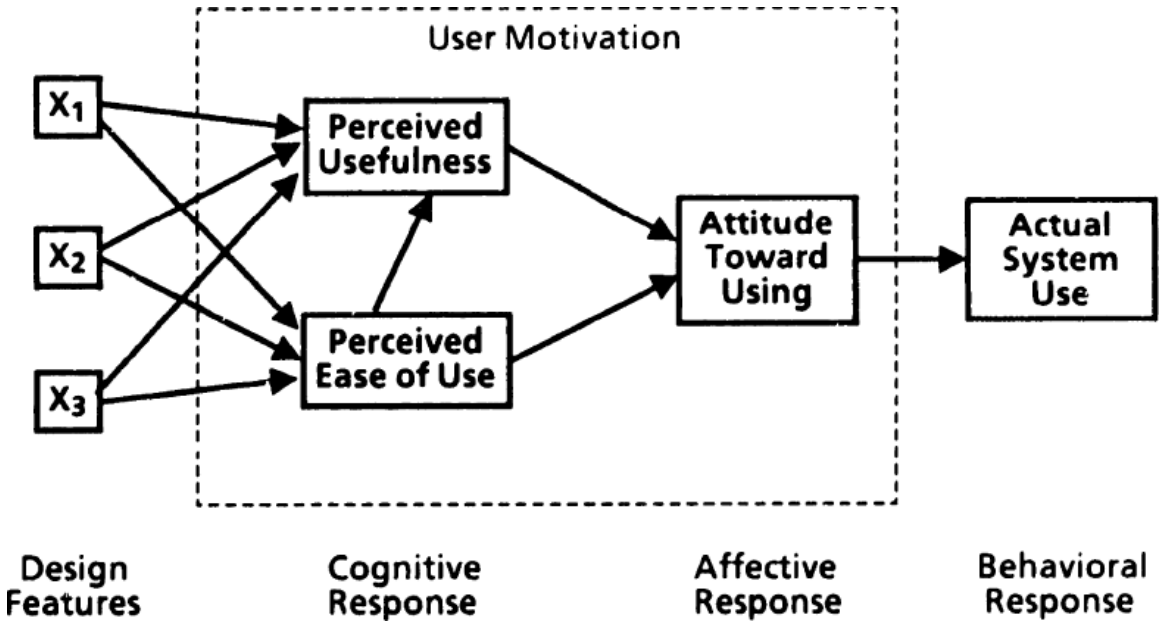

Figure 13: The Technology Acceptance Model

The most influential extension to TAM is TAM2, which was introduced by Venkatesh et al in 2000 in "A theoretical extension of the technology acceptance model: Four longitudinal field studies' [Venkatesh00]. This article has been cited 304[5] times since then.

TAM has been used on various kinds of information technologies. Technologies such as E-mail, voice mail, text-editors, spreadsheet programs, Windows 3.1 and debugging tools have been researched with TAM. Legris et al give a good overview of these different kinds of technologies in their article [Legris03]. Several of these researchers have also looked at the relationships between the usefulness, ease of use and system use [Davis.et.al89, Adams92, Segars93].

7.1.4 Criticism on TAM

Although TAM is a widely used theory to help understand and explain user behaviour of information systems, there is some criticism on the model. Legris et al. found three limitations in the use of TAM when comparing 22 articles on the subject:

> *" 1. Involving students*
> *Nine of the studies involved students. Although this minimised the costs, we think that research would be better if it was performed in a business environment.*
> *2. Type of applications*

[5] Web of Science, February 15th, 2008

We also noticed that most studies examined the introduction of office automation software or systems development applications. We think that research would benefit from examining the introduction of business process applications.

3. *Self-reported use*
   Since most of the studies do not measure systems use, what TAM actually measures is the variance in self reported use. Obviously this is not a precise measure. Not only is it difficult to measure rigorously, but it also involves problems. At best, self reported use should serve as a relative indicator." [Legris03, p. 202]

These limitations concern the use of TAM. Besides that, TAM itself also has some limitations. One of the limitations of TAM is that it does not cover all aspects of what affects the usefulness and ease of use. Venkatesh et al. argue that the social influence processes (subjective norm, voluntariness, and image) and cognitive instrumental processes (job relevance, output quality, result demonstrability) are not taken into account [Venkatesh00]. Therefore they considered TAM to be incomplete.

7.1.5 Current Research

Most of the current research on TAM is done on the extensions. Especially Venkatesh has done some large improvements to TAM in the past few years with the introduction of TAM2 and UTAUT. Although TAM is still used to measure the future usage of new computer systems, there is not much research to the model itself anymore.

7.2 TAM Used in this Research

Using TAM allowed us to answer the research questions of this research. But we did not look at the entire model, but concentrated on the perceived usefulness, ease of use and the features that affect them. To illustrate the part we did use, see figure 14.

