Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

David Lakatos

SUPPORTING ERROR PROPAGATION MODELING WITH JAVA PATHFINDER

Formal verification of networked Java applications

ADVISERS

Imre Kocsis

BUDAPEST, 2014

Table of Contents

HALLGATÓI NYILATKOZAT

Alulírott **Lakatos Dávid**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2014. 05. 15.

...…….

Lakatos Dávid

Összefoglaló

Az informatikai, rendszerszintű hibaterjedés-diagnosztika célja, hogy különálló szoftver- és hardverkomponensekből álló rendszerekben különböző hibaok-típusok hatásanalízisét és a hibahatások okának meghatározását lehetővé tegye. A rendszerszintű következtetés támogatásához azonban szükséges a rendszer alkotóelemeire vonatkozó hibaterjedési szabályok ismerete – egy adott komponens(típus) kimenetein milyen hibák jelen(het)nek meg külső és belső hibaokok hatására.

Komponensszintű hibaterjedési szabályok előállítására ismertek modellellenőrzési (model checking) technikák. Ezek gyakorlati alkalmazhatóságát azonban kérdésessé teszi, hogy a szokásosan alkalmazott modellellenőrző eszközök többsége komponens modellt, nem pedig rendelkezésre álló szoftvert vizsgál. Az ismert kivételek egyike a Java PathFinder, mely egy közvetlenül Java bytecode-ot vizsgáló modellellenőrző eszköz. Különlegessége, hogy a „külvilággal" (pl. állományrendszer, hálózat) való kommunikáció modellezését Java könyvtári osztályok egyfajta „lecserélésével" teszi lehetővé. A hálózatba kötött Java alkalmazások ellenőrzésére a Java PathFinderhez elérhető megoldások azonban erősen kezdetlegesek; hibaterjedési szabályok felderítésére csak korlátozásokkal alkalmazhatóak.

Célom olyan megoldás összeállítása, mellyel hibaterjedést tudok vizsgálni adatfolyam hálózatokban. A hiba komponensről komponensre terjedhet hálózati kommunikáció által, ezért a Java PathFindert fel kell készíteni hálózati forgalmat generáló, és hálózati forgalom által vezérelt működésű programok verifikációjára. A kommunikációs hálózatot használó programoknál ismertek különféle modellellenőrzési akadályok, melyeket előfeltételezésekkel, vagy más módon ki kell küszöbölni. A dolgozatban megoldási lehetőségeket vizsgálók meg, és ezeket összehasonlítom több szempont szerint.

Abstract

System diagnosis is a necessity in enormous IT infrastructures. These IT systems consist of both hardware and software components. A component's failure may cause a system-wide error. If every component's error propagation property is enlisted, engineers are able track system errors to their source by utilizing error propagation techniques in case of a system failure. Although analyzing a failure is useful, preventing it is more important. By simulating input errors, engineers are able to draw conclusions about the system-wide effects of the cause of error.

There are techniques which might be useful to create these component scoped error propagation rules. Model checking is one of these techniques. The problem with this specific type of verification is its application for real life situations. There are model checking tools which verify a specific input. These inputs usually are abstract software models, like UPPAAL [1] or Promela [2]. One of the exceptions is Java PathFinder [3]. It is a software verification tool which verifies Java bytecode instead of an abstract model. PathFinder has an extraordinary feature enabling us to verify software which communicates with the outside (network, file system, etc.). This feature is achieved by swapping the standard Java classes to altered, model checking enabled ones. Although support for applying network layer abstraction exists, analyzing error propagation by Java PathFinder is difficult and rudimentary.

My goal is to prepare Java PathFinder to support error propagation modeling. Since errors may propagate from node to node by network communication, PathFinder must be prepared to check software utilizing network communication. There are known issues considering model checking network based applications. These problems must be eliminated if possible. If not, premises are necessary. I show and compare several possible solutions in this document.

1 Introduction

Software verification is required in unique forms and at a different extent in each IT areas. One can think of an ordinary blog site which poses no threat to human life or to a large quantity of money. The blog site should not have the same type and extent of software verification as an embedded safety-critical airplane control system or an automated teller machine (ATM) network of a banking system. The latter, safety-critical and business-critical systems should be verified thoroughly in order to ensure their functional correctness.

A distributed system consists of many interacting components. A system-wide analysis should verify the components' behavior one-by-one and the system as a whole itself. One component's failure may cause a phenomenon of spreading error in a networked system, called error propagation. Verification engineers should be able to predict how a particular component's failure affects the system. This analysis requires an error propagation modeling technique which is able to enlist the possible component failures and predict their effect on the relying systems.

Model checking is usually a thorough but time consuming software verification technique. Model checking tools usually can't be easily utilized during the software development process. Tools which can be applied on the completed product exist, but they require specialized expertise. A necessary model transformation usually appears as a requirement in order to apply the formal verification tool. This transformation may be an error-prone human-performed process and may cause a radical increase in the cost and the complexity of the verification. A tool named Java PathFinder [3] [JPF] overcomes these kind of problems. There is no human-performed model transformation and its maturity now (2014) enables test and verification engineers to trust and apply the tool to verify software developed in Java. JPF is mainly engineered to find bugs involving multi-threaded applications but it has many other useful features like searching for possible uncaught exception and automatic test case generation. Formally verifying single threaded Java programs is difficult due to the huge or infinite state space caused by certain variables and objects, even if no interaction with other components is taken into account. This kind of interaction might be file operation or network communication. JPF is not able to verify applications utilizing I/O operations by default. As soon as the system under

test software [SuT] reaches for information originated from outside its Java Virtual Machine [JVM], JPF needs extensions engineered by 3rd parties to provide a proper abstraction of the outer world. Solutions usually require finite state space so they have a closed world model.

2 Model checking networked systems

Formal verification in general is usually a complex and resource demanding procedure. Most common problems originate from the nature of finite resources. One of these problems is state space explosion which may lead to high computation time and performance degradation due to lack of free memory (swapping or use of page file). Verification of a multi-threaded application often generates an exponentially growing state space with an input of the software's number of states. The yield is even bigger when networking interactions join multiple process' state space; it may lead to a Cartesian product of multiple state spaces. The case of two processes who have identical state space is illustrated on Figure 1.

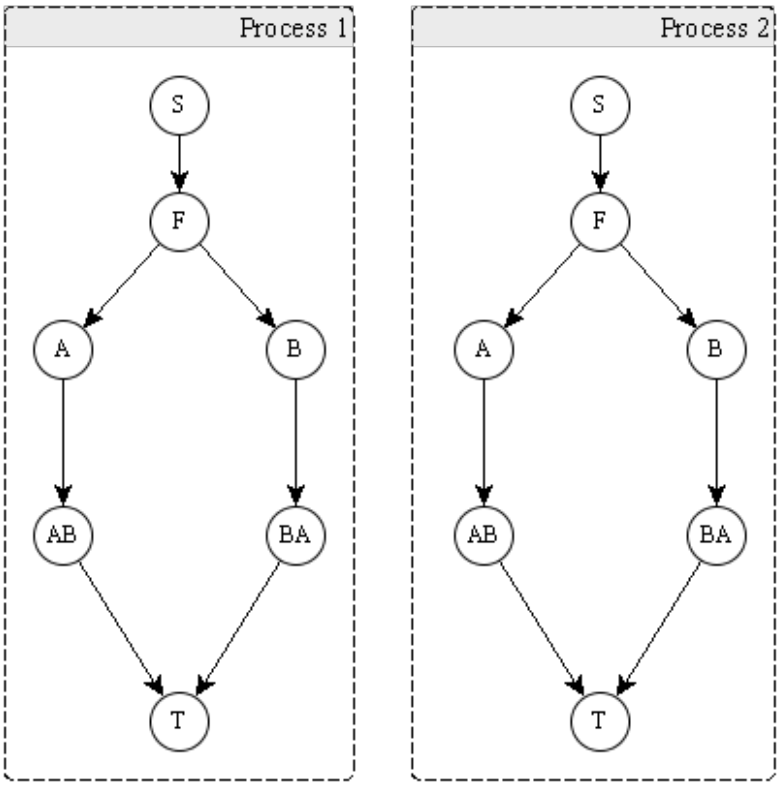

Figure 1 Two processes with identical state spaces

Cartesian product of state spaces contains every possible pair of process states as depicted on Figure 2. Each system state is constituted of a state of *Process 1* and a state of *Process 2*.

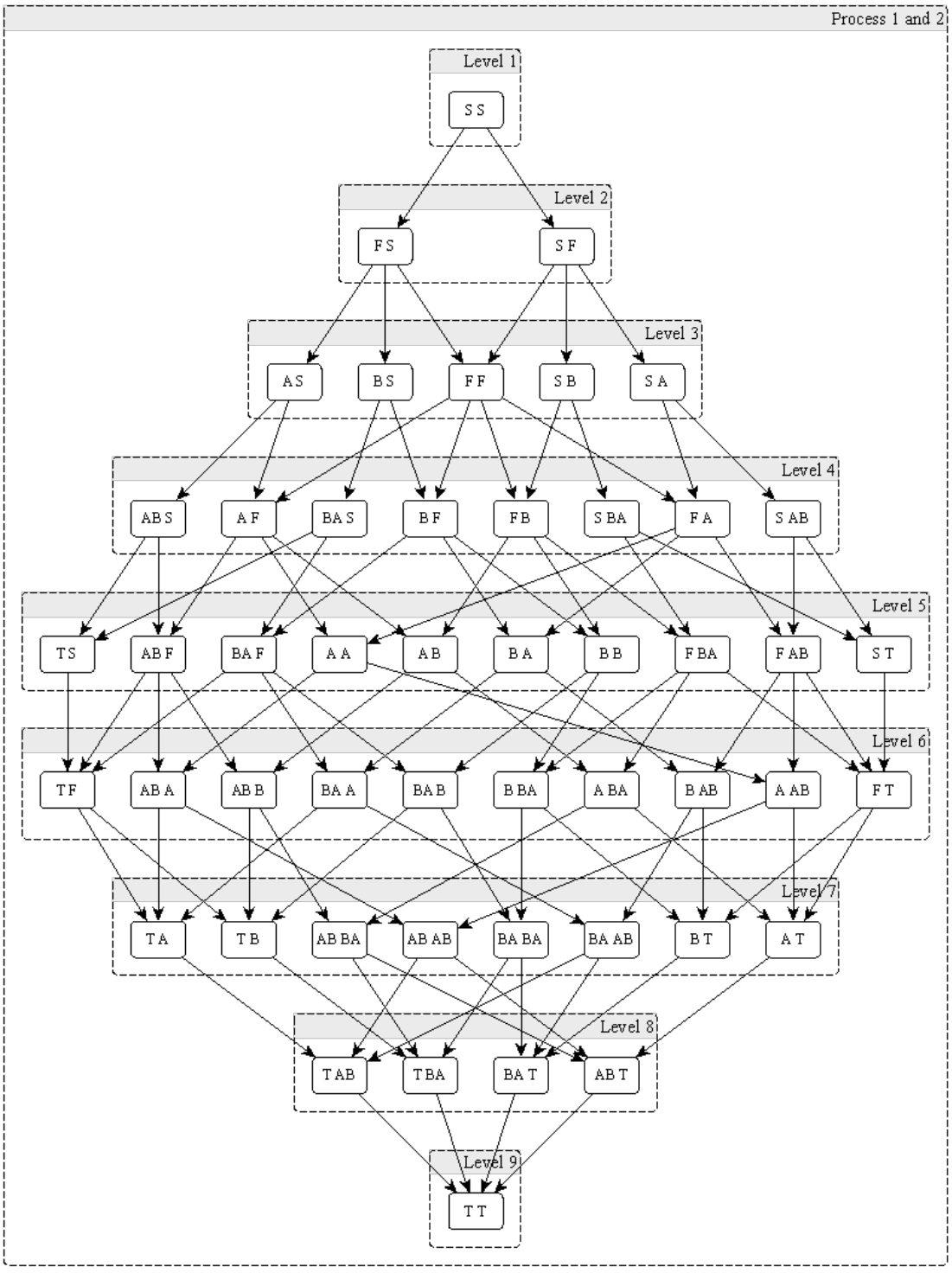

Figure 2 Cartesian product of two communicating process' state space

When considering model checking networked applications, some factors mitigate the rate of explosion by excluding unreachable system states. These unverified states can be identified by verification rules based on trivial facts (constraints).

Server applications traditionally listen on a local port waiting for a client to connect to it. Listening on a port requires a blocking method invocation. The method call

returns when connection to the port has been established by a client application. This mechanism requires server SuT to reach a verification state in which it listens on a local port before client SuT could connect to it. On the other side, a listening server thread cannot continue its execution until it received a connection on its listening port. These observations reduce the original, Cartesian product state space to a smaller one. The state space of a simplified client-server interaction is sketched on Figure 3.

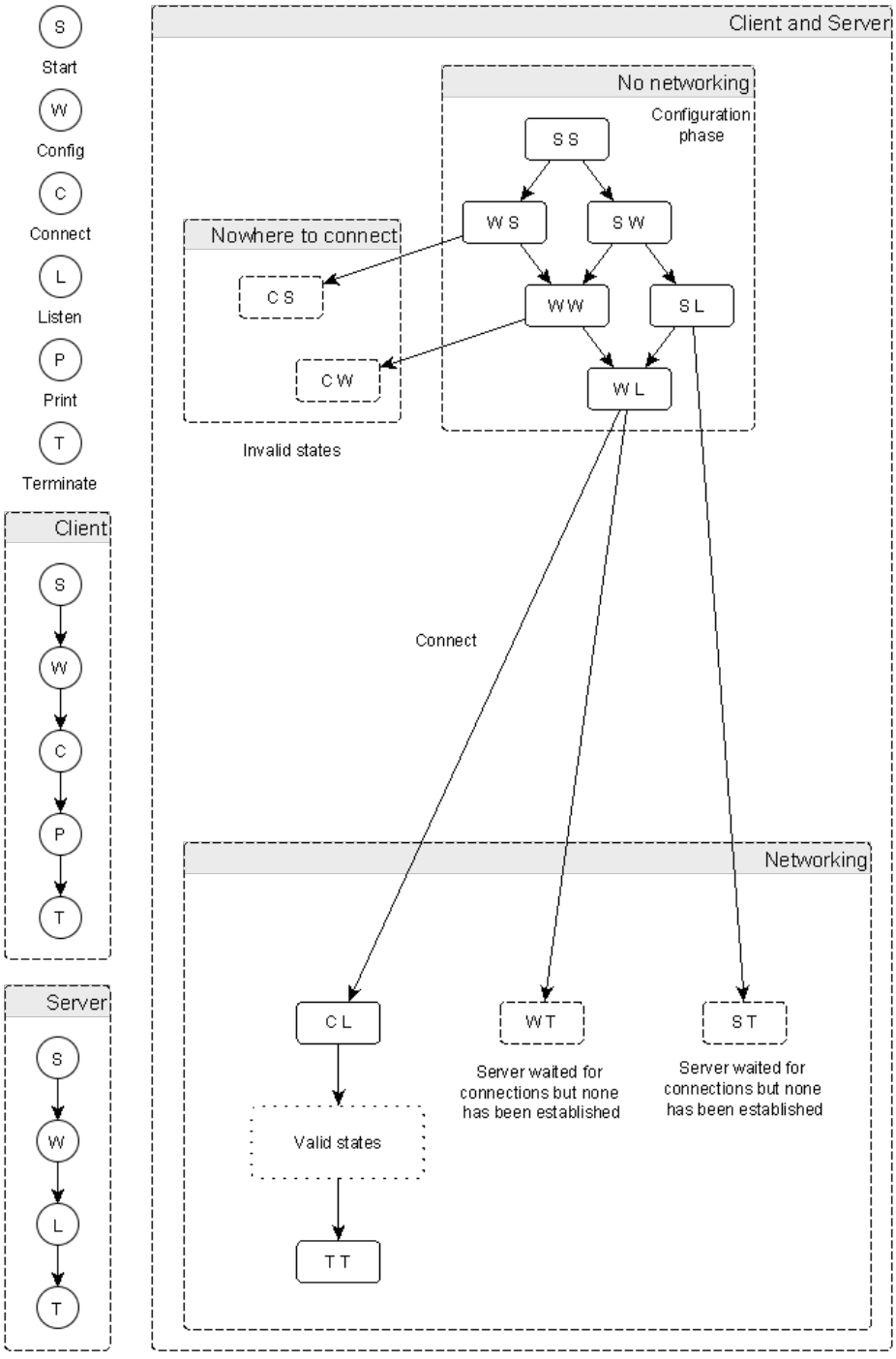

Figure 3 Model checking networked applications: state space reduction techniques

2.1 Formal verification strategies

Historically, support for explicitly modeling network communication between distributed nodes has not been a main concern of model checking. Classic model checking approaches necessitate capturing the SuT in the input language of the model checking tool. The necessary rewriting (or more commonly transformation) of system design, algorithm or source code into the input language can include the expression of the distributed nature of the problem and the characteristics of network-based communication.

