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, András Vörös

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. 11.

...…….

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 közvetlenül Java bytekódot 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 method 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 or Promela. One of the exceptions is Java PathFinder. 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 modified, model checking enabled classes. Although this brilliant feature exists, analyzing error propagation by Java PathFinder is difficult and rudimentary.

My goal is to assemble a complete solution for analyzing dataflow networks. 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.

Acknowledgements

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 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 too. One component's failure may cause a spreading error phenomenon called error propagation. Verification engineers should be able to predict how one component failure affect the system. This analysis requires an error propagation modeling technique which is able to enlist the possible component failures and predict its effect on the relying systems.

Formal verification is usually a thorough but time consuming software verification technique. The tools provided for formal verification usually cannot be utilized easily during software development nor on the completed product. 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 increases the cost and the complexity of the verification radically. A tool named Java PathFinder [1] (hereinafter 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 exceptions or automatic test case generation. Formally verifying single threaded Java programs is difficult due to the huge or infinite state space caused by 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. As soon the system under test software (hereinafter SuT) reaches for information outside its Java

Virtual Machine (hereinafter JVM) JPF needs extensions engineered by 3rd parties to provide a proper abstraction of the outer world.

2 Model checking networked systems

Formal verification in general is usually a complex and resource demanding procedure. Most common problem with finite resources is the problem of 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. The effect is illustrated on Figure 1.