We concentrate on the boxes within the dotted line. The definitions of perceived usefulness and perceived ease of use are taken from TAM, but are changed to fit the context of this research better. The definition of perceived usefulness for JPF is:

"The degree to which a person believes that using JPF would enhance the quality of software one builds"

We have done the same for the ease of use. The new definition of the perceived ease of use for JPF is:

"The degree to which a person believes that using JPF would be free from effort"

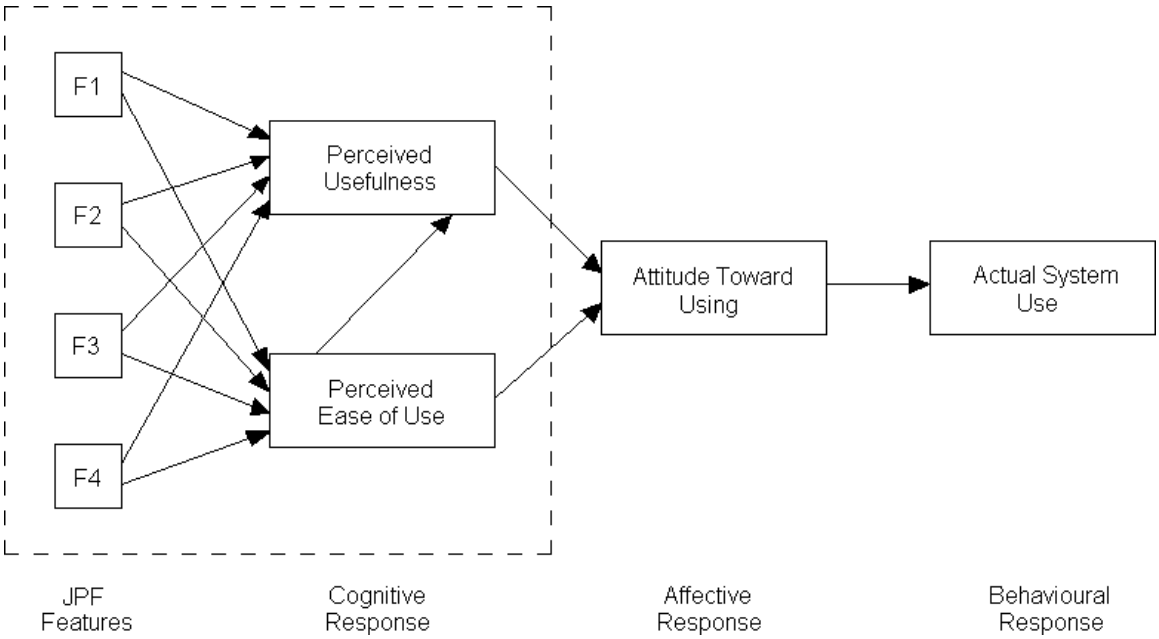

Figure 14: The part of the conceptual model used in this research

Both the subjects, PU and PEOU, were measured the same way as in TAM. Users were asked to fill in a questionnaire where they had to respond to statements and rank them on a scale from 1 (strongly agree) to 7 (strongly disagree). The system features that influence the usefulness and ease of use are not measured in traditional TAM investigations. To measure these, we asked the user some open questions about JPF.

With these open questions we asked users what they think should be changed or added to JPF to make it more useful or easier to use. With this information, we were able to fill in the functions of JPF that affect the usefulness and ease of use.

8. Research Strategy

To be able to answer the first two research questions (Q1 and Q2) correctly, we had to gather information on the usefulness and ease of use of JPF from the users. The standard approach used by TAM researchers to gathering this information is a questionnaire. TAM researchers have developed a list of questions to measure the perceived usefulness and perceived ease of use [Legris03]. Not every research uses the same questions, different selections were used.

For the first part of this research, four questions on each subject were picked from that list. These questions were adjusted to fit the JPF context, which mainly means that the words 'application' and 'tasks' were replaced by 'JPF' and 'finding concurrency issues'.

To answer the third research question (Q3), about which features of JPF influence the usefulness and ease of use, we had to create our own questions since this has not been done before. We chose to use open questions to ask the user why he/she finds JPF useful and easy to use. We also asked the user what he/she thinks should change in order to improve JPF. With the answers we tried to find what users find so useful and easy to use and why. That information was used to answer the third research question.

The questionnaire (see appendix B) is split into two parts, namely the first part that contains questions that concern the research questions Q1 and Q2. The second part contains the questions that concern research question Q3 and some general questions about the user's background. These variables were used to determine whether we had covered a wide part of the population. The larger the part of the population covered, the more accurate the outcome of the research is.

Due to the limited amount of time for this research, we did not wait any longer than 2 weeks for responses from users.

8.1 Gathering Data

There were different methods that could have been used to send the survey to JPF users, namely an email survey, mail survey, telephone survey, personal interview and web survey. For this research we chose to use an email survey for the several reasons. Email is fast; one can send a lot of emails at once and receive many answers in a short period of time. There would also be no costs involved. A disadvantage of sending an email survey is that one might have to wait a while for an answer, since not everyone answers their emails quickly. Also, many people never answer their email.

Because this project had a limited amount of time and resources available, we chose to use an email survey to collect the data needed. This survey was send to

all the users that are registered at the JPF users' mailing list. This mailing list contains users of JPF from all over the world. The size of this list[6] is 62, the size of the entire JPF community is not known (since not every user is a member of the mailing list). The sample size needed can usually be determined by the Simple Random Sampling (SRS) and a finite population correction [Retzer]. To use that technique, the population size is needed, which was not available to us. Therefore, it was impossible to determine the sample size needed to make the outcome valid. Because we could not determine the sample size, we took any sample size we could get and had to take that into account when evaluating the results. Without a known population and sample size, the accuracy of the outcome would be uncertain.