In contrast to the classic approach, the main distinguishing feature of JPF is that model checking targets directly the Java bytecode, thus eliminating the step of transforming the SuT into a tool-specific representation. JPF does not require verification engineers to alter the SuT in any ways that may unintentionally alter its behavior. This concept minimalizes the possibility of performing a non-representative verification due to faulty model transformations and potentially reduces the cost and increases the speed of the whole process as well.

However, the core JPF technology was designed to verify the dynamic behavior of a single JVM process under a variant of the closed world assumption; the JVM is not allowed to communicate with its environment disallowing file I/O and network communication. When communication can or should not be abstracted away in order to perform representative verification, classes that perform such communication can be substituted with so-called model classes. These classes can simulate communication with the environment while providing facilities necessary for model checking (e.g. backtracking and replaying read operations from a specific file).

Due to JPF's lack of networking support, model checking networked applications with it is not a straightforward (and noninvasive) procedure currently. Still, bytecode verification is not fundamentally incompatible with verifying networked systems; as we will see, this current limitation only means that the feature set of JPF has to be extended.

Verification engineers in need of a formal verification tool which supports model checking networked applications may choose from the two fundamental approaches.

2.1.1 Classic centralized model checking

The main challenge in performing networked software model checking is to implement a solution which abstracts away the network communication in order to manipulate the state space exploration of the system as a whole while leaving execution logic intact as much as possible. Centralized model checking is a formal verification technique which simplifies the problem of network abstraction by applying a model transformation on the SuT. The technique unites multiple, communicating processes into one process. On a technical level, SuT (operating system) processes are transformed into threads; network communication is often transformed into function calls. After the transformation, software verification targets a multi-threaded, single process application that does not use networking. Using this pattern, PathFinder needs no modification in order to perform the system's model checking [4].

This approach requires model transformation which radically increases the probability of faulty verification due to an incorrect model transformation step. Moreover, the concept loses the strength of PathFinder's bytecode verification feature which would make it unnecessary to modify the SuT source code.

2.1.2 Network communication handled by JPF

Due to centralized model checking's disadvantages described at the end of section 2.1.1, model checking networked applications necessitate a more elegant and less invasive network abstraction which requires no SuT model transformation. PathFinder offers verification engineers an application programming interface [API] to implement JPF's network abstraction through 3rd party JPF modules [5].

2.2 Primary goal

The following chapter contains certain dependability engineering related technical terms which have to be agreed on.

- This document regards a JVM-confined Java process as an independent component. It manages its own memory address range and schedules its threads.

- An aggregation of independent components constitute an autonomous component. Autonomous components are able to mitigate their internal errors.
- This document uses the terms of *fault*, *error* and *failure* according to the Laprie dependability engineering taxonomy [6].
- Failure classification follows the Bondavalli-Simoncini [7] taxonomy.

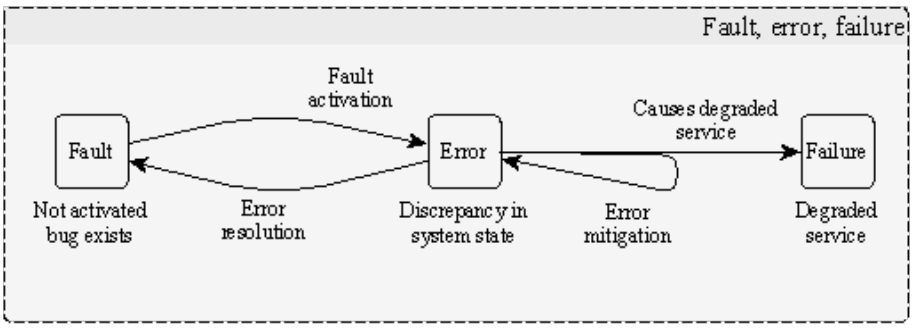

Figure 4 Laprie's fault, error, and failure

A complex Java system consists of interacting autonomous components. When one component fails other relying component(s) might suffer the consequences. We speak of error propagation when one autonomous component's (the one on the left on Figure 5) internal state is compromised (error) due to a fault activation and this state acts as another component's fault activation. Eventually, the latter component is driven to a state of error too. On the other hand, an error mitigation occurs when the latter autonomous component's fault activation does cause a compromised state of error but none of the relying components are vulnerable to this situation. The error's propagation ends at the unaffected component.

A verification engineer have to answer the following questions during performing an error propagation analysis:

- What kind of autonomous components exist in the system?
- What kind of propagated errors are we looking for?
- How does the directed data flow graph look like in the system?

If all these questions are answered the verification process may continue at the level of autonomous components. As the verification engineer explored, there might be

possible paths of error propagation due to functional dependencies between the autonomous components. Possible error propagation paths are sequences of autonomous components constituted by client-server pairs. Section 4.2.2 describes an algorithm which can be applied on client-server pairs in order to explore possible failures emitted to the outside world in the form of miscomputed data or as a loss of service.

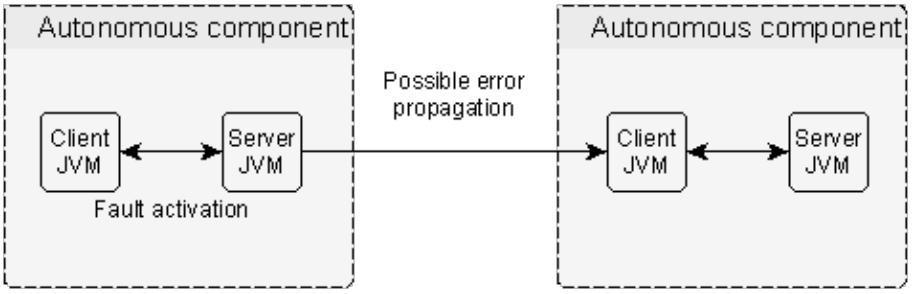

Figure 5 Possible error propagation

The verification engineer should apply a tool that enlists the possible emitted errors from an autonomous component and they should match this type of error with the relying components. An error propagation occurs only when the autonomous component on the right is vulnerable to the errors emitted by the one on the left. Engineers should be able to identify error types possibly emitted from autonomous components.

2.3 Existing tools

As described in section 2.1.2, JPF provides an API which enables PathFinder's feature extension through. JPF modules may apply network abstraction in order to alter PathFinder's verification behavior. This section describes two existing JPF modules which enable PathFinder to verify networked applications.

2.3.1 Jpf-net-iocache

The extension intends to overcome JPF's lack of networking support by applying a network abstraction layer between the client and the server. Watcharin Leungwattanakit [8] started the project as part of his thesis [9] with the assistance of Cyrille Artho [10]. The authors of `net-iocache` [11] created the most commonly accepted solution which enables JPF to verify networked applications. The project is still under rapid development. It enables engineers to perform client software verification.

`Net-iocache` forms a cache layer between the client SuT and the server SuT. Caching mechanism performs the networking abstraction through model classes (described later in 3.3.1) which substitute classes of the JVM's `java.net` package. The cache layer is a message mediator which forwards requests to the server SuT and their responses back to the client SuT. This mediator is able to observe all message exchange and perform message based analysis.

Furthermore, caching mechanism improves the performance of the verification. The increase in performance originates from the speed difference between network communication and memory access. The module caches responses from the server to forwarded requests as depicted on Figure 6.

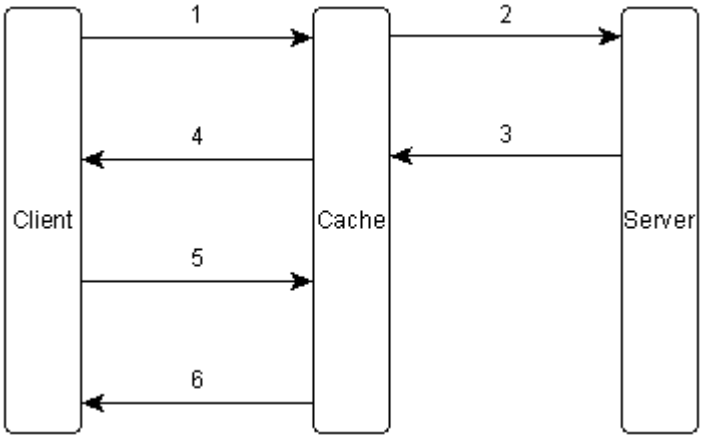

Figure 6 Accelerating verification by network caching mechanism

The solution has its limits regarding the bug types it is able to find. The search could only be performed on client applications at the time the research of this thesis started. `Net-iocache` starts the server as a simple Java process out of JPF's scope. While JPF explores the client application's state space the cache layer proxies the requests to the server process and caches the replies on the way back. Although it provides high-performance client side verification, the concept does not support model checking of server applications.

The cache layer solution does not support the verification of client-server interactions. A simple interaction related bug might be the one depicted on the figure below.

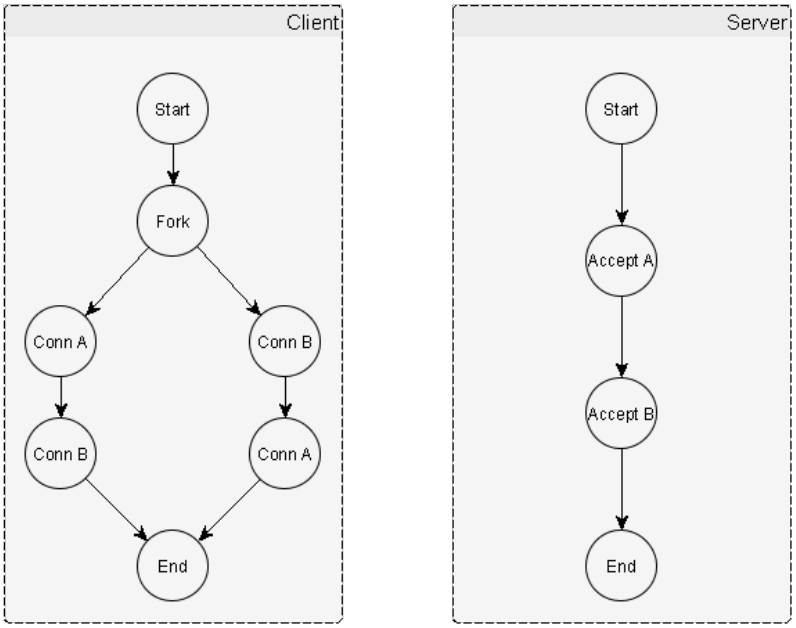

Figure 7 Networking interaction example

Figure 7 depicts the state space of an interacting pair of client and server, where the client starts two threads. One of them tries to connect to the server's port *A*, the other one tries to connect to port *B*. The JVM's thread scheduling decides whether the application will connect to port *A* or *B* first, and to the other later. The server sequentially accepts connections on port *A* and *B*.

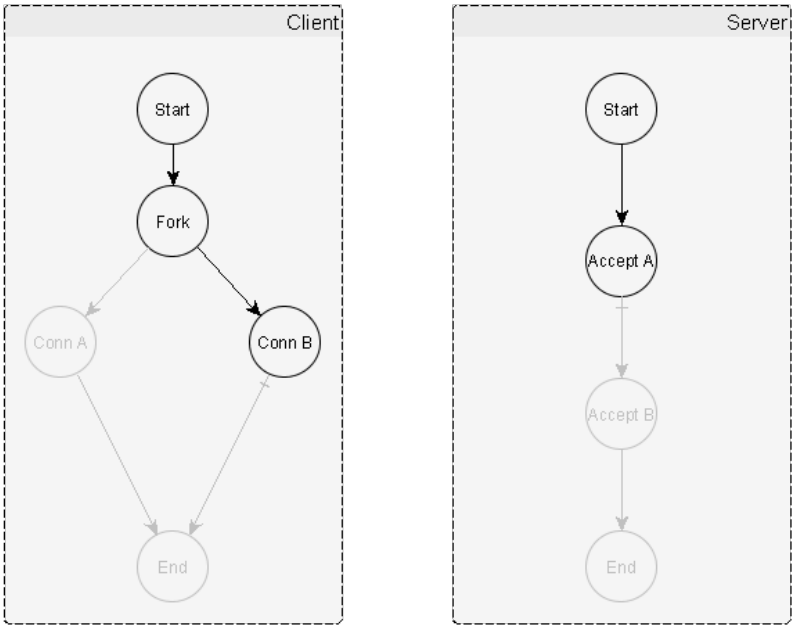

Figure 8 Networking interaction problem

As Figure 8 shows, there is a thread scheduling scenario which causes malfunction in the system. The client might connect to port *B* first which causes deadlock (`java.net.ConnectException` thrown after timeout). Distributed errors such as the possibility of system-wide deadlock should be noticed by formal verification. However, the recognition of deadlocks in such situations is not achievable with the assistance of `net-iocache`.

2.3.2 Jpf-net-master-slave

Sergio Feo started the development of a still unpublished JPF module [12] during his PhD studies at Albert-Ludwigs-Universität Freiburg. Mr. Feo's verification strategy seems to be very similar to the one described in chapter 4.2.2 but it remained unknown for the JPF community. The project was started during Mr. Feo's internship to National Institute of Informatics (Japan) [13] in 2013 but never published the tool nor prepared its documentation. He was assisted by Prof. Yoshinori Tanabe [14] who supported the development of `jpf-net-iocache` as well. Mr. Feo was kind enough to provide access to the project's source code which confirmed his verification technique's similarity to the one this document defines in section 4.2.2.