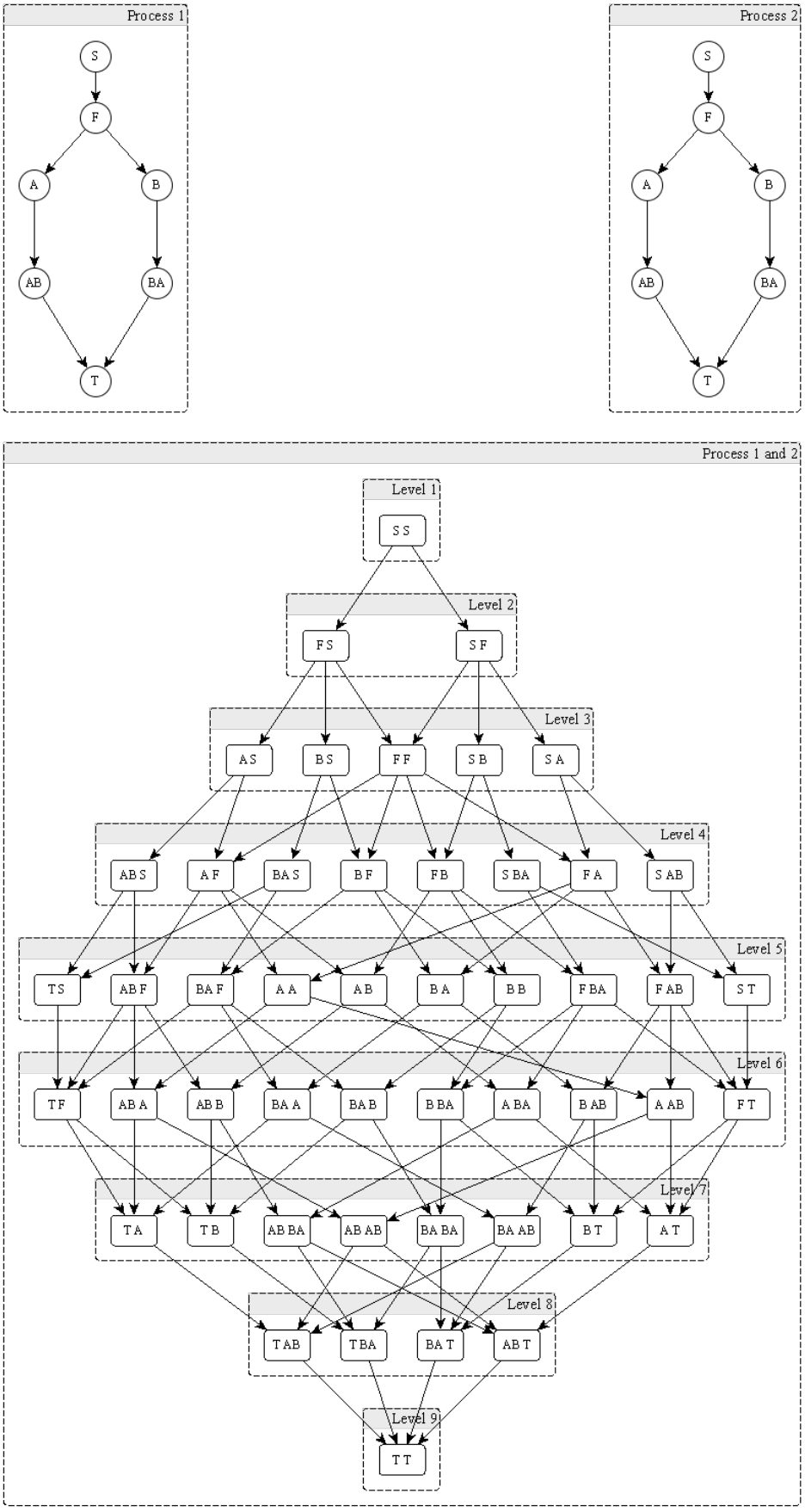

Figure 1 Joint state space of two communicating processes

When considering model checking networked applications, some factors mitigate the rate of explosion. For example, when the server SuT has not reached the verification state in which it listens on a local port, the client SuT should not connect to it yet. The reduced state space of a simplified client-server is sketched on Figure 2.