8.2 Avoiding Possible Problems

While doing this research, several problems could have occurred that could jeopardise the quality and outcome. The most important problem that could have occurred was the lack of responses. It was possible that (almost) no one would respond to the survey. Without responses, it would have been impossible to say anything about the usefulness, ease of use or about what JPF features affect those items.

To increase the change for response we offered a copy of the results to everyone who completed the survey. We also kept the survey small, so it would not take too much time to complete it. The survey was send as a plain text file in an email, since most users of JPF are researchers and many of them work on Unix or Linux machines. Because those machines do not have MS Word installed, it would be less difficult for them to answer the questionnaire in a simple text email. We also send a reminder after one week, asking the user to send in the questionnaire. By doing these things, we hoped more people would answer our survey.

Another problem that could have occurred was that the data received would not valid. This could have happened when for example all users had the same background and used JPF for exactly the same thing. That way the answers they gave would probably be the same and no conclusions can be drawn from that. This is called a biased sample, a group that has the same view on a product. Possible biased samples for this research were:

- Developers of JPF; they will probably be very favourable about JPF.
- Scientists that use JPF in their research; probably a bit critical, but generally favourable, otherwise they would not have chosen JPF for their research.
- Ex-user of JPF; this group is probably not favourable about JPF, otherwise they would still use is.

We had to be careful that not all respondents were from the same group, but that they have different backgrounds. This way a biased sample would be avoidable and the results would be better.

[6] February 6th 2008

One problem that could easily be avoided is that the questionnaire is not clear enough. Questions could be interpreted in different ways or the way of answering may not be understandable. To avoid this, it is good practise to let the questionnaire be checked before sending it out. Our questionnaire was tested by Martijn Hendriks, who is a user of JPF and a member of the target group. Possible remarks on the clarity of the questionnaire were changed before sending the survey to other JPF users.

8.3 Solving Problems

The biggest problem that might have occurred is not enough responses from JPF users. Even though we have tried to avoid any problems from happening, it was no guarantee that none would occur. When no one had answered to the survey, only two completed surveys would have been available for this research, namely those from Martijn Hendriks en myself. This would not give a good outcome of the usefulness, ease of use and affecting features, but would still be better than no result at all. When this would have been the case, we had to take into account that the results would be far from solid.

8.4 Expected Outcome

Every research has an expected outcome. Because we worked with JPF ourselves, we had some expectations about the outcome of the survey. The expected outcome of every item of the survey is discussed below.

For the usefulness we expected a moderate outcome. JPF could be very useful for small programs, but has limitations for large applications. When applications become too large, the shortage of memory and duration of the runs become issues. This makes JPF less useful for large applications and that affects the usefulness depending on what the users use JPF for. JPF has good features and can usually find concurrency problems faster than other techniques/tools. Therefore, the expected outcome of the usefulness was above average.

The expected outcome of the ease of use was similar to the expected outcome of the usefulness. Some parts of Jackrabbit ran directly in JPF, but some had to be adjusted in order for JPF to accept them. Because JPF does not support some features like dynamic class loading and file I/O, large scale real-life programs have to be adjusted so they do not use these features anymore. For smaller programs that do not use these features, JPF is fairly easy to use. So the outcome of the ease of use also depended on what the users use JPF for. The expected outcome was therefore a little above average.

The features of JPF that influence the usefulness and ease of use could have been the following:

- The amount of Java packages supported
- The quality of the outcome reports produced

- Extendibility of the code
- Documentation available

Although these features might influence the usefulness and ease of use, we did not know this for sure. How users answered this question depends on their usage of JPF and could therefore not be accurately predicted.

9. Analysis and Results

Unfortunately only seven people responded to our survey (including me), which means 11% of the list responded. From those seven people, six filled in the questionnaire correctly and one only filled in the second half of it. So in total six and a half survey answers were available, for the perceived usefulness and ease of use six surveys, and for the usability features seven surveys.

Because the amount of answers to our survey was so small, it was hard to make the outcome of this research valid. But we tried to look at the outcome and find in what direction the usefulness and ease of use tend to go and what features might influence them.

9.1 The Respondents

The respondents represent a part of the entire population. It is important that one knows the respondents to see they are not all the same kind of people. This would make the outcome less valid, because then a part of the entire population of users is not taken into account.

In our case the coverage of the population was quite wide. In total two students, four software engineers and one scientist responded. Also the level of education of the respondents varied; there was one Bachelor, four Masters and two PhD's. The years of experience with model checking varied very much. All the students and one engineer had only one year of experience or less. While the other engineers had between three and six years of experience and the scientist had eight years of experience. The amount of experience in Java did not vary a lot, between three and seven years. Also the age did not vary much, most of the respondents were in their twenties, one was in his thirties and one in his forties. All users only had a brief experience with JPF, this varied between three months and one year. The purpose of using JPF also varied between the respondents, four people used it for finding concurrency issues, one for building an extension, one for experimental usage and one for comparison against other software model checkers.

The most important parts of the population were covered with our respondents, because we had people with different job backgrounds, different amount of experience in model checking and different educations. The fact that all the respondents had only brief experience with JPF is a plus. TAM is usually used on people who have little experience with the new tool/technique, because it aims to determine whether a user will use it in the future [Davis93]. When the respondents are already long-term users, it might be unnecessary to determine the future usage.

Overall we are pleased with the coverage of the population with our respondents. We did not get a biased sample, but covered a wider part of the entire population. This will make the outcome of the research slightly better.