Both of the similar solutions perform the verification of networked applications based on trails of exchanged messages. Message trails provide interaction causality management during the verification process. This feature allows them to examine the system's behavior in detail which leads to a more accurate verification strategy compared to `net-iocache`.

3 Model checking Java bytecode with Java PathFinder

NASA scientists felt the necessity of a model checking tool to verify the agency's increasingly complex and mission-critical software systems. To reach this goal the development of Java PathFinder was started in 1999 as a Java-to-Promela translator without any model checking capabilities. Around 2000 its main objective changed to being a standalone model checker. It became open source in 2005 and has been a fascinating area of academic research ever since [15] [16] [17] [9]. JPF has proven its usefulness during the verification of the K9 Mars rover's control unit [18], the EO-1 spacecraft's diagnostic system and even at the testing of Fujitsu's Web applications [19] as well.

3.1 Architecture

The most recent version of JPF is a single threaded Java 7 SE software. JPF runs in a standard JVM and simulates the functionality of a full JVM when checking applications. The following figure depicts how PathFinder controls the SuT, and how it is able to become its internal state observer during verification.

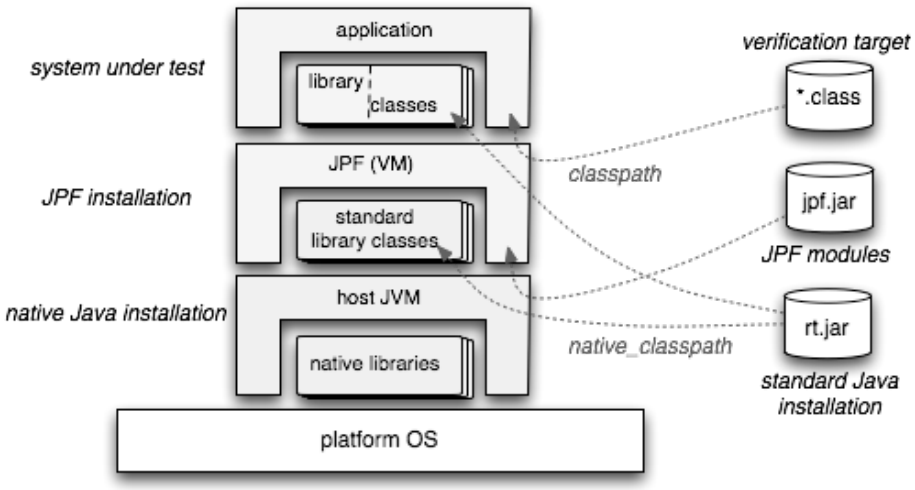

Figure 9 JPF stack [20]

JPF acts like a JVM but implements the Java Virtual Machine specification [21] in a unique way in order to become the SuT's perfect observer [7]. The SuT bytecode is executed in a different way as it would be done by a normal JVM. The execution of the SuT is similar as if it was modeled through the Java Reflection API [22]. JPF starts with

reading the bytecode of the SuT and builds up an execution environment for the verification. Each bytecode operation (defined in [21], *Chapter 6: The Java Virtual Machine Instruction Set*) is modeled by a bytecode class. When JPF "executes" a bytecode instruction, the `execute()` function of the corresponding class is called. The function simulates the effects of bytecode execution by modifying the JPF virtual machine's state. For example, an opcode with the mnemonic `ALOAD` pushes a Java reference onto the JPF stack, where the stack itself is a JPF provided Java class' instance managed by the host JVM.

3.1.1 PathFinder modules

JPF has a modular architecture which makes the core project extendable. PathFinder enables developers to override its behavior by adding their own JPF modules to it. `Jpf-net-iocache` and `jpf-net-master-slave` are JPF modules which can be plugged into the core project, into `jpf-core`. There are several ways to manipulate JPF by a module; some of the ways are described in Chapter 0 in detail.

Since JPF became open source a significant community has formed around it with active developers and end-users from academia as well as industry. The extensibility have put PathFinder in focus of progressive research over the years.

3.2 Choice generation

The execution continues through a complex choice generation mechanism which enables JPF to explore all possible thread scheduling scenarios which may happen during real-life execution. The choice generation mechanism enables verification engineers to explore all the possible scheduling and input data scenarios. These choices are made by a series of thread choice generators and data choice generators.

3.2.1 Data choice

This feature of dealing with data non-determinism makes PathFinder able to verify software which utilize some kind of non-deterministic data source. The example below [23] depicts a simple source code whose execution depends on a non-deterministic data source, in this case the class `Random`.

```
import java.util.Random;

public class Rand {
  public static void main(String[] args) {
    Random random = new Random(42); // (1)

    int a = random.nextInt(2); // (2)
    System.out.println("a=" + a);

    // ... lots of code here

    int b = random.nextInt(3); // (3)
    System.out.println("  b=" + b);

    int c = a / (b + a - 2); // (4)
    System.out.println("    c=" + c);
  }
}
```

The execution shows verification error explained by an error message. Error occurs when variables have the following value.

```
java.lang.ArithmeticException
divide by zero
a + b == 2
```

JPF explores the state space through data choice generators and shows a counterexample [23] where the property `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty` is violated. The verification stops as soon as a property is violated (configurable behavior), and the trace violating the specification is shown to the engineer.

```
> bin/jpf +cg.enumerate_random=true Rand
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
====================================================== system under test
application: /Users/pcmehlitz/tmp/Rand.java

======================================================  search  started:
5/23/07 11:49 PM
a=0
  b=0
    c=0
  b=1
    c=0
  b=2

====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
        at Rand.main(Rand.java:15)
....
>
```

In the example variable *a* (and *b* as well) has a random value due to the following line of Java code.

```
int a = random.nextInt(2);
```

The Java code compiles to the following bytecode.

```
ALOAD 1         #push reference to stack from var. 1 (random)
ICONST_2        #push the constant integer 2 to stack
INVOKEVIRTUAL java/util/Random.nextInt (I)I
                #JNI call to the JVM with one integer argument
                #the return type is integer as well
ISTORE 2        #pop the random integer result to variable 2 (a)
```

PathFinder captures the JNI call, realizes the necessity of data choice, and creates a data choice generator. The choice generator is created according to the required data choice. In this example the first random call returns with an integer 0 or 1. The choice generator is created with two possible choices. As a result the choice generator branches the state space into two separate sub-trees. In the first branch variable *a* has a value of 0 while in the other branch it has a value of 1.

As described above, data choices branch state space when the software tries to read from a non-deterministic data source. This way of exploring the state space is only sufficient when the SuT is a single-threaded software.

3.2.2 Thread scheduling choice

JPF is an efficient tool looking for concurrency related problems in multi-threaded software. It is mainly designed to detect deadlocks and race conditions. The functionality of finding concurrency bugs requires a technique which enables PathFinder to explore all the thread scheduling scenarios.

Thread scheduling choices are made when multiple threads are able to be scheduled for execution. Each choice is represented by a thread choice generator which enumerates these threads of `RUNNABLE` [24] state. During state space exploration, PathFinder picks the thread chosen by the choice generator and executes the bytecode pointed by the thread's program counter. As the execution advances one `RUNNABLE` thread's state may change into non-runnable and another thread's non-runnable state may alter to `RUNNABLE` as effect of synchronization or event driven operations. For example, change in a `RUNNABLE` thread's state may be a cause of the bytecode it executed like entering a Java monitor by calling `wait()` on a synchronization object. A non-runnable thread may become `RUNNABLE` again due to an external event. An event like this might be a `notify()` call issued by another thread on the same synchronization object which would force the blocked thread to exit the object's Java monitor and become `RUNNABLE` again.

Executing a bytecode instruction is a linear transformation between JVM states. States are identified by a complex hashing mechanism. Different thread scheduling scenarios may lead to the same system state. When the destination state is already visited (its hash value is already in the visited states' hash set) PathFinder backtracks the state space exploration.

3.2.2.1 Partial order reduction

Most formal verification techniques have a common state space explosion problem which has to be dealt with. PathFinder has a way to ease this issue. As JPF makes thread scheduling choices while generating the state space, many concurrency irrelevant scheduling scenarios are generated along the interesting ones. In order to reduce the rate of the state space explosion, a technique called partial order reduction [POR] is utilized.

PathFinder provides this feature on-the-fly. When POR is enabled by the JPF configuration, state space exploration collapses a thread's concurrency irrelevant instructions into an atomic transition. By doing so, JPF excludes states from the state space by eliminating concurrency-irrelevant thread interleaving. POR reduces the state

space by 70% according to official calculations based on an average multi-threaded scenario [25]. Figure 10 depicts the extent of state space reduction due to applying POR during the verification process.

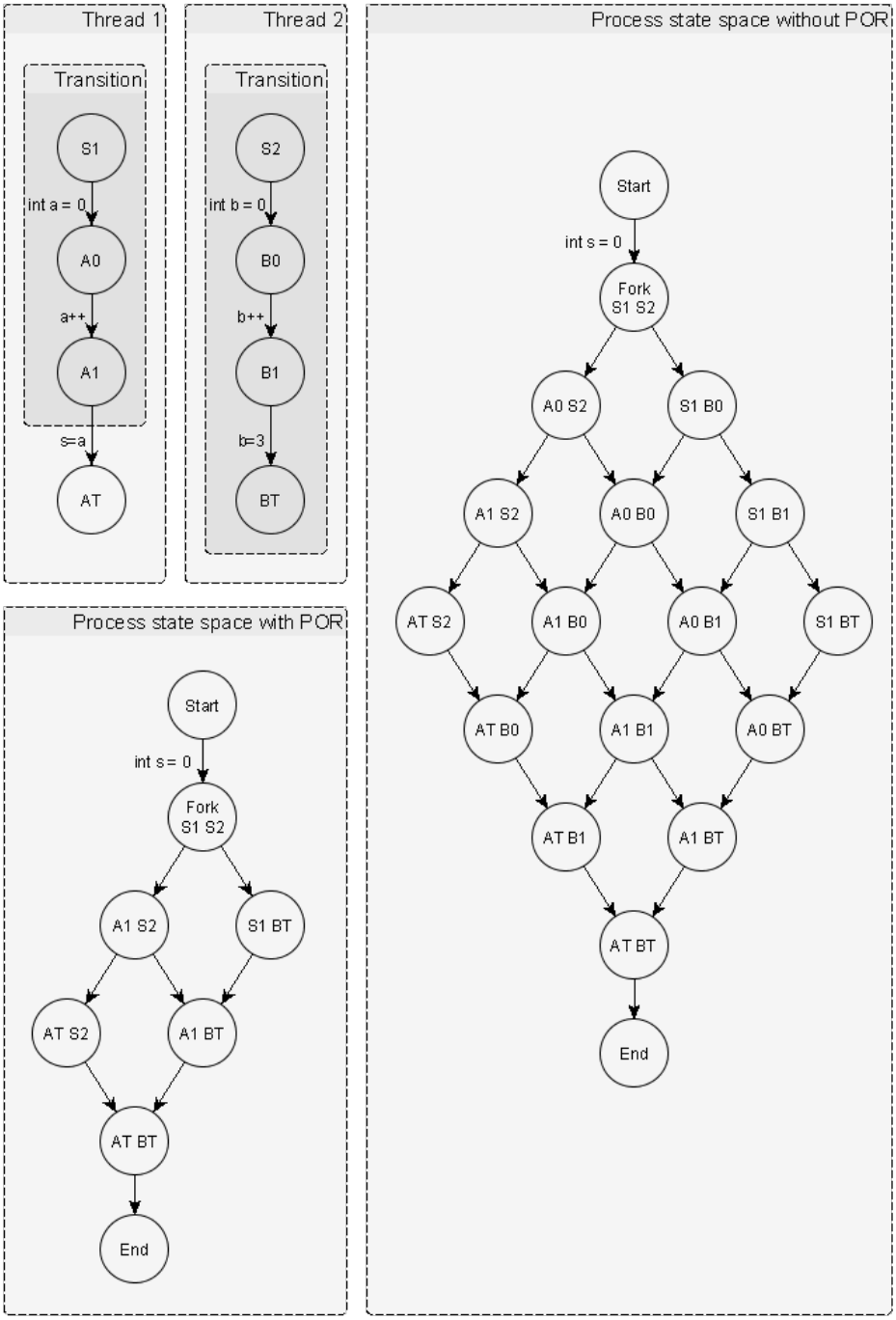

Figure 10 Difference between state space sizes with and without partial order reduction

POR is essential to perform formal verification on multi-threaded software, but the technique is not specific to JPF. The topic has extensive academic research background.

3.2.3 Choice generation effects on state space exploration

State space exploration is structured by the choice generation mechanism. Choice generators are responsible for generating a well-structured sequence of possible choices by iterating through them. Since there are situations where the number of possible choices is too high for viable formal verification (e.g. a random integer) certain heuristics must be applied. Because of these data choice heuristics, state space exploration usually cannot be perfectly exhaustive, but it can be as relevant as possible by fine-grained choice generator tuning. `Jpf-core` delivered heuristic data choice generators have parameter driven behavior. They can be configured through PathFinder's standard configuration mechanism.

3.3 Model Java Interface

A standard Java Virtual Machine provides an interface called Java Native Interface [JNI] [26] which enables virtual machine interaction with non-Java software. JVMs have platform dependent (native) modules which implement low-level operations These platform dependent modules usually are written in C and built for the processor architectures and operating systems the specific JVM version is intended to run on case by case.

JPF has an API similar to the JNI called *Model Java Interface* [MJI] [27]. It enables verification engineers to intercept native calls, apply any kind of "outer world abstraction" they intend to use and create choice generators (see Figure 11) according to the intercepted method call.

```
@MJI
  public int nextInt____I (MJIEnv env, int objRef){
    if (enumerateRandom){
      return JPF_gov_nasa_jpf_vm_Verify.getIntFromList(env,
defaultIntSet);

    } else {
      restoreRandomState(env, objRef, delegatee);
      int ret = delegatee.nextInt();
      storeRandomState(env, objRef, delegatee);
      return ret;
    }
}
```

Figure 11 Choice generation by utilizing JPF's Verify API

The source code above shows an example of choice generator insertion from the class `java.util.Random`'s `JPF_java_util_Random` native peer. The method call `JPF_gov_nasa_jpf_vm_Verify.getIntFromList` provides a set of integers to the Verify API if the verification of random data choices are enabled by PathFinder's configuration (`enumerateRandom`). JPF creates a data choice generator in its current verification state with the set of possible integer choices. If random data choice generation is disabled, then no data choice generator is created and a random integer is returned to the SuT.

3.3.1 Model class

Java software usually utilize classes provided by the Java Runtime Environment [JRE] like `java.util.Random`. When a non-JVM class is loaded which has a reference to a JVM provided class the classloader loads both of the classes. The JVM has a file located at `lib/rt.jar` which provides the necessary bytecode for further execution.

`ClassLoader` [28] is an abstract class defined by the The Java® Virtual Machine Specification [21]. The purpose of a classloader is loading Java class bytecode as required into the computer's operational memory. A JVM may have multiple classloaders structured into a classloader hierarchy. Hierarchy elements may have class loading realms. When class bytecode loading is required to continue process execution, the class' realm classloader is invoked to perform the task (according to its own implementation).