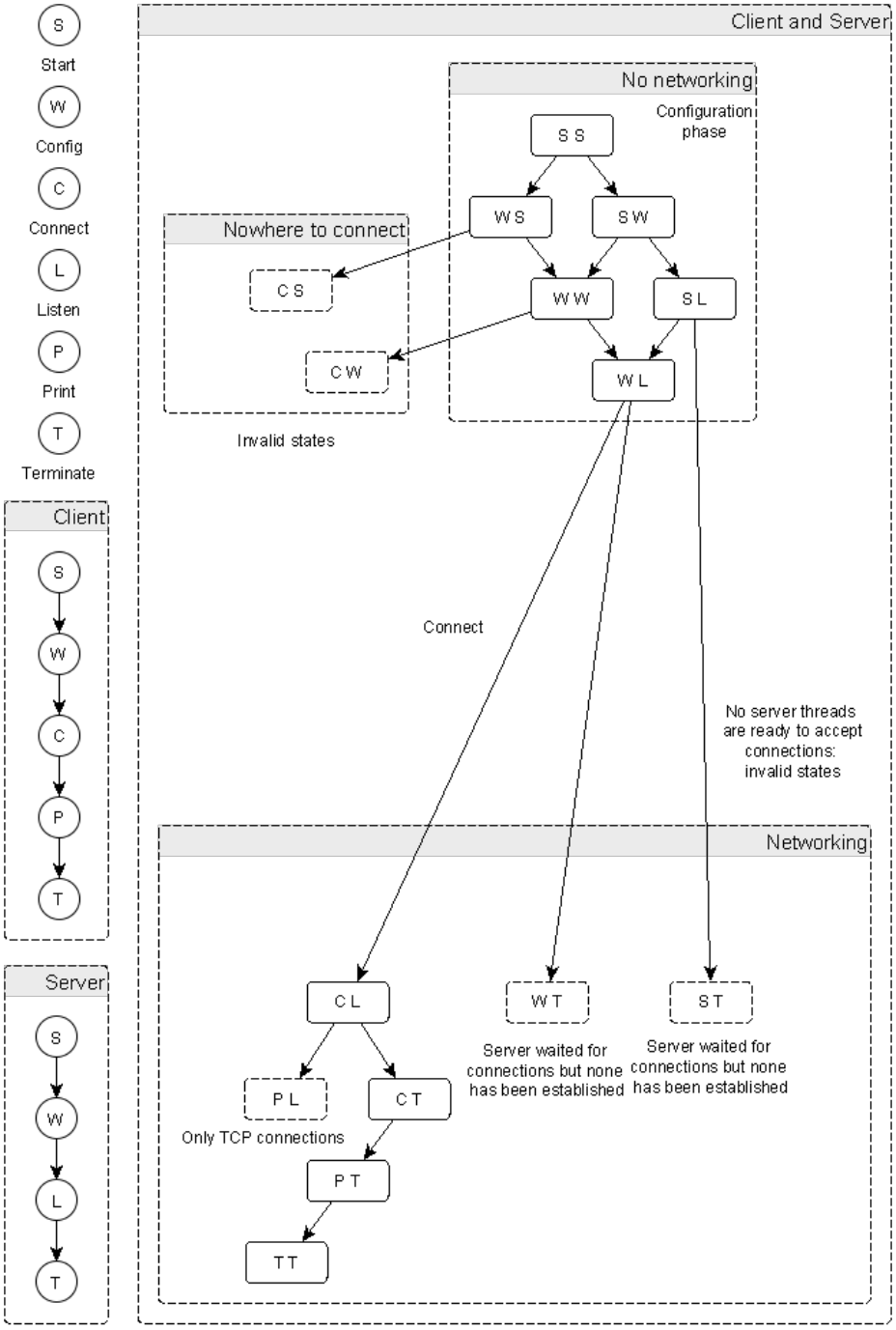

Figure 2 Possible state space reduction techniques when model checking networked applications

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 transformation and reduces the cost 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 a 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).

Because of JPF's no-networking limitation model checking networked applications is not a straight-forward procedure to perform. Verification engineers in need of a formal verification tool which supports model checking networked applications may choose from the two following options.

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 but leaves the program logic intact. 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

the multiple process system into one process. SuT processes are transformed into threads, network communication is often transformed into function calls. After the transformation software verification targets a multi-threaded single process software without the use of networking. PathFinder needs no modification in order to perform the system's model checking [2].

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 unnecessary to modify the SuT source code.

2.2 Primary goal

Before engaging into dependability engineering the following technical terms have to be agreed on:

- This document regards a JVM confined Java server or Java client process as an atomic component. It manages its own memory address range and schedules its threads.
- An aggregation of atomic components constitutes an autonomous component.
- This document uses the terms of *fault*, *error* and *failure* according to the Laprie dependability engineering terminus technicus [3].
- Failure classification follows the Bondavalli-Simoncini taxonomy.

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, Figure 3) 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 this component.

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

- What kind of autonomous components exist in the complex 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 process may continue at the level of autonomous components. As the test 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.

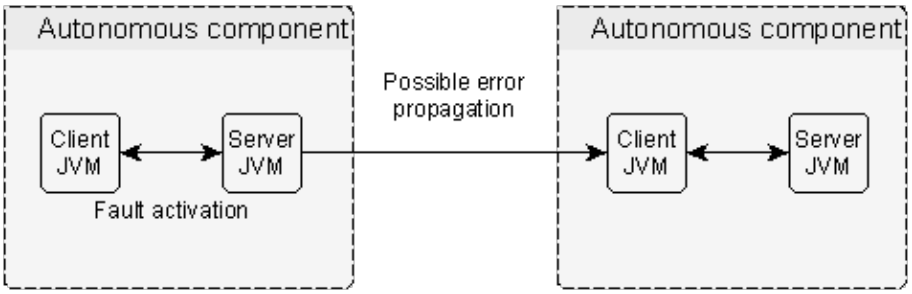

Figure 3 Possible error propagation

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 interacting 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.

2.3 Existing tools

2.3.1 Jpf-net-iocache

The module net-iocache [4] is the most commonly accepted solution still under rapid development (2014). The project enables JPF for client software verification.