9.2 Perceived Usefulness

For the perceived usefulness there were six surveys available. The answers on each question ranged quite a bit, especially for question 4, where two people answered with a 3 and one person with a 7. Overall it can be concluded that the people are slightly positive about the perceived usefulness of JPF. The average perceived usefulness was 3.00. Table 3 shows the results of the perceived usefulness questions.

	Q1	Q2	Q3	Q4	Average
Mean	3.50	2.33	2.00	4.17	3.00
Median	3.50	2.50	2.00	4.00	3.00
Standard Deviation	1.05	0.82	0.89	1.47	1.06
Minimum	2	1	1	3	
Maximum	5	3	3	7	

Table 3: Results perceived usefulness

Questions 4, about the whether they think JPF is useful in their job, people were slightly less positive. Question 2 and 3 scored quite high with a low standard deviation, these questions cover whether JPF is capable of finding concurrency problems quickly and problems that are otherwise hard to find. Since this is the general purpose of JPF, the tool succeeded on what it promises.

9.3 Perceived Ease of Use

For the perceived ease of use there were also six surveys available. For the perceived ease of use, the respondents were less positive. The overall outcome is 4.75, which was slightly negative. Table 4 shows the results of the perceived ease of use questions.

	Q5	Q6	Q7	Q8	Average
Mean	4.17	4.67	5.00	5.17	4.75
Median	3.50	5.00	5.00	5.00	4.63
Standard Deviation	1.94	1.51	1.41	1.17	1.51
Minimum	2	3	3	4	
Maximum	7	7	7	7	

Table 4: Results perceived ease of use

For the perceived ease of use, the range of answers was much wider. The standard deviation was the highest for the fifth question. Because the range was so wide, the outcome is less valid. But even so, it can be said that the outcome tends to be slightly negative. This can also be concluded from the fact that almost every respondent answered with a 4 or higher. Only one two and five threes were given in total.

The answers of one person, the principle scientist, are extremely negative and differ from the answers of the others. The answers that were given by the scientist were all sevens. The rest of the respondents responded more moderately negative.

9.4 Usability Features

There were seven surveys available for the usability features. Four questions were about the usability features in the survey, two on the usefulness and two on the ease of use. There were several features named in the answers to those questions, some more than others.

The first question (Q10) was why they find JPF useful or not. Six out of seven people answered that they find JPF useful for finding concurrency problems that are otherwise not easily found. One person answered that JPF was not very useful, because the costs of using it exceeded the value of its benefits. Unfortunately none of the people that responded sad anything about exactly what features of JPF he/she finds useful.

The second question (Q11), on the other hand, about what needs to be changed/added to JPF to make it more useful; the respondents had a lot of suggestions. The following features were named (between the parentheses it says how many times it was mentioned):

Needs to be improved:

- Better and more documentation (2)
- Better memory usage (1)
- Support more of the Java API (1)
- Better/simpler configuration, better intuitive runtime defaults (1)
- More sensible default feedback (1)
- Better integration with JUnit (1)

Should be added to JPF:

- A stable release (2)
- Graphical user interface (1)

There were many ideas on what needs to be changed to JPF to make it more useful. There were not many respondents that mention the same issues. Only better/more documentation and a stable release version are mentioned twice. This means that more people think that it should be changed and therefore those items are a bit more important than the others.

The third question (Q12) asked the users why they find JPF easy to use or not. Most respondents answer that they find JPF hard to use. Reasons for this, that are mentioned, are that it is complicated, too many bugs, difficult to learn and no good documentation.

There were also several ideas on what needs to be changed/added to JPF to make it easier to use. The following features were named:

Needs to be improved:

- Better and more documentation (5)

- Better/simpler configuration, better intuitive runtime defaults (2)
- More examples (1)
- Better integration with JUnit (1)
- Reduce features to only the truly useful once (1)
- More sensible default feedback (1)
- Support more of the Java API (1)

Should be added to JPF:

- A stable release (1)
- Distributed JPF (1)

The most important feature that needs to be changed to JPF is more and better documentation. Five out of seven people mentioned this to improve JPF. The other important issue that needs to be changed is a better configuration options. The current configuration is considered to be too difficult.

9.5 Results

The results match the expected outcome most of the time. The expected outcome of the usefulness was above average (average being 4). As can be seen in section 9.2, the actual result for the perceived usefulness is 3.00 and this value was expected. JPF is found to be a bit useful for finding concurrency issues in Java code.

The results for the ease of use did not match the expected outcome of this item. As can be read in chapter 8, the expected outcome of the ease of use was also a little above average. But with the final result of 4.75, this was lower than expected. The users found JPF not to be very easy to use.

JPF features that influence the usefulness and ease of use are the most interesting part of the results of this research. With the low number of responses we got many features that were mentioned only once. This makes it hard to determine whether these features actually influence the usefulness and ease of use. There were a few features that where mentioned several times and can be stated to have an influence on these subjects.

The features that were mentioned several times and can be considered to have an influence on the usefulness and ease of use are:

- Available documentation
- Stable Release
- Good/simple configuration options

The other features that could enhance the usability of JPF and were mentioned once in the questionnaire were:

- More examples

- Better integration with JUnit
- Reduce features to only the truly useful ones
- More sensible default feedback
- Support more of the Java API
- Distributed JPF
- Graphical user interface
- Better memory usage

From these last features it cannot be said with certainty that they influence the usefulness and ease of use because they were only mentioned once, but they might have some effect on it. This has to be further researched in the future. Future research has to prove whether the features above have any effect on the perceived usefulness and perceived ease of use. Three out of the four features that have been predicted to have an influence on the usability were also mentioned in the questionnaire. But only one of them was in the first list with features that were mentioned multiple times.