JPF has its own classloader and class loading mechanism in its JVM. It loads the bytecode similarly to a normal JVM but implements an abstraction mechanism to loading JVM provided classes. Verification engineers are able to override JVM provided classes with their own by creating model classes. For example, in order to override `java.util.Random` provided by `lib/rt.jar`, a class with the same name must be created and placed under the JPF project's classpath variable. The classloader looks up class names on the project's classpath first. If the class in not overridden, and the lookup was unsuccessful, then lookup continues on the classpath of the host JVM (see Figure 9).

Model classes are not executed by the JVM but they define how the JVM resources should work. If the SuT utilizes a modeled JVM resource (e.g. `java.io.InputStream`), the model class' bytecode is to be loaded instead of the JVM provided one. This mechanism allows verification engineers to apply any kind of JVM abstraction they intend to.

The necessity of model class invoked code execution might arise during state space exploration. For example, if the developer wants JPF to tunnel messages written by `java.io.OutputStream` to another Java process, they should override `java.io.OutputStream`'s write operation and call a custom native peer (described later in section 3.3.2) from the method body. The native peer will be invoked by JPF with the parameters provided by the model class' write method, and the native peer method will be executed by the host JVM.

The native call is matched to a native peer class according to MJI name mangling [29] mentioned in the beginning of section 3.3.2. The discussion of the example native invocation continues in the referenced section.

3.3.2 Native peer

Native calls issued by model classes are intercepted on a lower abstraction level. The interceptors are Java classes which follow a special naming convention called MJI name mangling [29] similar to the JNI name mangling [30]. These classes are referred to as native peers. The native keyword refers to their host JVM executed nature.

Section 3.2.1 showed why data choice generators are necessary in order to explore the SuT state space but did not mention how choice generators are created. JPF provides a model class named `java.util.Random` that substitutes the JVM provided one. Let us take a look at the bytecode mentioned in section 3.2.1.

```
INVOKEVIRTUAL java/util/Random.nextInt (I)I
```

According to JPF's name mangling convention the execution may be intercepted by a class named `JPF_java_util_Random`. A native peer must extend the class `gov.nasa.jpf.vm.NativePeer` and must be located in the JPF project's `native_classpath` variable. If both conditions are met JPF locates the native peer class as if the native method is called by using the Java Reflection API. The bytecode above causes JPF to invoke the method in the native peer with the following prototype:

```
int nextInt__I__I(MJIEnv env, int objRef, int n)
```

The method branches the state space by creating a data choice generator which will enumerate all possible choices in order to explore all possible executions:

```
…
IntChoiceGenerator cg = new IntIntervalGenerator("verifyGetInt(II)",
min,max);
return registerChoiceGenerator(env,ss,ti,cg,0);
…
```

Another example may be observed if the reader takes a look back at the code of the model class `java.io.OutputStream` discussed in section 3.3.1, its invocation continues at the following native peer class' method:

```
public class JPF_java_io_OutputStream extends NativePeer {
@MJI
  …
  public int native_write___3BIII__I(MJIEnv env, int objRef, int
messageRef, int offset, int length) throws IOException {
      …
  }
  …
}
```

The method may implement the verification engineer's desired functionality which will be executed whenever the SuT calls `java.io.OutputStream:write()` during the state space exploration. For example, a sought functionality might be the application of a network communication abstraction; transparently redirecting the SuT traffic to a custom location and perform required verification steps before or after the re-routed transmission.

3.4 Listener

PathFinder provides an event handling mechanism which enables developers to subscribe to certain events. Event handlers are able to read or modify JPF's internal verification state. Listeners are separated into two groups regarding their event type subscription.

3.4.1 Search listener

When a verification search is ongoing PathFinder notifies search listeners about the verification algorithm's internal events. A listener which is to subscribe for search events, ought to implement the `SearchListener` interface. Search listener is usually utilized to perform verification initialization, show error trace at the end of verification and trigger on-the-fly verification analysis.

3.4.2 VM listener

VM listeners are similar to search listeners. The difference between them is that they enable JPF modules to subscribe to `jpf-core`'s JVM events instead of search related ones. VM listener may handle choice generator, concurrency detection, bytecode execution, and thread scheduling related events declared by the interface. VM listeners must implement the interface `VMListener`.

3.5 Search object

Verification strategies are encapsulated by search objects. Each of them extends the abstract class `Search`. Verification algorithm ought to be implemented in the overridden abstract method `search()`.

As soon as JPF initialized its JVM it instantiates a search object based on the configuration. Verification strategy execution starts by calling the `search()` method. During this document's preliminary research an undocumented limitation has been identified and it has been confirmed by Peter Mehlitz [31] (lead developer of Java PathFinder). Nested search performed by multiple search objects is not supported yet due to JVM implementation limitations.

3.6 Bytecode factory

As PathFinder prepares to perform verification, SuT class files are scanned for bytecode. Each opcode type is modeled by JPF. PathFinder instantiates these classes with certain parameters (bytecode arguments) if the equivalent opcode has been recognized as the next one in the class file. Bytecode instances are linked sequentially as if they were located in a class file, so it preserves the original bytecode sequence.

Bytecode factory is responsible for the bytecode instantiation. Developers are able to override bytecode factories with their own by JPF configuration. Hereby they can implement a custom bytecode instance creation logic.

3.7 JPF attribute

JPF objects such as bytecode instances have a special variable called JPF attribute. Attributes enable module developers to extend JPF objects with custom information. This is a generic field to allow for maximum flexibility w.r.t. enabling custom object annotations (specifically, `java.lang.Object`).

4 Transparent client-server co-verification

Need of a networking enabled model checking tool emerges when a verification engineer intends to perform error propagation modeling. Since the engineer's main objective is to accomplish a viable formal verification as accurate as possible, a correct verification result providing network abstraction is required in the process.

4.1 Approach

Any abstraction layer may cause unwanted effects during formal verification. The more abstraction is in place the more verification differs from real life situations. The goal is to simulate the client-server interactions as they would happen during normal execution involving minimal verification abstraction. The solution should not utilize any unnecessary verification assumption about the client and the server software.

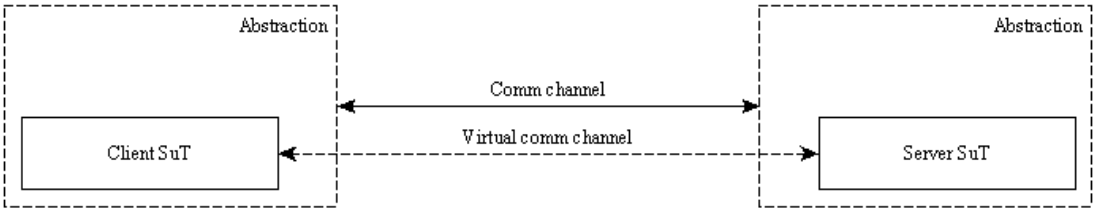

Figure 12 Applying networking verification abstraction between client and server

4.1.1 Interaction driven verification

A Java process which performs networking operations usually has an infinite state space which obviously is not formally verifiable in an exhaustive way. The proper abstraction logic may transform this state space to a finite form introducing a verification assumption. This document assumes that the SuT, in this case a client-server pair as one has no outgoing network communication outside the SuT. This presumption is necessary in order to avoid the necessity of an outer World abstraction mechanism which would possibly alter the correctness of the verification; this presumption causes the verification to be based on closed World assumption.

This thesis documents a JPF module called `jpf-jointstates` [32] [`jointstates`] which is an open-source software product developed during the research by David Lakatos. The source code is downloadable to the public from Bitbucket's [33] Mercurial [34] repository [32]. `Jointstates` is able to observe causality relations between

networking interactions. The figure below shows an example to a classic causality relationship between messages *A* and *B*.

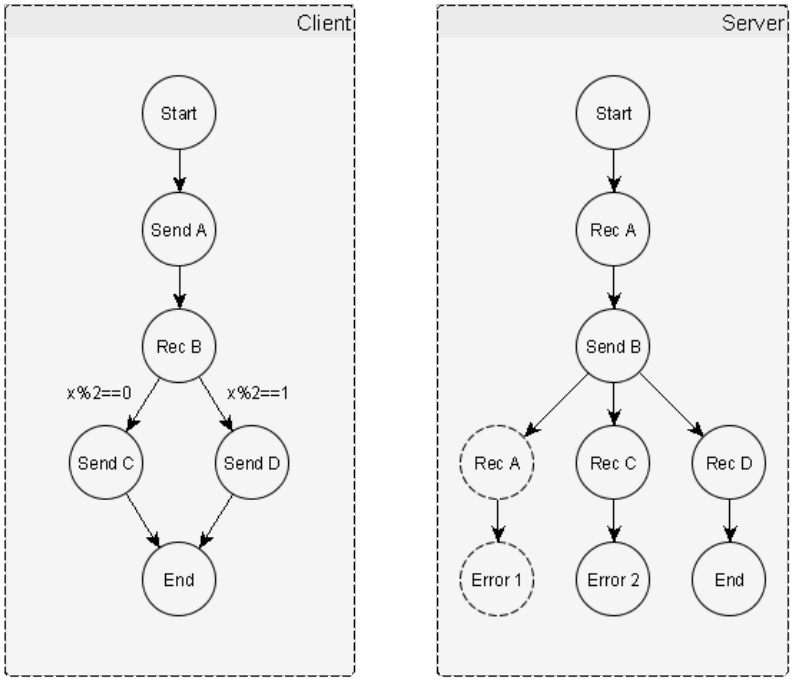

Figure 13 Example of interaction causality management

Variable *x* on the client side represents a random integer typed value like `System.currentTimeMillis()` function's return value. Client starts the interaction by sending message A to the server and the server replies B. A wrong network abstraction logic would be to exclude message causality relationship from the verification by allowing JPF to send message *A* to the server at this point. As one can observe this message leads to a state which would never be observed during real execution because of the message causality Send(A)→Rec(B)→Send(C|D) on the client side. PathFinder would detect an error which would never occur normally. Or better to say, would never occur if the closed world model that we have imposed on the server by defining its client holds. The client can model realistic usage scenarios to the extent the engineer wishes, up to and including "fuzzing". The approach can be extended for multiple clients too, but many interesting cases can be covered by a single, multi-threaded client. The main point is that the verification engineer is in full control w.r.t. modeling the environment of the server. This way, proper handling of message causality eliminates the verification of unrealistic system states. Another example of the need of proper network abstraction is depicted on the figure below.

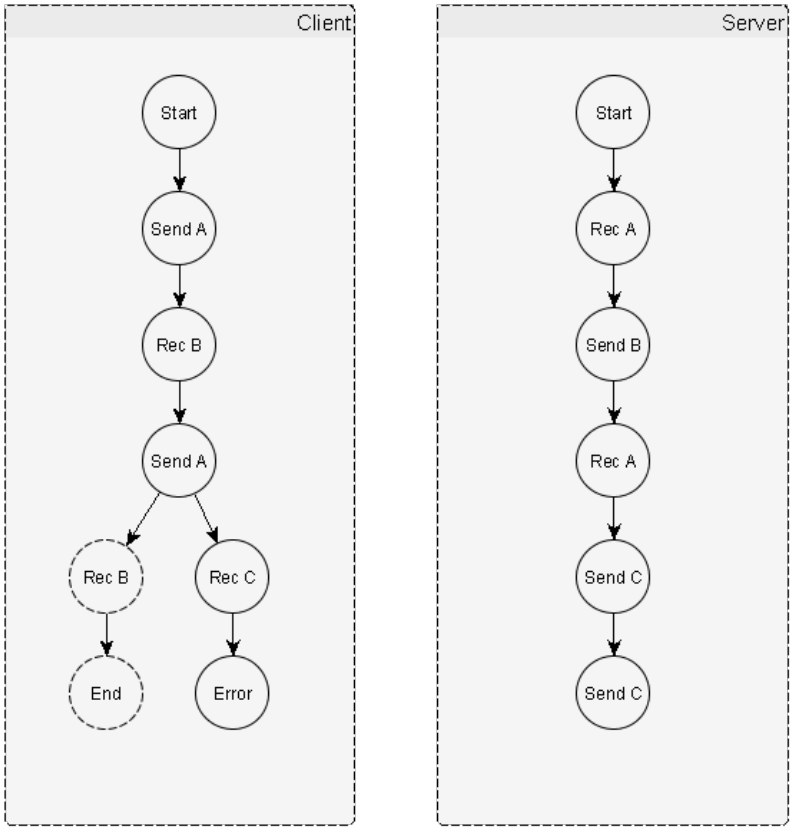

Figure 14 Interaction causality on the server side

The client sends message *A* and awaits a response *B*. The server does not return to the state before receiving message *A* (e.g. increments a variable). The client sends the message *A* again to the server. `Jpf-net-iocache` would return the response B to the client because of the caching mechanism but the server actually responds *C*. `Net-iocache` would falsely finish the verification showing no sign of any error. `Jpf-jointstates` finds this problem by maintaining the causality relationship between transferred messages and linking them to the client's and server's verification state.

4.2 Architecture

Providing a transparent networked solution with PathFinder is a challenging task with numerous implementation difficulties. At the research's early phase, the modification of PathFinder's internal state storage technique was attempted unsuccessfully. The failed concept was based on a JPF instance which explores two networked process' state space simultaneously. PathFinder's hard-wired implementation prevented the idea from succeeding, but it inspired the concept of a new architecture.

4.2.1 The big picture

The simplified verification topology is depicted on the figure below.

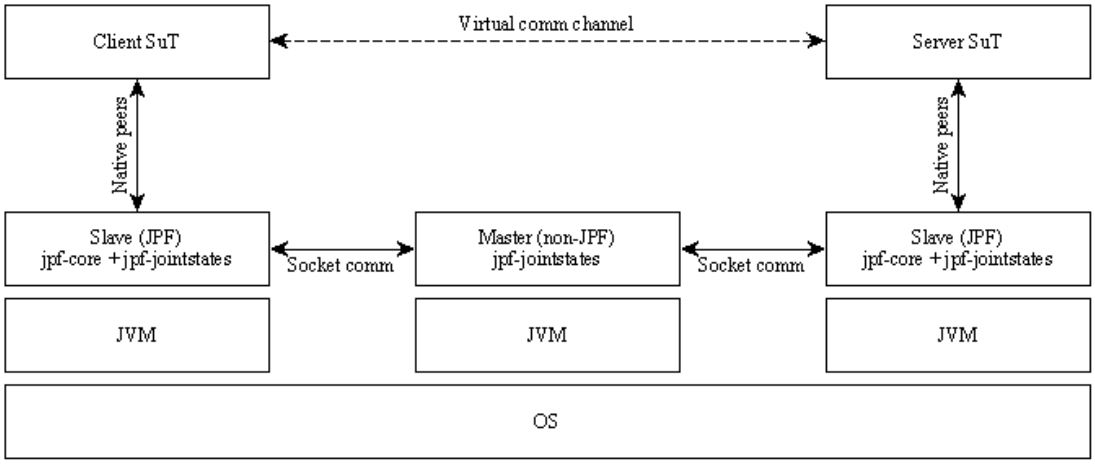

Figure 15 Jointstates architecture

Project `jpf-jointstates` acts as a JPF module which provides the functionality to alter PathFinder's default behavior as required (slaves). `Jointstates` provides a standalone Java software as well which acts as the coordinator and data aggregator of the distributed verification (master).