It is unfortunate that we had very low response to the survey. This made the outcome of this research less valid and means that more research is needed in the future. But even though the results were less accurate, we were still able to reveal some features of JPF that need to be improved to make it more useful and easier to use.

One other remark that has to be made about is that respondents generally see usefulness and ease of use as the same thing. This can be concluded from the fact that most respondents answered similar things in both categories. Although there is an essential difference between the definitions of the two terms, users see it as a similar thing. In the future it might be wise to enter a definition of these terms to the survey so there would be no confusion.

10. Conclusions

This section gives the conclusions that we drew from our research. It first answers the research questions. Second it gives the overall outcome of this project and third a discussion.

10.1 Research Questions

In this project we tried to answer several research questions that were mentioned in the context chapters. First we answer the technical question by starting to answer the sub questions and then the main question. Second we answer the management questions.

How useful are model checkers for analysing real-life software systems?

In this research we used Java PathFinder as a model checker to analyse a real-life software system. Although the idea of using model checkers on real-life applications is very appealing, the fact is that most model checkers are not useful for model checking real-life software systems.

There are very few model checkers that actually work directly on the software code. This is a necessity, because otherwise a new model has to be made of the system, mistakes can be made in the translation and it will take more time. A model checker that runs directly on the source code would be far easier. There are only a few of those model checkers available, Java PathFinder being one of them.

The main problem with all model checkers is the state space explosion. This is also the case for Java PathFinder, when trying to analyse a simple model with only two threads, the state space grew exponentially, taking up a lot of memory and time. Since most real-life software systems have multiple threads and a huge state-space, it would be very difficult to model check the entire software system.

Model checkers could be very useful for analysing real-life software systems, but they are not yet well enough developed to run smoothly on large business applications. They can find problems in software, but only in small programs. In the future, when model checking techniques are improved, we think it would be possible to use it on large scaled applications.

Is it possible to use Java PathFinder in the build-lifecycle of a Java program, for example in the build of Apache Jackrabbit?

Technically it would be possible; in practice it is not yet feasible. Like we mentioned above, JPF is not yet suitable for analysing real-life software systems. This is also the case for JPF. JPF can now only be used to analyse parts of software systems or small software programs. So technically it will be possible to use JPF in the build of a Java

program, such as Apache Jackrabbit, but since most real-life Java applications are too large for JPF, it will take too much time and memory to complete.

JPF does have potential to be used in the build-lifecycle of Java programs, but first it has to be improved before it can be included. It needs to consume less memory and has to finish in a reasonable amount of time, for example within one night. When this is possible, JPF can be included in the build-lifecycle.

Is it possible to find the known deadlocks in Jackrabbit with model checkers?

Maybe, but we did not find any in this research. Although no deadlocks were found in Jackrabbit using JPF, we still belief it should be possible. Jackrabbit is a complex system with quite many layers that the data needs to pass before being stored in the database. It is very well possible that our models did not touch the right part of the code to trigger the deadlock.

The most difficult part of using JPF to find deadlocks in Jackrabbit is to create correct models that will touch the right part of the code. Because we did not succeeded in building these models does not mean it is not possible. We do think that knowing where deadlocks might occur is a necessity. Jackrabbit is too large to just haphazardly start searching. To increase the chances of finding concurrency issues one needs to search in the right direction. JPF can find deadlocks in example programs, so there is no need to assume it could not find them in a large program like Jackrabbit. When searched the right way, we think it should be possible to find the known deadlocks in Jackrabbit, but it will take time.

Is it feasible to use model checking technology to analyse a real-life software system, for example Jackrabbit?

Not yet. There are model checkers available these days that run directly on software source code, for example Java PathFinder. But because the state-space of real-life software systems is generally quite large, it is impossible to search the entire state-space within a reasonable amount of time/resources.

In the future, further research is needed to improve software model checkers so they will be able to handle real-life software applications. Chapter 11 will give some more information on the kind of research that might be needed. Now it is only possible to search a small part of the software system, so the chance the one misses some problems is reasonably large. In the future it might be feasible to use model checking technology to analyse real-life software systems, such as Jackrabbit.

To what extent is Java PathFinder useful for finding concurrency issues?

To measure the extent in which Java PathFinder is useful, we sent out a survey to JPF users. Java PathFinder is considered to be slightly useful by the users that filled in our questionnaire. Most users found it slightly useful for finding concurrency issues. JPF scored a 3.00 on a scale from 1 to 7 (1 being very useful). This score was about

the same as what we expected from our own experience. Unfortunately this score is not exact enough, due to the lack of responses (only 7 people responded to our survey).

To what extent is Java PathFinder easy to use when searching for concurrency issues?

With our survey we also tried to measure the extent to which Java PathFinder is considered to be easy to use. The users who filled in the questionnaire considered JPF to be slightly difficult to use; they gave an average score of 4.75 on a scale from 1 to 7. Generally they found it difficult to use and that there were too many bugs in the system. The final score was lower than expected, from our own experience we found JPF not easy to use, but not difficult either. There is a lot of room for improvement to make JPF easier to use.