4.2.2 Verification algorithm

The solution aims to verify networked software solutions simulating the interactions as realistically as possible; simulating real execution by state space exploration. `Jointstates` should explore states which might exist during an execution but should not cover unreachable ones. The verification should not explore states which are unreachable by the SuT logic; the verification should be "representative" in this specific sense. Representative verification suggests the need to handle causality relations between exchanged messages.

Message causality may be represented by a directed graph where message *B* and *C* might follow message *A* (as depicted below). Uncertainty of the message type following message *A* is caused by the sender software's data choices (in JPF terminology). The source of uncertainty may be e.g. a non-deterministic data source which has an effect on execution control. Obviously, the receiver has no control over the type of message the sender transmits. Pseudo code below shows a control logic that has the pseudo state space as depicted.

```
var msg1 = RECEIVE
PRINT msg1

var msg2 = RECEIVE
PRINT msg2
```

Figure 17 Client pseudo code

```
SEND ‘A’
IF(Random.nextBoolean == true)
     SEND ‘B’
ELSE
     SEND ‘C’
```

Figure 16 Server pseudo code

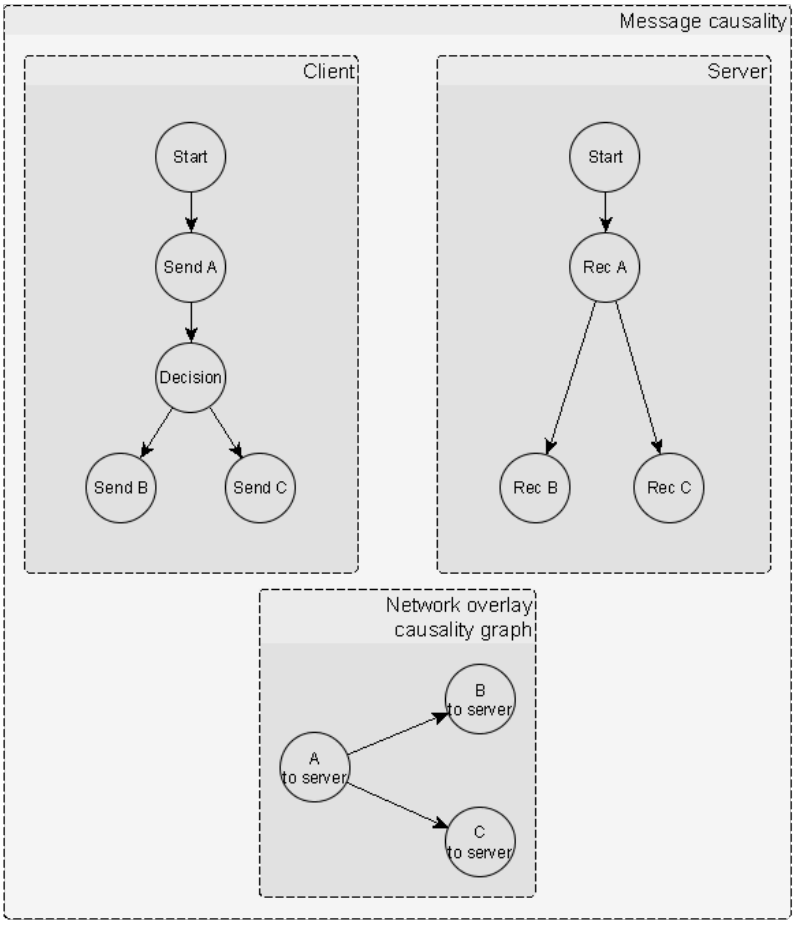

Figure 18 Message causality overlay’s connection to the SuT’s state space

From the dependability engineering point of view, `jointstates` is able to explore data error characteristics in the communication (as well as the internal states of client and

server) – however, not timing errors (see e.g. [7] for a taxonomy). This is mainly due to the verification mechanism's start time-synchronization (described in section 4.2.2).

4.2.2.1 Joint state

Joint states are utilized for message causality management. A joint state consists of two process states; the client SuT verification state and the server SuT verification state. Two process states are managed by two separate Java PathFinder instances as seen in section 4.2.1. SuT verification states are linked to each other by a joint state identifier [JSID] in certain situations (detailed description of these situations is located in section 4.2.2.2). Joint state creation occurs when the verification algorithm identifies a new message type on the actual joint state depth (also defined in section 4.2.2.2).

Joint state transitions define the possible JSID transitions according to the message type and message destination as depicted below. According to Figure 19, when the server verification is in JSID 1 a branching occurs because of a data choice; the receiver may receive message *B* or *C* as well. This situation is similar to a random data choice described in section 3.2.1. On the message recipient side, a data choice generator is created and both scenarios are checked by `jointstates`.

A joint state is a simplified representation of a message exchange trail. JSID carries the information of all joint state transitions which lead to the actual joint state from the beginning of verification. The data structure's simplicity has a performance enhancing effect on the verification process compared to a possible implementation with actual exchanged message history management.

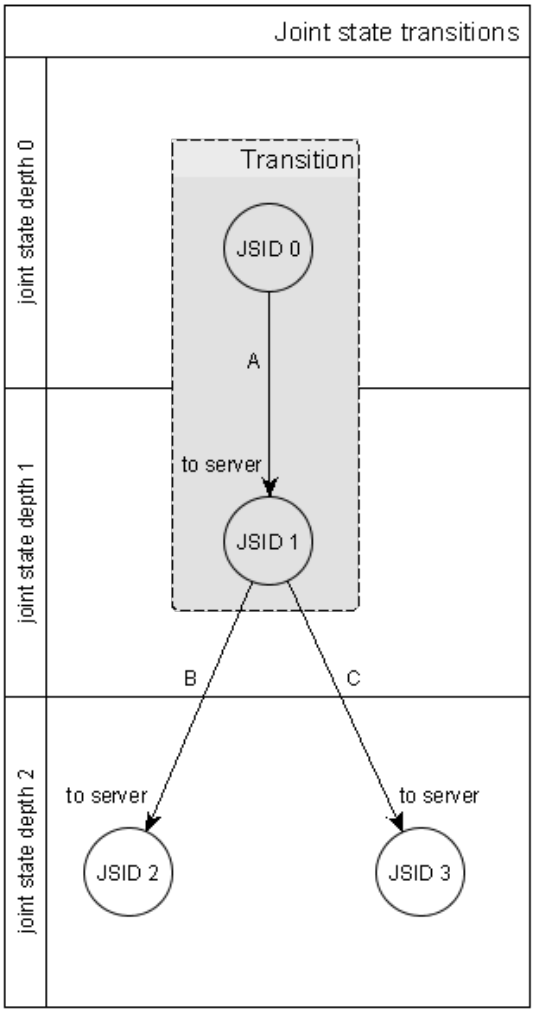

Figure 19 Joint state transition of a joint state

4.2.2.2 Algorithm types

`Jointstates`' verification mechanism can be separated into two separate algorithms. The master instance executes a very simple synchronization stepping algorithm (depicted on Figure 30) while slaves run the networking capable JPF verification algorithm stepped according to the master's synchronization commands. Slaves run a heuristic algorithm which implements the state space exploration logic. The heuristic combines breadth-first search [BFS] and depth-first search [DFS] algorithms.

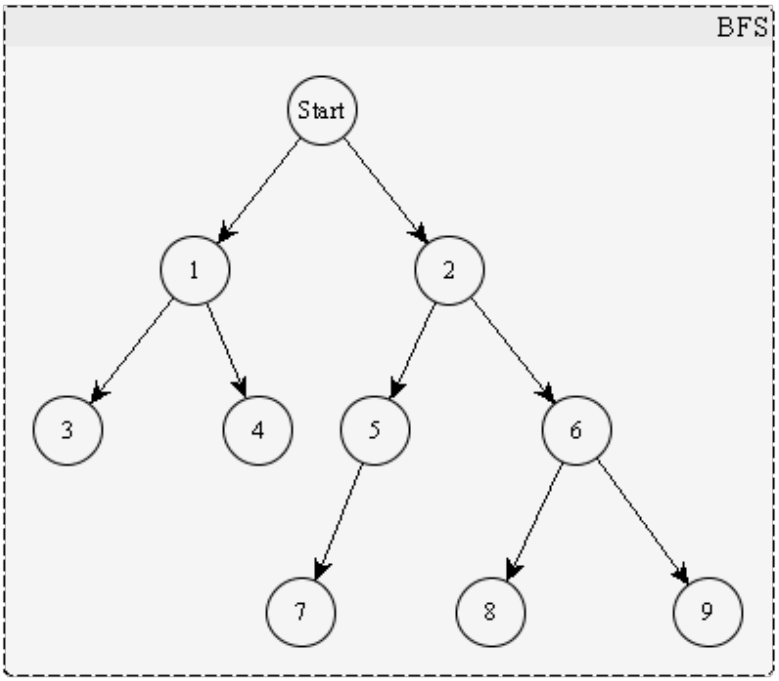

Figure 20 BFS state space exploration

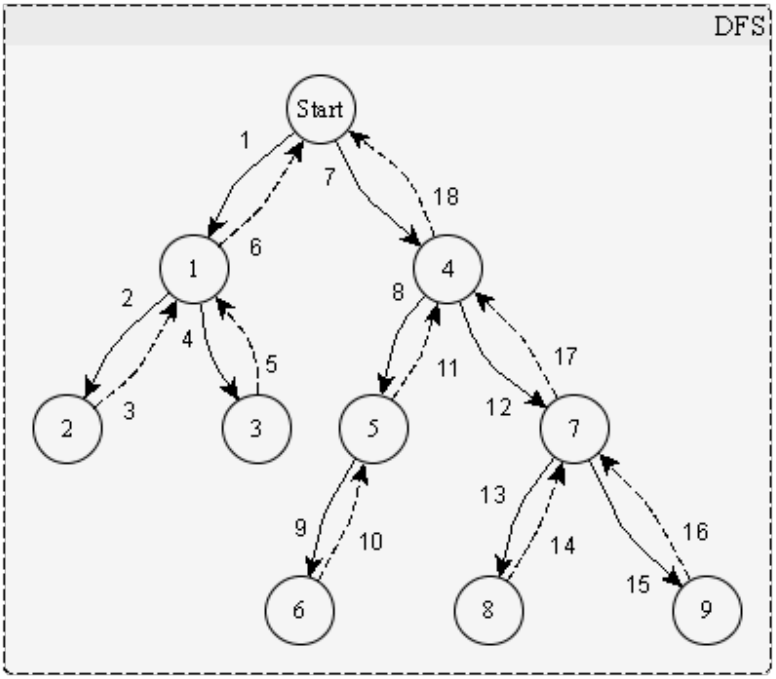

Figure 21 DFS state space exploration

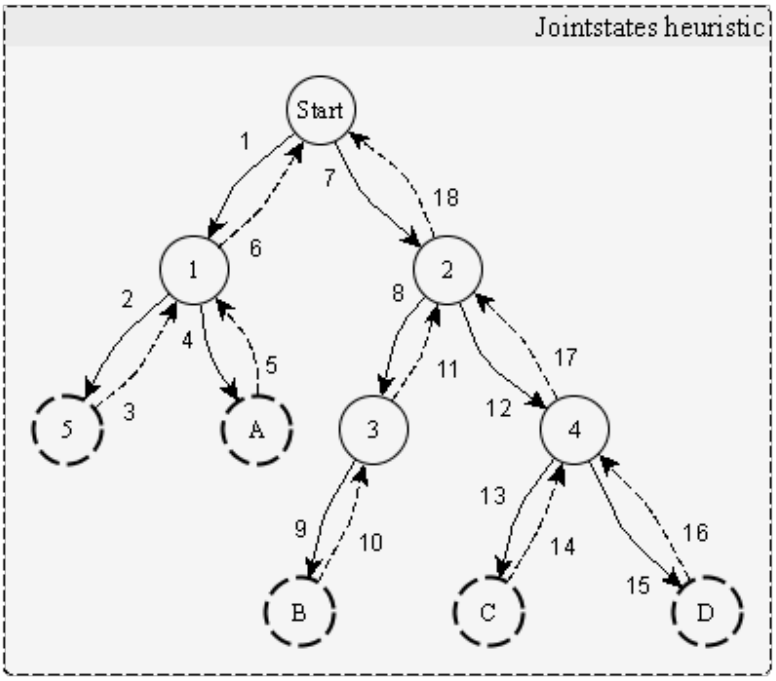

Figure 22 Jointstates state space exploration (simplified version)

The following pseudo code represents the `jointstates` slave heuristic DFS+BFS algorithm, followed by detailed explanation.

```
ENQUEUE
WHILE true {
     var root = DEQUEUE
     IF(root == NULL) {
          BREAK
     }
     DFS.setRoot(root)
     WHILE (var currentState = DFS.next()) != NULL {
          IF currentState IS JOINTSTATE {
               IF currentState IS WRITESTATE {
                    setPriority(getJointStateDepth() + HIGH_PRIORITY)
               } ELSE currentState IS READSTATE {
                    setPriority(getJointStateDepth() + LOW_PRIORITY)
               }
               ENQUEUE
               DFS.doBacktrack()
          } else {
               JPF.checkProperties()
          }
     }
}
```

The search proceeds mainly as a BFS calling multiple DFS instances. The DFS exploration continues until it hits a joint state marked by interrupted outline on Figure 22. The algorithm enqueues the joint state into a prioritized queue and backtracks as if DFS

reached the bottom of the graph. When DFS terminates (at the Start state at first), BFS dequeues the next joint state and starts a DFS with a root of the dequeued state (marked with *5*). The DFS just started explores normal and joint states below this state. When it finishes, BFS dequeues the next state (marked with *A*) and performs the next DFS exploration.

The model checker instance verifying the message sender SuT needs no input from the outer world to continue the verification. On the other hand, the receiver side model checker instance requires the possible inputs to create a data choice generator with the possible choices. The use of a prioritized queue is required because of this asymmetry.

`Jointstates` defines the indicator called joint state depth. A state has a joint state depth of *x* if it has *x* pieces of joint state ancestors. For example, state *B+3* has joint state depth 1 because it has only one joint state ancestor (which is state *B*).

The algorithm enqueues states with almost the same priority if they have the same joint state depth; they form a priority group. In to BFS, higher joint state depth has lower priority. States enqueued into the same priority group are separated into two priority subgroups; a write state group and a read state group. Because of write operations do not require, but read operations do require information from the outside World, write state subgroups have higher priorities compared to read state subgroups located in the same priority group (Figure 24 and Figure 28).

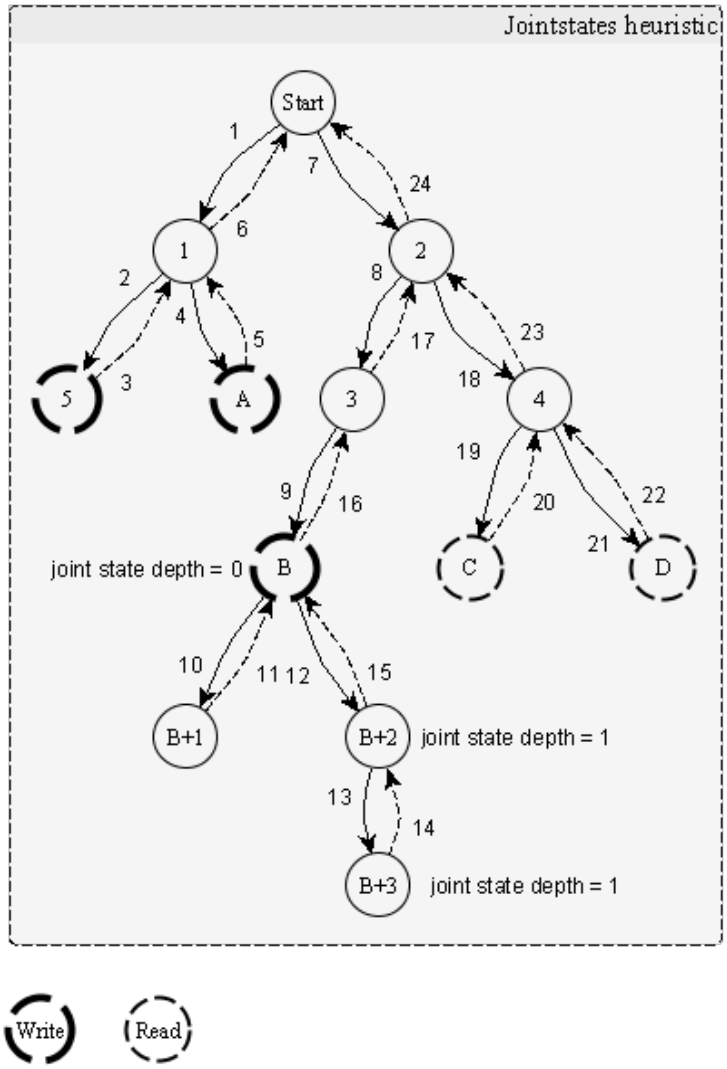

Figure 23 Write and read states

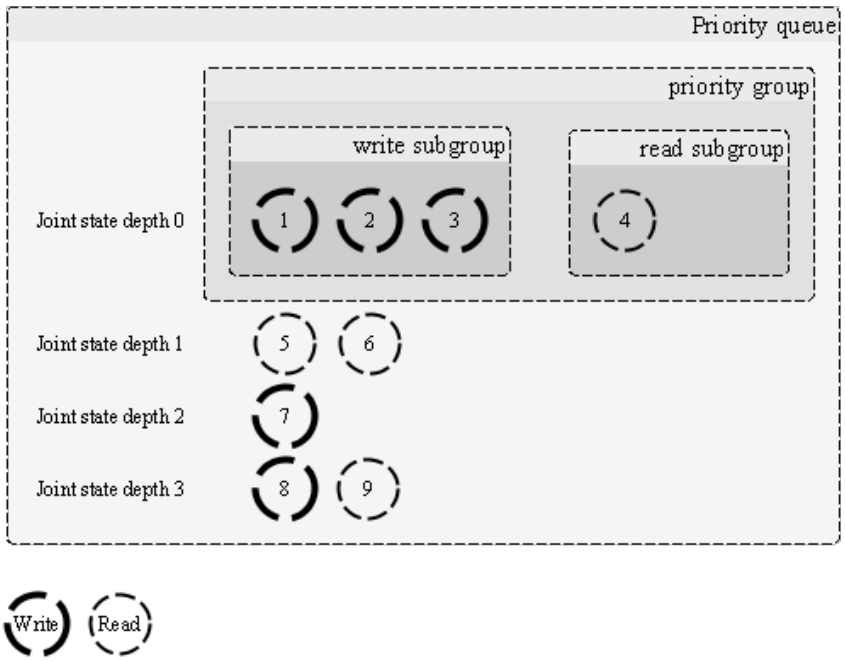

Figure 24 Priority queue's dequeued joint state order

4.3 Implementation

4.3.1 Jointstates modules

Figure 15 shows only a glimpse of `jointstates`' architectural concept. The figures below extend it with general and JPF specific implementation data about `jointstates`' architecture.

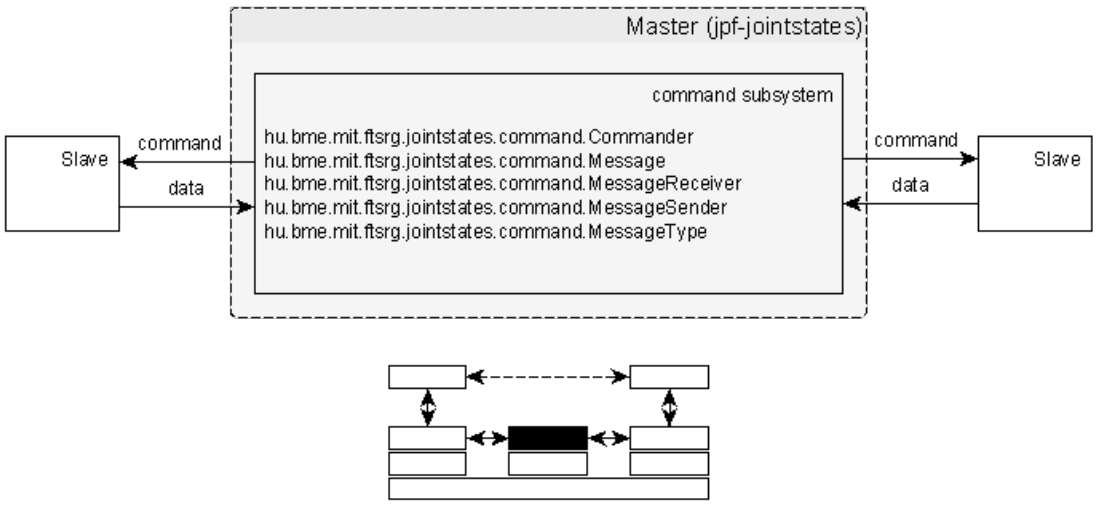

Figure 25 Master module

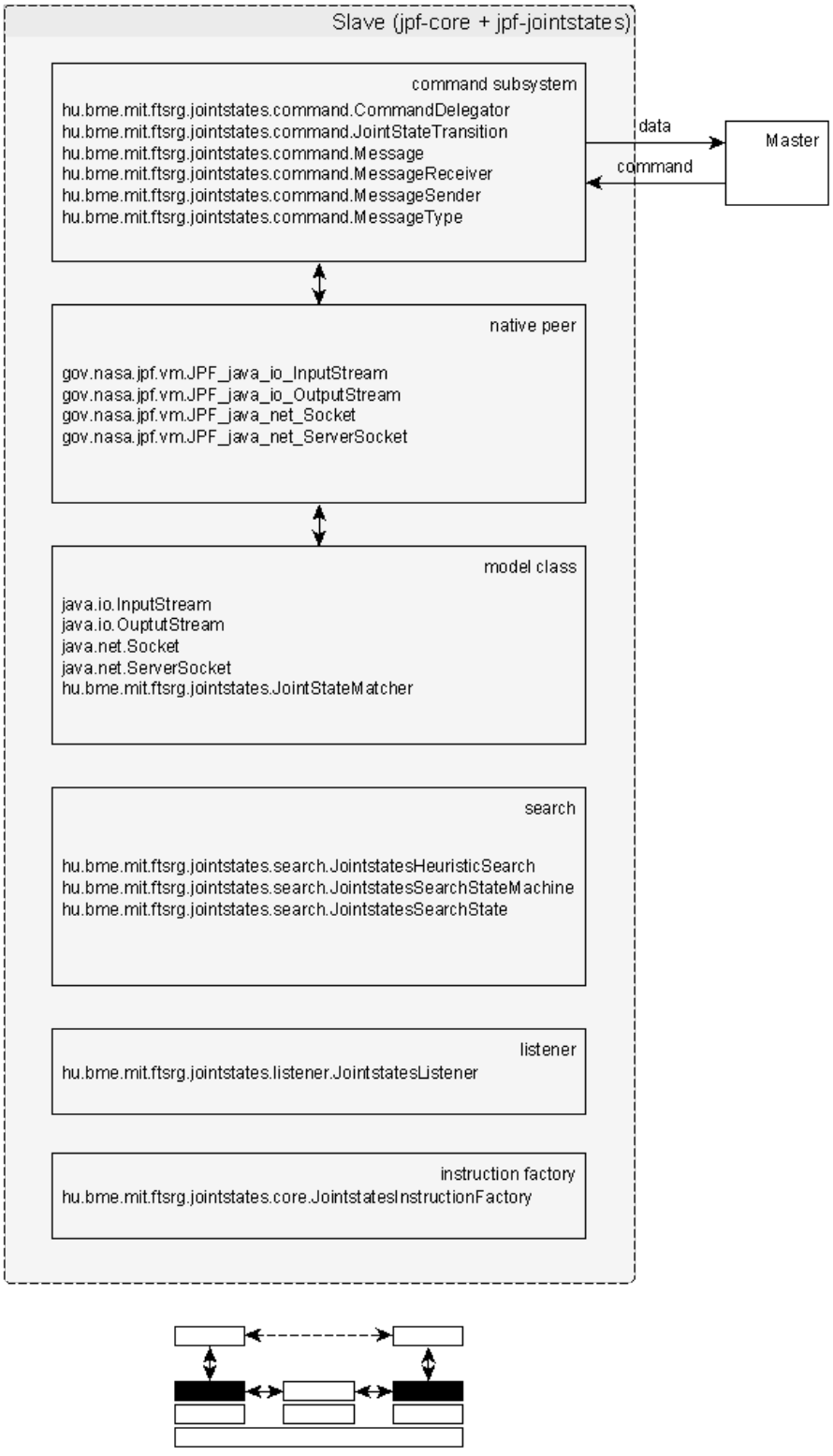

Figure 26 Slave modules

4.3.2 JPF component

As described in section 2.1.2, JPF may be extended due to its modular architecture. `Jointstates` performs the network abstraction by providing special classes

to `jpf-core`. These classes are model classes (section 3.3.1), native peers (section 3.3.1), a bytecode factory (section 3.6), and a heuristic search object (section 3.5).

4.3.2.1 Model class

Model classes override the networking related classes provided by the host JVM. Please note that these classes are not to be executed; they only manipulate the state space exploration. When the SuT bytecode references a class modeled by `jointstates`, JPF loads the modeled class instead of the one provided by the host JVM or `jpf-core`. PathFinder loads it by its own classloader and generates bytecode model instances according to the model class bytecode.

The following classes have been overridden by `jointstates`:

`java.io.InputStream`	Branches the state space when the SuT reads from a socket's input stream by calling the JPF Verify API [35]. Stores value of read state depth.
`java.io.OutputStream`	Notifies the `jointstates` master about new message types on certain joint state depths when the SuT writes to a socket's output stream. Stores value of write state depth. Note that joint state depth can be calculated from the values *InputStream* and *OutputStream* store (read state depth + write state depth).
`java.net.InetAddress`	Routes any networking traffic to *localhost*.
`java.net.InetSocketAddress`	Abstraction required for the class Socket.
`java.net.ServerSocket`	Eliminates the blocking effect of *ServerSocket.accept()* calls.
`java.net.Socket`	Applies socket creation abstraction.

4.3.2.2 Native peer

Native peers capture model class native calls. Message transfer occurs between a slave instance and the master instance when SuT writes to an output stream or reads from

an input stream. Model classes are not executed so transfer needs to be implemented on the host JVM level by native peers. Two native peers (implemented by `jointstates`) capture stream interactions.

```
gov.nasa.jpf.vm.JPF_java_io_InputStream
gov.nasa.jpf.vm.JPF_java_io_OutputStream
```

`Jointstates` applies the networking abstraction when SuT invokes one of the following methods:

```
int java.io.InputStream:read();

void java.io.OutputStream:write(int b);
void java.io.OutputStream:write(byte[] b);
void java.io.OutputStream:write(byte[] b, int off, int len);
```

All modeled write operations are channeled into *java.io.OutputStream:write(byte[] b, int off, int len)* so only one native method exists for write operations as well. In order to pass the invocation to the host JVM the following native method declarations are necessary in the appropriate model classes.

```
native int[] InputStream.native_read(int lastJointStateId, int
  readDepth);

native int OutputStream.native_write(byte[] b, int off, int len, int
  lastJointStateId);
```

The following native peer methods capture the model class native invocations.

```
int JPF_java_io_InputStream.native_read__II___3I(MJIEnv env, int objRef,
  int lastJointStateId, int readDepth);

int JPF_java_io_OutputStream.native_write___3BIII__I(MJIEnv env, int
  objRef, int messageRef, int offset, int length, int lastJointStateId);
```

Note that the return type integer has different meaning for the read and write native method. While write returns a plain integer of the next JSID read operation returns an integer which is a Java reference to an integer array of possible joint state transitions. PathFinder realizes this difference by the native method return type declaration located in the model class, and performs the necessary Java reference resolution transparently.

4.3.2.3 Bytecode factory

`Jointstates` overrides PathFinder's default bytecode factory with *JointstatesInstructionFactory* and adds networking verification related metadata to the bytecode instances when they are created (solution suggested by Peter Mehlitz in personal correspondence). *INVOKEVIRTUAL* bytecode instances which perform socket read or socket write operations are flagged with a unique JPF attribute (described in section 3.7) in order to increase verification performance. Java reference comparison is faster than string value comparison (by orders of magnitude).

4.3.2.4 Listener

`Jointstates` provides a listener called *JointstatesListener*. It is instantiated when `jpf-core` initializes the configured external JPF modules. The listener's overridden event handler functions are called during the verification process.

Most of the event handler methods have logging and initialization purpose. On the other hand, one called *instructionExecuted* – inherited and overridden from the interface *VMListener* – does more than that; it is essential to the verification algorithm. Invocation occurs every time PathFinder executed SuT bytecode. `Jointstates` is able to check whether the next bytecode to be executed is an *INVOKEVIRTUAL* instance flagged by *JointstatesInstructionFactory* or not. If the next bytecode to be executed is flagged, it means the DFS search has reached the bottom of its state space scope. As described in section 4.2.2.2, DFS backtracks only if it hit a leaf node (or a processed state). In order to fake a leaf node deep inside the state space, a dummy choice generator called *BreakGenerator* is injected after the last processed state (which is a joint state). As *BreakGenerator* has no possible choices, DFS is forced to backtrack.

4.3.2.5 Search object

A heuristic algorithm and its heuristic function implements the DFS+BFS algorithm described in section 4.2.2.2. Thankfully, JPF already has extensive support for heuristic verification search strategies located in `jpf-core`'s package *gov.nasa.jpf.search.heuristic*. PathFinder offers developers a simple interface to implement their search strategies through generalization. `Jointstates` implements its heuristic algorithm in *JointstatesHeuristicSearch*. The implementation is based on PathFinder's heuristic solution.

JPF heuristic starts with a state called *root* which represents the process state before the first bytecode execution. Exploration is performed by the method *generateChildren()* which generates the subordinate states of *[root]*. When a state is generated it is enqueued into a priority queue also described in 4.2.2.2. The DFS+BFS algorithm dequeues the state of highest priority and calls *generateChildren()* to explore its subordinate states.