What features of Java PathFinder affect its usefulness and ease of use?

Even with the small amount of responses we were able to find three features that affect the usefulness and ease of use of JPF. These features are available documentation, stable release version and good/simple configuration options. Those features were the only ones that are mentioned multiple times by our respondents. Several other features were mentioned too, but only once and therefore we cannot say with certainty that they will affect the perceived usefulness and ease of use. Further research is needed to determine if these features actually have an influence.

Because JPF is found difficult to use and only slightly useful, we suggest several improvements be made. As the users already mentioned, there needs to be more and better documentation, a stable release version and a better and simplified configuration scheme. In our work we also found these features not adequate. And the most important feature that needs to be changed, in our point of view, is the memory consumption. At the present moment, JPF uses too much memory to search the state-space. Better memory usage is needed to assure that JPF can be used for large real-life applications in the future.

10.2 Overall Outcome

This research looked at the use of Java Pathfinder for verifying software, in our case Jackrabbit. We only partly succeeded in this goal. Although we were able to use the model checker JPF on a business application, we did not find the concurrency problems we hoped to find. There were two reasons we did not succeed in this goal. The first is that we might have used the wrong models. Jackrabbit is a large system and it is possible our models did not touch the piece of code that triggered the deadlock. And second because we were not able to model check all the models fully or build bigger ones.

We were able to make some small improvements on JPF concerning the memory usage. One new fully working backtracker was build that does not save

unnecessary states. This reduced the amount of memory needed to run a model. But it did not reduce the amount of memory enough, and we were still not able to run all our models. But the improvements that were made can be considered a success.

We also looked at what other users think about JPF. Even with the low amount of responses, we were able to see what people think about Java PathFinder. We also found some features that need to be changed according to those users in order to make JPF easier to use and more useful. This is a small success and we hope that the JPF community will improve these features and look for more features that need to be changed.

10.3 Discussion

Java PathFinder is a typical academic tool in our opinion. It is developed by a NASA research group to experiment with model checking on Java software. The tool is mainly developed to test new techniques and ideas, and to write papers about. Because it is mainly used for that purpose, it has not been commercialized yet.

Like many academic tools, researchers concentrated on the usefulness of JPF. By adding more and better technical features, they hope to improve the tool and write papers about that. But by doing so, they forget to make it easy to use. Adding things like a good interface, configuration options and documentation is not interesting from an academic point of view, but it does cost time and money. Therefore, these things are usually not added, which makes it difficult for any academic tool to cross to the commercial field. These characteristics can also be found in JPF.

Model checking is still an ongoing process. This year, Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis won the Turing Award for their original and continuing research in the quality assurance process known as model checking [ACM]. The Turing award is considered to be the most prestigious award in computing. It is a yearly award that is offered to computer scientists and engineers who created the systems and underlying theoretical foundations that have propelled the information technology industry [ACM].

More and more companies, mainly large corporations, are adopting model checking in there development process. Common examples of the usage include verification of the designs for integrated circuits such as microprocessors, as well as communication protocols, software device drivers, real-time embedded systems, and security algorithms [ACM]. Companies like Intel and Google are already using the technique to improve their products. But even though model checking is getting more and more adopted, it is still almost only used in large corporations and not by small and medium sized businesses. We think that a reason for that may be the complexity of model checking and the time it takes to verify a system. Small and medium sized companies need a shorter ‘time to market’ for their products then large corporations that can afford a longer development process.

11. Further Research

During this research some areas were found that need more research in the future. They are addressed in this chapter.

The main problem we discovered in JPF was the state-space explosion and the amount of memory needed to store the state-space. Further research is needed to find new ways to improve this. When the amount of memory that the state-space needs is reduced, larger models can be checked.

One way to reduce the state-space might be to not use JPF on an entire model, but use it more like a unit test to verify a very small part of the system. When this is done, JPF might be able to check small and critical parts of a software system. This will not give any guarantee about the absence of concurrency issues, but it might improve the quality of the software. Further research is needed to see if it is possible to use JPF in this way and if it improves the quality of the software that is checked.

Another way to reduce the size of the state-space might be to enter JVM scheduling into JPF. At the present moment, JPF searches the entire state-space and checks every state. In real life, there are many states that can never be reached because of the JVM scheduler. Therefore it might be an option to merge the JVM scheduling algorithm into JPF to determine which states to enter and to save. By doing this, it might be possible to reduce the state-space drastically. The down-side is that not all problems are found, because not every state is checked. But the problems that are not found cannot happen in real-life.

Further research is also needed to see what other features of JPF need to change in order to improve the usefulness and ease of use. Therefore the conceptual model en survey we created can be used on a larger population. More responses are needed to make accurate predictions on the features that need to be improved.

12. Bibliography

12.1 Literature

[Adams92] Perceived Usefulness, Ease of Use, and Usage of Information Technology: A Replication, D.A. Adams, R.R. Nelson, P.A. Todd, MIS Quarterly 16, 1992

[Ajzen77] Belief, Attitude, Intention, and Behavior: An Introduction to Theory and research, I. Ajzen, M. Fishbein, Psychological Bulletin 84, 1977

[Bailey83] Development of a tool for measuring and analysing computer user satisfaction, J.E. Bailey, S.W. Pearson, Management Sciences 29, 1983

[Barik06] Introducing the Java Content Repository – T. Barik – IBM DeveloperWorks – August 23, 2005, updated June 27, 2006

[Behrmann03] To Store or Not To Store – G. Bergmann, K. Larsen, R. Pelánek – Proceedings of the 15th International Conference on Computer Aided Verification – Volume 2725 of Lecture Notes in Computer Science, July 2003

[Biggs03] Discrete Mathematics – Norman Biggs – Oxford University Press – 2003 – ISBN 0918507178

[Coffman71] System deadlocks – E.G. Coffman JR, M.J. Elphick and A. Shoshani – ACM Computing Surveys (CSUR) – Volume 3, Issue 2, June 1971

[Davis86] A technology acceptance model for empirically testing new end-user information systems: Theory and results, Doctoral dissertation, Fred D. Davis, Sloan School of Management, Massachusetts Institute of Technology, 1986

[Davis89] Perceived Usefulness, Perceived Ease of Use, and user Acceptance of Information Technology, Fred D. Davis, MIS Quarterly, September 1989

[Davis.et.al89] User Acceptance of Computer Technology: A Comparison of Two Theoretical Models, Fred D. Davis, Richard P. Bagozzi, Paul R. Warshaw, Management Science, Augustus 1989

[Davis93] User Acceptance of Information Technology: System Characteristics, user perceptions and Behavioral Impacts, Fred D. Davis, International Journal of Man Machine Studies 38, 1993

[Fielding05] JSR-170 Overview: Standardizing the Content Repository Interface – R.T. Fielding – Day Management AG – March 13, 2005

[Goodhue95] Task-Technology Fit and Individual Performance, Dale L. Goodhue, Ronald L. Thompson, MIS Quarterly, June 1995

[Legris03] Why do People Use Information Technology? A Critical Review of the Technology Acceptance Model, Paul Legris, John Ingham, Pierre Collerette, Information and Management 40, 2003

[Mahapartra00] Transaction Management under J2EE 1.2 – Sanjay Mahapatra – Java World – July 14, 2000

[Peled97] Ten Years of Partial Order Reduction – Doron Peled – Computer Aided Verification – 10th International Conference, CAV'98 – June/July 1998

[Patil06] What is Java Content Repository – S. Patil – O'Reilly OnJava.com – April 10, 2006

[Segars93] Re-Examining Perceived Ease of Use and Usefulness: A Confirmatory Factor Analysis, Albert H. Segars, Varun Grover, MIS Quarterly, December 1993

[Shub03] A unified treatment of deadlock – C.M. Shub – Journal of Computer Sciences in Colleges – Volume 19, Issue 1, October 2003

[Sutherland07] Distributed Scrum: Agile Project Management with Outsourced Development Teams – Sutherland, J., Viktorov, A., Blount, J., and Puntikov, N – HICSS'40, January 2007

[Venkatesh00] A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies, Viswanath Venkatesh, Fred D. Davis, Management Science, February 2000

[Venkatesh03] User Acceptance of Information Technology: Toward a Unified View, Viswanath Venkatesh, Michael G. Morris, Gordon B. Davis, Fred D. Davis, MIS Quarterly, September 2003

[Visser03] Model Checking Programs – W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda – Automated Software Engineering Journal – Volume 10, Number 2, April 2003

12.2 Websites

[ACM] Turing Award press release – The Association for Computing Machinery (ACM) – http://www.acm.org/press-room/news-releases/turing-award-07

[Apache] The Apache Software Foundation – http://www.apache.org

[CMS] CMS Matrix – http://www.cmsmatrix.org/

[GX] GX Creative Online Development – http://www.gx.nl

[Jackrabbit] Jackrabbit project – Apache Software Foundation – http://jackrabbit.apache.org

[JCR-447] Issue JCR-447 – Deadlock on Concurrent Commit/Locking – http://issues.apache.org/jira/browse/JCR-447

[JCR-962] Issue JCR-962 – Deadlocks on ConcurrentVersioningWithTransactionsTest – http://issues.apache.org/jira/browse/JCR-962

[JCR-1148] Issue JCR 1148 – NullPointerException in ItemState – https://issues.apache.org/jira/browse/JCR-1148

[JCR-1412] Issue JCR-1412 – Java Based Test Configuration of Jackrabbit (No repository.xml Needed) – https://issues.apache.org/jira/browse/JCR-1412

[JPF] Java PathFinder – http://javapathfinder.sourceforge.net/

[JSR-170] Java Specification Request 170 – http://jcp.org/en/jsr/detail?id=170

[JSR-166] Java Specification Request 166 – http://jcp.org/en/jsr/detail?id=166

[Retzer] Introduction to Survey Sampling, Karen Foote Retzer, spring 2007, Survey Research Laboratory, University of Illinois at Chicago, http://www.srl.uic.edu/SEMINARS/sampintro.htm

[Spin] The Spin Model Checker – http://www.spinroot.com

[Uppaal] Uppaal – http://www.uppaal.com/

Appendix A. GX Company Description

GX is a successful web technology specialist and the largest independent supplier of web content management solutions in the Netherlands. GX's perceptive view of the market and clear strategy strongly differentiate the company from its competitors:

Figure 15: GX logo

A.1 Vision

The internet is a mature communications medium that now also offers attractive commercial opportunities. The second internet revolution, or more accurately evolution, has already started. Less conspicuous than the first, but with far-reaching consequences. The behaviour and expectations of consumers who currently use the on-line channel have changed fundamentally. Moreover, today's possibilities encourage and reward the new consumer's changed behaviour and expectations. The generation that will flow onto the labour market in a number of years sees the internet as a channel without limitations and the difference between the traditional consumer and the on-line consumer will soon disappear.