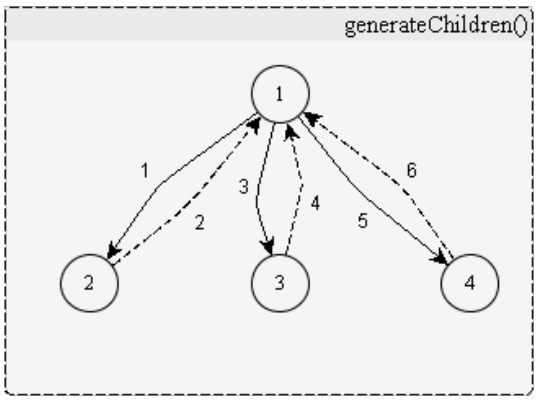

Figure 27 Heuristic state generation

`GenerateChildren()` is implemented by `jpf-core` but `JointstatesHeuristicSearch` wraps it by method override. Necessity of wrapping it comes from children generation of a read or write state. It may only proceed when the master orders to do so.

State priority is calculated by `computeHeuristicValue()`. The function priority computation logic implements the DFS+BFS algorithm in the following way (lower value has higher priority).

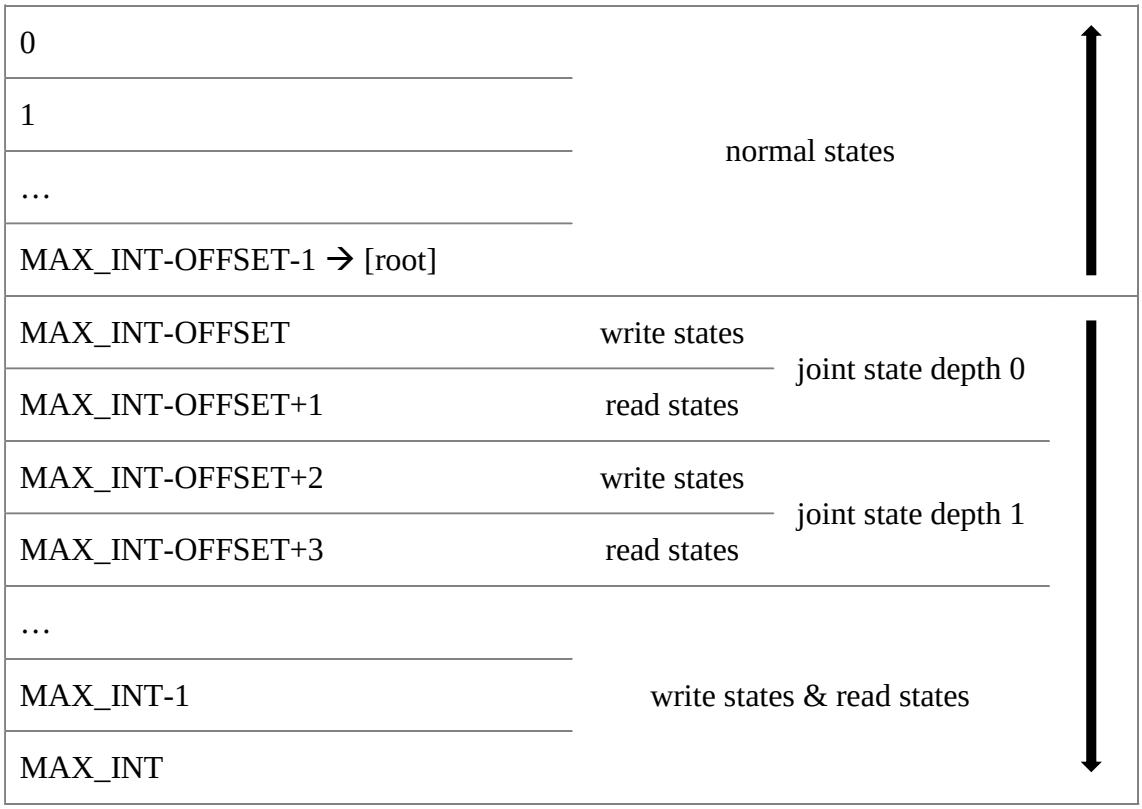

Figure 28 State priority assignment; lower value has higher priority

4.3.3 Message transfer system

`Jointstates` utilizes a two-way message delivery mechanism which transports commands and verification data transparently.

Figure 29 Message transfer

4.3.3.1 Command distribution

Commands are issued by the master in order to synchronize the search mechanism described in section 4.2.2.2. Synchronization algorithm is represented by the following state chart diagram.

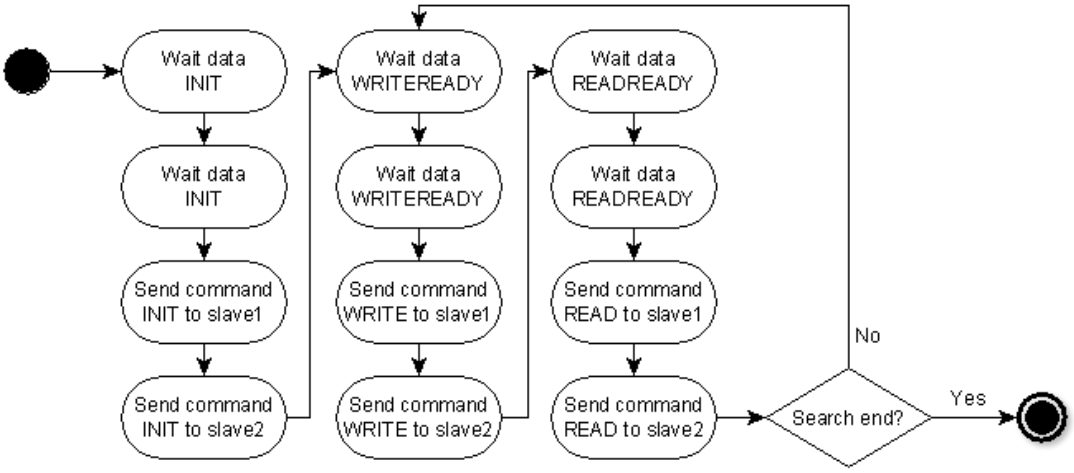

Figure 30 Synchronization algorithm

The distributed search is preceded by an initialization phase. Slave sends an `INIT` message when it is able to start the verification. The next two columns step the write state and read state dequeue mechanism depicted on Figure 23 and Figure 24 by joint state depths until the distributed search ends.

Master's algorithm sends these messages by utilizing the `MessageSender` class. Its sole purpose is to read the next queued message from a thread-safe outbound queue, to read its recipient and to deliver it. Messages are received by the class `MessageReceiver` which enqueues the message into a thread-safe FIFO queue.

4.3.3.2 Data provision

Verification data is provided and utilized also by slave instances. When the heuristic search invokes a native peer to perform network operations data provision occurs.

Master instance contains a component called `Aggregator` which collects provided verification data from slave instances. The class stores joint state transitions on each joint state depth per message recipient. When a SuT performs a write or read operation one of its native peers performs an `ADD` or a `QUERY` operation on the `Aggregator`.

`ADD` operations may extend the `Aggregator`'s joint state transition database through `JPF_java_io_OuptutStream` when a SuT performs a write operation. Input required to add a new joint state transition is the current joint state identifier, message recipient and the message itself. The `Aggregator` returns the next joint state identifier the SuT gained by sending the message. The new JSID is saved by the slave's model class

`JointStateMatcher`. Its model class nature attaches the joint state identifier to the JPF verification state which is forwarded and backtracked along with it, saved and restored from time to time.

QUERY occurs when an SuT hits a read operation. The native peer `JPF_java_io_InputStream` starts a `QUERY` on the `Aggregator`. By doing so, the slave provides the current JSID to the `Aggregator` which responds with an array of joint state transitions. These transitions serve as the input for branching the SuT's state space. After the branching occurred and one of the transitions is selected by the current choice generator, the next JSID is extracted from the selected transition, and saved into the model class `JointStateMatcher`. At the end, the native peer returns with the transition provided message.

5 Example

`Jointstates`' source code [32] provides three client-server pairs in the package `hu.bme.mit.ftsrg.jointstates.examples`. One of them represents a fault-free system (`DummyClient` and `DummyServer`), the other two contains concurrency related bug (`LuckyClient` and `LuckyServer`, `BuggyClient` and `BuggyServer`). The three systems' source code is almost identical. `Dummy` system's behavior may be represented by the sequence diagram below. The client chooses an integer from a set of two and sends it. Server returns it multiplied by ten. The other two systems are based on this one.

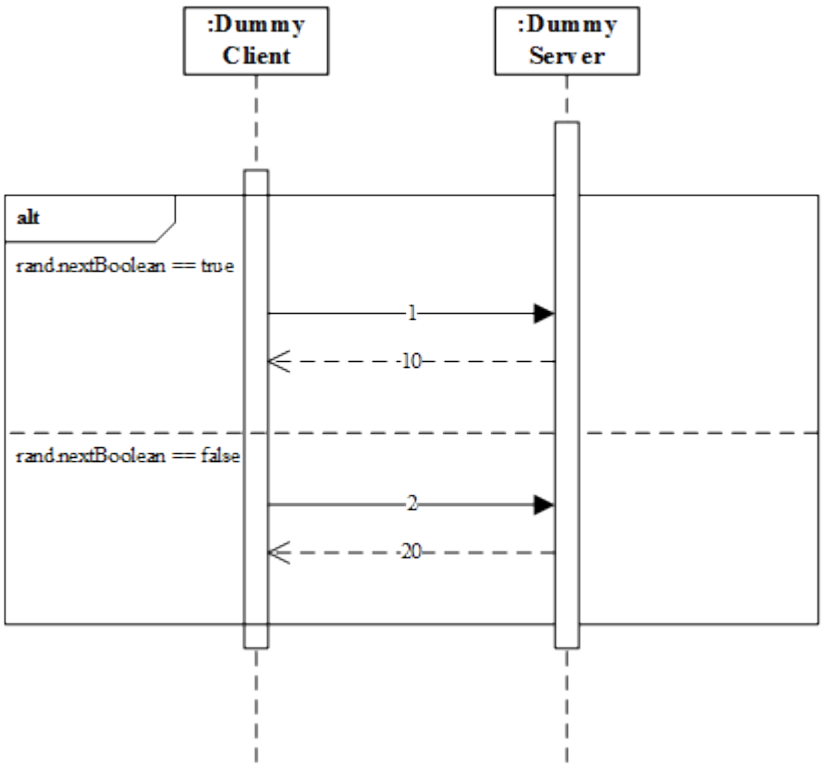

Figure 31 Dummy system behavior

The verification found no errors as expected. `Jointstates` generated the following verification output:

```
DummyClient.jpf
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:05
states:             new=12,visited=1,backtracked=13,end=2
search:             maxDepth=7,constraints=0
choice               generators:                                thread=10
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=8), data=2
heap:               new=503,released=87,maxLive=405,gcCycles=13
instructions:       4322
max memory:         244MB
loaded code:        classes=64,methods=1390

======================================================  search  finished:
5/3/14 6:41 PM


DummyServer.jpf
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:01
states:             new=12,visited=1,backtracked=13,end=2
search:             maxDepth=7,constraints=0
choice               generators:                                thread=10
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=8), data=2
heap:               new=477,released=83,maxLive=395,gcCycles=13
instructions:       4212
max memory:         244MB
loaded code:        classes=63,methods=1374

======================================================  search  finished:
5/3/14 6:41 PM
```

`Lucky` system's behavior is identical to `Dummy`'s when executed. The only difference is located in the client's program control. It has a control branch which is executed if the client receives an unexpected integer. Client enters its own Java monitor by calling `this.wait()`. This monitor enter should be considered as a concurrency bug regarding there are no threads alive which would issue `notify()` or `notifyAll()`. `Jointstates` would report this bug if it would work according to an open World assumption. Since the solution assumes a closed World environment this is not a bug; the client will issue `this.wait()` under no circumstances because `LuckyServer` always returns the expected integer during verification (and execution as well). The `Lucky` system's behavior is depicted on the following diagram.

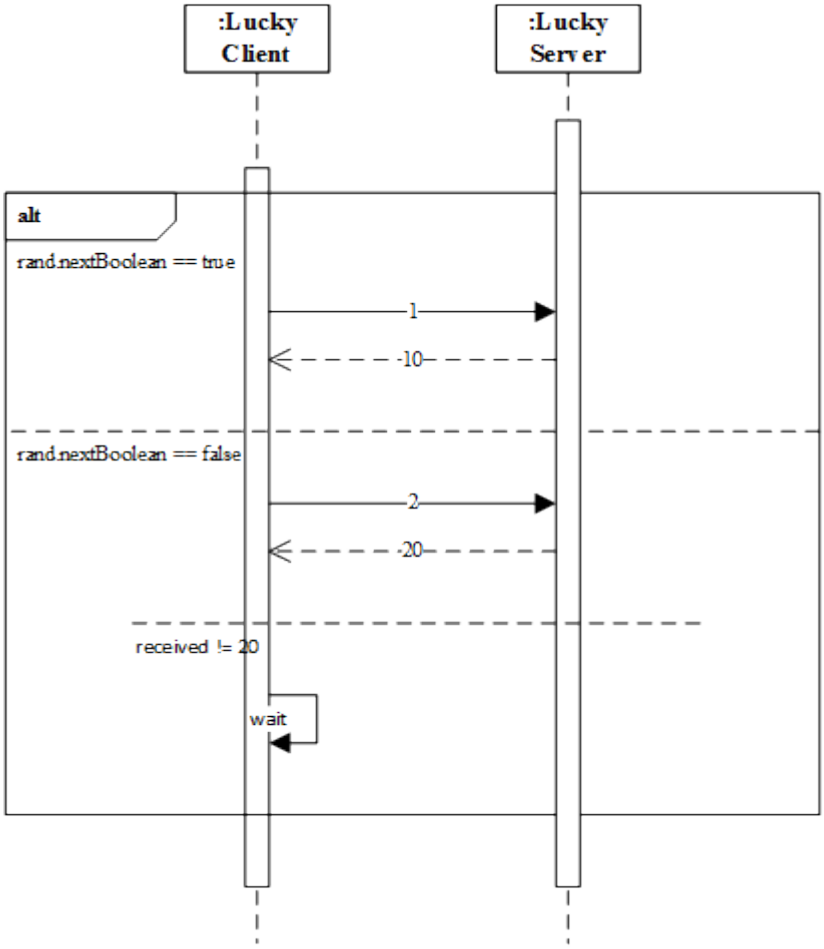

Figure 32 Lucky system behavior

When `jointstates` verifies the system it finds no bugs as expected. The verification generates the following output.

```
LuckyClient.jpf
================================================== results
no errors detected

================================================== statistics
elapsed time:       00:00:09
states:             new=12,visited=1,backtracked=13,end=2
search:             maxDepth=7,constraints=0
choice               generators:                              thread=10
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=8), data=2
heap:               new=503,released=87,maxLive=405,gcCycles=13
instructions:       4322
max memory:         244MB
loaded code:        classes=64,methods=1390

==================================================  search  finished:
5/3/14 6:47 PM

LuckyServer.jpf
================================================== results
no errors detected

================================================== statistics
elapsed time:       00:00:01
states:             new=12,visited=1,backtracked=13,end=2
search:             maxDepth=7,constraints=0
choice               generators:                              thread=10
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=8), data=2
heap:               new=477,released=83,maxLive=395,gcCycles=13
instructions:       4212
max memory:         244MB
loaded code:        classes=63,methods=1374

==================================================  search  finished:
5/3/14 6:47 PM
```

The most complex (yet still relatively simple) scenario among the examples is the case of system `Buggy`. Like at the previous one client executes `this.wait()` if it receives an unexpected integer. The difference between the systems *Lucky* and *Buggy* is their server's control logic. `BuggyServer` sometimes responds an unexpected integer so `BuggyClient`'s fault is able to activate. In this case the client contains a concurrency bug according to the closed World assumption as well. The system's behavior may be observed on Figure 33.