Existing software systems that support business processes and handle transactions are based on internal processes. The gap between consumer expectations and existing business software is becoming increasingly larger and will be more and more difficult to bridge in the future. Software suppliers will need to approach business processes from the opposite direction in order to satisfactorily service the new digital consumer, in other words from the outside in.

A.2 Mission

GX's objective is to support organizations and businesses that focus on the modern digital consumer and improve their competitive position by developing products and solutions that support the business processes involved in an appropriate way. Regardless of what our customer's customer wants to do on-line - buy goods, provide information, lodge a complaint, play games, access entertainment, submit an enquiry, participate in on-line activities, etc. - GX, as the leading supplier in the Netherlands, aims to provide the best systems and solutions.

A.3 Strategy

GX's strategy is based on the principle of ‘Outside in’. An increasing share of business processes is initiated from outside the organisation via the on-line channel. GX

WebManager was specially developed to support processes of this nature. A robust and flexible platform that acts as a stable foundation for the many solution frameworks offered by GX and GX's implementation partners for specific market segments or applications. This approach ensures maximum flexibility and effectiveness in adapting to the continuous stream of new developments in this field and making them available to customers. A natural consequence is that the functionality of GX WebManager will infiltrate ever further into the organisation, but always starting from the outside. Outside in.

Source: GX homepage [GX]

Appendix B. Questionnaire

This questionnaire contains some questions about your experience with Java PathFinder (JPF). All questions in the first half should be answered on a seven-point scale. The second half contains open questions. It will take approximately 10 minutes to fill in this form.

Scale Used:

1	2	3	4	5	6	7
Strongly Agree	Moderately Agree	Slightly Agree	Neutral	Slightly Disagree	Moderately Disagree	Strongly Disagree

Perceived Usefulness

1. Using JPF improves the quality of software I build.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

2. JPF enables me to find concurrency issues more quickly.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

3. Using JPF allows me to find more concurrency problems than would otherwise be possible.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

4. Overall, I find JPF useful in my job.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

Perceived Ease of Use

5. I find interacting with JPF easy.

1	2	3	4	5	6	7

Strongly Agree — Strongly Disagree

6. I find it easy to get JPF to do what I want it to do.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

7. Learning to operate JPF was easy for me.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

8. Overall, I find JPF easy to use.

1	2	3	4	5	6	7
Strongly Agree						Strongly Disagree

Predicted Future Use

9. I predict I will use JPF on a regular basis in the future.

1	2	3	4	5	6	7
Likely						Unlikely

JPF features

10. Why do you find JPF useful or not?

11. What needs to be changed/added to JPF to make it more useful according to you?

12. Why do you find JPF easy to use or not?

13. What needs to be changed/added to JPF to make it easier to use according to you?

General questions

I am a

- □ Man
- □ Woman

How old are you?

.....

What is your highest level of education?

- □ Bachelor
- □ Master
- □ PhD
- □ Other, namely .....

How many years of experience with model checking do you have?

.....

How many years of experience with Java do you have?

.....

What is your job title?

.....

How long have you been working with JPF?

.....

Where do you use JPF for?

.....

Do you know anyone else who might use JPF, if so, could you forward this email to him/her?

Thank you very much for filling in this questionnaire.

Appendix C. Accompanying Letter

Dear sir/madam,

I am Jantien Sessink, a student in Computer Science at the Radboud University in Nijmegen, the Netherlands. For my master thesis I am doing research on the usability of Java PathFinder. Therefore I need some information from JPF users.

I am gathering information on the usefulness and the ease of use of JPF and the features of JPF that influence these two aspects. This information will be used to evaluate which aspect of JPF could be changed in order to increase its usability. To obtain this information, I have put together a questionnaire containing various questions about this subject.

I would be very grateful if you could help me and fill in the questionnaire at the bottom of this email. It will take approximately 10 minutes to answer the questions, the information will not be used outside of this research and the questionnaire results are used anonymously. If you fill in the questionnaire, I will send you a copy of the research outcome.

Kind regards,

Jantien Sessink
Radboud University
ja.sessink@student.ru.nl

Appendix D. Reminder Letter

Dear sir/madam,

I am Jantien Sessink, a student in Computer Science at the Radboud University in Nijmegen, the Netherlands. For my master thesis I am doing research on the usability of Java PathFinder. Therefore I need some information from JPF users.

I am gathering information on the usefulness and the ease of use of JPF and the features of JPF that influence these two aspects. This information will be used to evaluate which aspect of JPF could be changed in order to increase its usability. To obtain this information, I have put together a questionnaire containing various questions about this subject.

Last week I sent you the questionnaire about the usability of JPF. If you have not filled in this survey, I would be very grateful if you would still complete it. You can fill in the questionnaire at the bottom of this email and simply press reply. It will take approximately 10 minutes to answer the questions, the information will not be used outside of this research and the questionnaire results are used anonymously. If you fill in the questionnaire, I will send you a copy of the research outcome.

Kind regards,

Jantien Sessink
Radboud University
ja.sessink@student.ru.nl