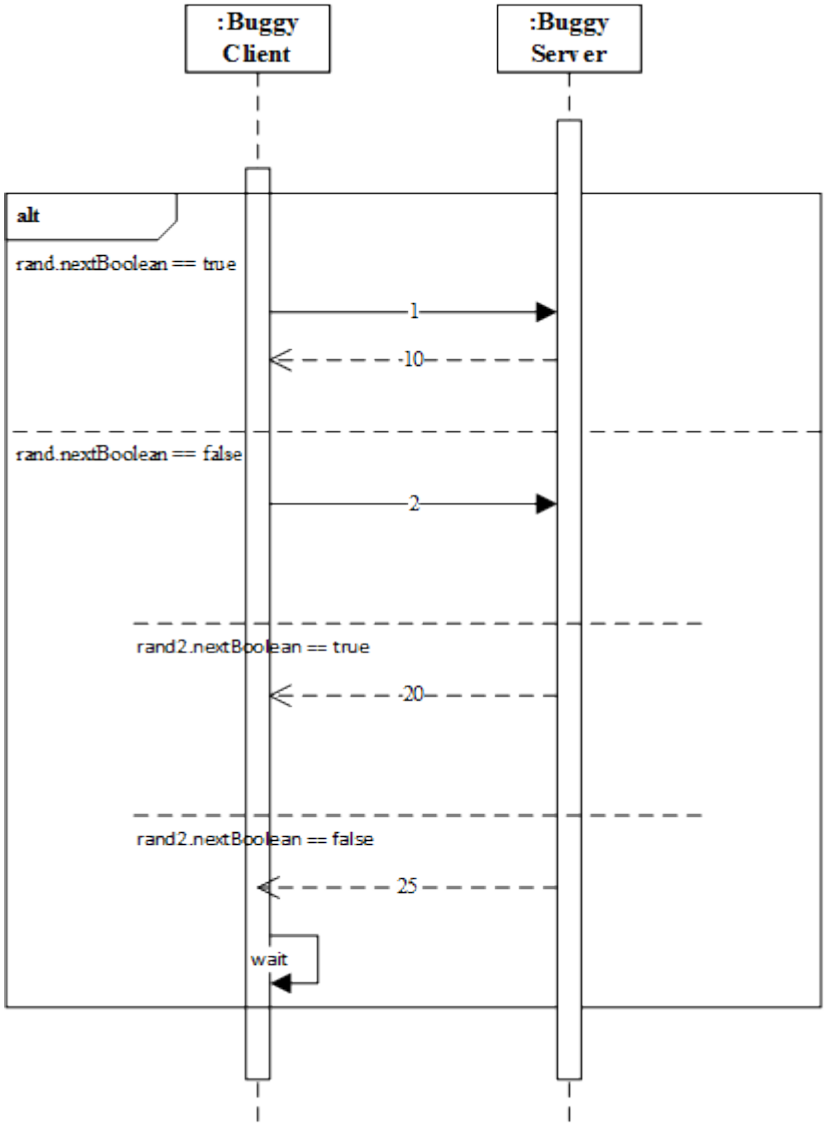

Figure 33 Buggy system behavior

`Jointstates` finds the bug in the client's source code and prints the code lines which lead to it.

```
BuggyClient.jpf
====================================================== thread ops #1
   1     trans      insn          loc                : stmt
------- --------------------------------------------------
W:165                           18                        invokevirtual
hu/bme/mit/ftsrg/jointstates/examples/BuggyClient.java:90 : this.wait();
   S         0
====================================================== results
error #1: gov.nasa.jpf.vm.NotDeadlockedProperty "deadlock encountered:
thread java.lang.Thread:{..."

====================================================== statistics
elapsed time:       00:00:02
states:             new=19,visited=0,backtracked=18,end=2
search:             maxDepth=8,constraints=0
choice               generators:                                thread=12
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=10), data=3
heap:               new=519,released=80,maxLive=395,gcCycles=19
instructions:       4413
max memory:         244MB
loaded code:        classes=64,methods=1390

======================================================  search  finished:
5/3/14 7:14 PM

BuggyServer.jpf
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:05
states:             new=26,visited=3,backtracked=29,end=6
search:             maxDepth=8,constraints=0
choice               generators:                                thread=20
(signal=0,lock=1,sharedRef=0,threadApi=1,reschedule=18), data=4
heap:               new=575,released=201,maxLive=399,gcCycles=29
instructions:       4834
max memory:         244MB
loaded code:        classes=64,methods=1386

======================================================  search  finished:
5/3/14 7:14 PM
```

6 Conclusions and Future Work

6.1 Current limitations

`Jointstates` is able to verify a pair of Java client and server with the following set of limitations:

- Communication between sides must take place on one port. The default port is 8080; it cannot be modified at this time. This limitation can be overcome by further development.
- Master and the two slave instances must run on the same host. This limitation can be easily overcome by further development in configurability since commands and verification data is transmitted through socket communication. Hardcoded values must be substituted by configuration provided values.
- Client and server must communicate through native Java sockets and exchange integers only. This limitation can be overcome by further feature development.
- Verification has lower performance compared to `net-iocache` which in fact questions the feasibility of an industrial software's verification. This limitation comes from `jointstates`' verification strategy, it may not be overcome easily. The nature of `jointstates`' verification algorithm eases this symptom by providing the possibility of bounded model checking. This possibility comes from the BFS wrapper algorithm which iterates through the verified joint state depths. An engineer-provided maximum depth may serve as the bound of verification.
- SuT must utilize `InputStream` and `OutputStream` for socket purposes only. This limitation can be overcome by further development.
- Master instance does not stop when slaves have finished. This limitation can be easily overcome by further development. Master should act appropriately when *END* messages arrived from both sides to stop its execution.

6.2 Advantages of the approach

Regarding the existing published solutions which enable PathFinder to verify networked applications (see section 2.3) `jointstates` has a different approach in its verification strategy. While `net-iocache` has the strength of verifying applications with high performance causing a degraded verification correctness, `jointstates` aims to maximize its verification correctness over performance. The solution does not allow the possibility of reaching unreachable system states thanks to its message causality concept.

The state space reduction possibilities depicted on Figure 3 are implemented in `jointstates` in an implicit way. The concept of its verification strategy disallows the verification of unnecessary joint states which results in a desired, smaller joint state space.

6.3 Further development

Limitations described in section 6.1 should have top priority in the project's future development. They must be overcome before any kind of feature development.

6.3.1 Desired features

JPF defines normal behavior with JPF Properties. Java PathFinder - like most of the model checking tools – provide a counterexample for a violated property when found. `Jointstates` should provide the exchanged message trail when a property is violated.

The concept is not confined to client-server verification. Implementation expects any number of communicating sides. `Jointstates` does not assume a strict client-server communication where only clients may initiate the communication. A general implementation should lead to a solution which is able to verify a peer-to-peer [P2P] system with any number of nodes.

Acknowledgements

I would like to thank Imre Kocsis (assistant lecturer at Budapest University of Technology and Economics, Department of Measurement and Information Systems, Fault Tolerant Systems Research Group) for accepting me as his student. He provided continuous professional guidance on the field of reliable engineering during my two and a half years of PathFinder research. His extensive article writing experience has been valuable during the document review process.

I am grateful to András Vörös (assistant lecturer at Budapest University of Technology and Economics, Department of Measurement and Information Systems, Fault Tolerant Systems Research Group) for helping me with his impressive model checking knowledge and for his support regarding the review of this document.

I would like to thank Peter Mehlitz (computer scientist at NASA, Ames Research Center, Robust Software Engineering Group) and Sergio Feo (computer science researcher at Albert–Ludwigs-Universität Freiburg) for helping me with Java PathFinder related technical challenges.

References

[1] Uppsala University, “UPPAAL,” 2010. [Online]. Available: http://www.uppaal.org/. [Accessed: 05-May-2014].

[2] Bell Labs, “Spin.” [Online]. Available: http://spinroot.com/spin/whatispin.html. [Accessed: 13-May-2014].

[3] NASA, “Java PathFinder - Homepage.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed: 27-Apr-2014].

[4] C. Artho and P. Garoche, “Accurate Centralization for Applying Model Checking on Networked Applications,” *21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 177–188, 2006.

[5] NASA, “Java PathFinder - JPF modules.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/start. [Accessed: 14-May-2014].

[6] A. Avizienis and J. Laprie, “Basic concepts and taxonomy of dependable and secure computing,” … *Secur. Comput.* …, vol. 1, no. 1, pp. 11–33, 2004.

[7] A. Bondavalli and L. Simoncini, “Failure classification with respect to detection,” in *Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990, pp. 47–53.

[8] DBLP Computer Science Bibloigraphy, “Watcharin Leungwattanakit - Bibliography.” [Online]. Available: http://www.informatik.uni-trier.de/~ley/pers/hd/l/Leungwattanakit:Watcharin. [Accessed: 14-May-2014].

[9] W. Leungwattanakit, “Networked software model checking by extending Java PathFinder,” University of Tokyo, 2008.

[10] C. Artho, “Homepage.” [Online]. Available: https://staff.aist.go.jp/c.artho/. [Accessed: 14-May-2014].

[11] C. Artho and W. Leungwattanakit, “Net-iocache homepage.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/net-iocache. [Accessed: 27-Apr-2014].

[12] S. Feo, “jpf-net-master-slave @ Assembla,” 2013. [Online]. Available: https://www.assembla.com/spaces/jpf-net-master-slave/new_items. [Accessed: 09-May-2014].

[13] National Institute of Informatics, “国立情報学研究所/National Institute of Informatics homepage.” [Online]. Available: http://www.nii.ac.jp/en/. [Accessed: 27-Apr-2014].

[14] Y. Tanabe, “田辺良則/Tanabe Yoshinori homepage.” [Online]. Available: http://cent.xii.jp/tanabe.yoshinori. [Accessed: 27-Apr-2014].

[15] I. J. Sessink, “Formal Analysis of Jackrabbit Software Using Java PathFinder,” *Analysis*, 2008.

[16] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, “Cache-Based Model Checking of Networked Applications: From Linear to Branching Time,” *2009 IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 447–458, Nov. 2009.

[17] W. Leungwattanakit, “I / O Cache Implementation Guide 2012.

[18] D. Giannakopoulou, C. S. Pasareanu, M. Lowry, and R. Washington, “Lifecycle Verification of the NASA Ames K9 Rover Executive,” *http://ti.arc.nasa.gov/publications*, pp. 1–11, 2004.

[19] Fujitsu, “Fujitsu Accelerates Exhaustive Verification of Java Software Through Parallel Processing,” 2010. [Online]. Available: http://www.fujitsu.com/global/news/pr/archives/month/2010/20101217-02.html. [Accessed: 15-May-2014].

[20] NASA, “Java PathFinder - stack figure.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/user/components/jpf-stack.png. [Accessed: 27-Apr-2014].

[21] Oracle Corporation, “The Java® Virtual Machine Specification, Java SE 7 Edition,” 2013.

[22] Oracle Corporation, “Java™ Reflection API.” [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/reflection/. [Accessed: 27-Apr-2014].

[23] NASA, “Non-deterministic data acquisition example.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/random_example. [Accessed: 27-Apr-2014].

[24] Oracle Corporation, “Java™ Platform SE 7 Thread.State,” 2013. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html. [Accessed: 27-Apr-2014].

[25] NASA, “Java PathFinder - Partial Order Reduction (POR).” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/partial_order_reduction. [Accessed: 27-Apr-2014].

[26] Oracle Corporation, “Java™ Native Interface 6.0 specification,” 2013. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. [Accessed: 27-Apr-2014].

★ Table of Cont

[27] NASA, “Java PathFinder - Model Java Interface (MJI).” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji. [Accessed: 27-Apr-2014].

[28] Oracle Corporation, “Java™ Platform SE 7 ClassLoader,” 2013. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html. [Accessed: 27-Apr-2014].

[29] NASA, “Java PathFinder MJI name mangling.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji/mangling. [Accessed: 28-Apr-2014].

[30] Oracle Corporation, “Java™ Native Interface name mangling.” [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp615. [Accessed: 28-Apr-2014].

[31] NASA, “Peter Mehlitz.” [Online]. Available: http://ti.arc.nasa.gov/profile/pcmehlitz/. [Accessed: 11-May-2014].

[32] D. Lakatos, “jpf-jointstates,” 2013. [Online]. Available: https://bitbucket.org/conoyes/jpf-jointstates. [Accessed: 03-May-2014].

[33] Atlassian, “Mercurial by Bitbucket.” [Online]. Available: https://bitbucket.org/. [Accessed: 15-May-2014].

[34] Selenic, “Mercurial SCM.” [Online]. Available: http://mercurial.selenic.com/. [Accessed: 15-May-2014].

[35] NASA, “Java PathFinder Verify API.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/api#TheVerifyAPI1. [Accessed: 28-Apr-2014].

Appendix

Heuristic function

```
/*
 * The heuristic goes this way. Do a DFS search but don't cross joint state
 * depths. When out of uninteresting states produced by DFS, get the next
 * interesting joint state and explore its state space by DFS.
 * @see
 * gov.nasa.jpf.search.heuristic.SimplePriorityHeuristic#
   computeHeuristicValue
 * ()
 */
@Override
protected int computeHeuristicValue() {
  int heuristicValue = -1;
  int currentJointStatesDepth = getJointStatesDepth();
  int nextJointStatesDepth = currentJointStatesDepth + 1;

  ThreadInfo ti = this.vm.getCurrentThread();
  Object attr = null;

  if (ti != null) {
    if (ti.getPC() != null) {
      attr = ti.getPC().getAttr();
    }
  }

  // The state is a joint state
  if (attr != null) {
    if (attr == JointstatesInstructionFactory.writeFlag) {
      logger.warning("jointstates computing heuristic "
          + "value for write state");

      // has higher priority than the read tasks (-1)
      heuristicValue = Integer.MAX_VALUE - PRIORITY_OFFSET + 2
          * nextJointStatesDepth - 1;
    } else if (attr == JointstatesInstructionFactory.readFlag) {
      logger.warning("jointstates computing heuristic "
          + "value for read state");

      // has lower priority than the write tasks (missing -1)
      heuristicValue = Integer.MAX_VALUE - PRIORITY_OFFSET + 2
          * nextJointStatesDepth;
    }
  }
  // The state is a normal state (simple DFS)
  else {
    logger.warning("jointstates computing heuristic "
        + "value for normal state");

    heuristicValue = Integer.MAX_VALUE - PRIORITY_OFFSET
        - this.vm.getPathLength() - 1;
  }

  logger.warning("jointstates computed heuristic value " + heuristicValue);
```

```
  // -100 is because we would like to avoid priority collisions between the
  // original DFS states and the joint states
  return heuristicValue;
}
```