Net-iocache forms a cache layer between the client and the server application. The cache layer performs the networking abstraction through model classes which substitute

classes of the java.net package. Caching aside the abstraction functionality also improves the performance of the verification.

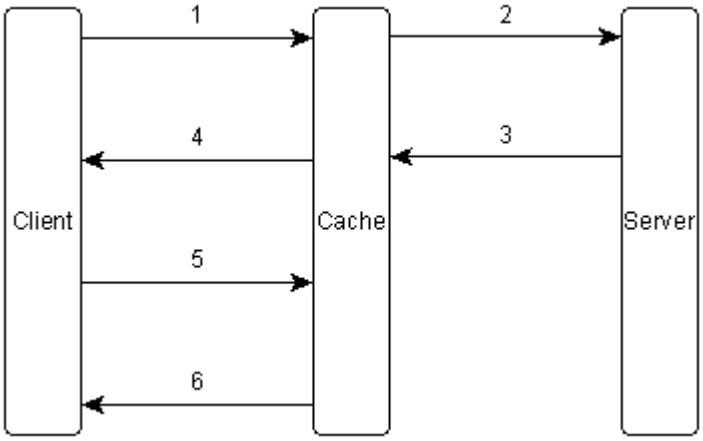

Figure 4 Caching mechanism in net-iocache

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 server applications' model checking.

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 5 Networking interaction example

The client starts two threads. One of them wants 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.

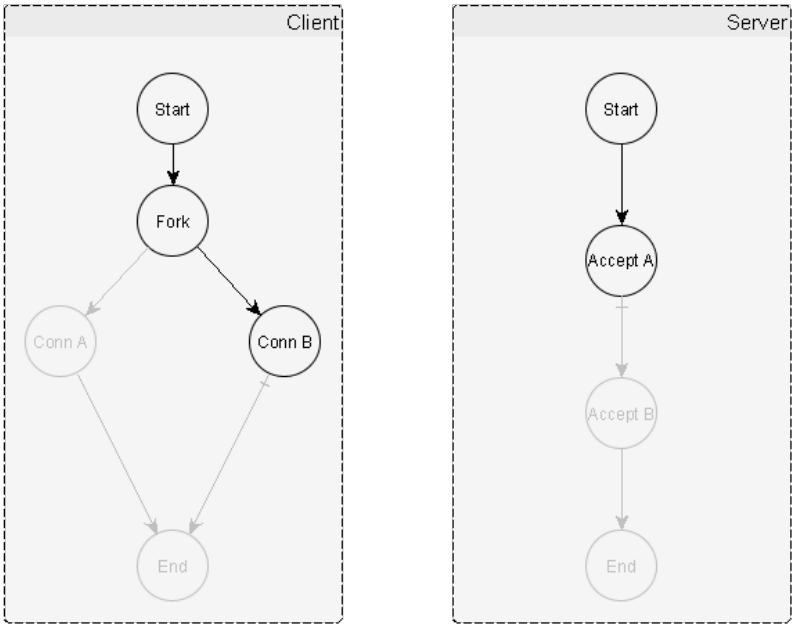

Figure 6 Networking interaction problem

As the figure shows there is a 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 help of net-iocache.

2.3.2 Jpf-net-master-slave

Sergio Feo started the development of a still unpublished JPF module [5] during his PhD studies at Albert Ludwigs University of 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) [6] in 2013 but never published the tool nor prepared its documentation. He was assisted by Prof. Yoshinori Tanabe [7] 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.

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, said David Korsmeyer (Director of the Engineering Directorate at Ames Research Center). 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 has changed to being a standalone model checker. It has become open source in 2005 and has been a fascinating area of academic research ever since [8] [9] [10] [11]. JPF has proven its usefulness during the verification of the K9 Mars rover's control unit, the EO-1 spacecraft's diagnostic system and even at the testing of Fujitsu's Web applications 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.

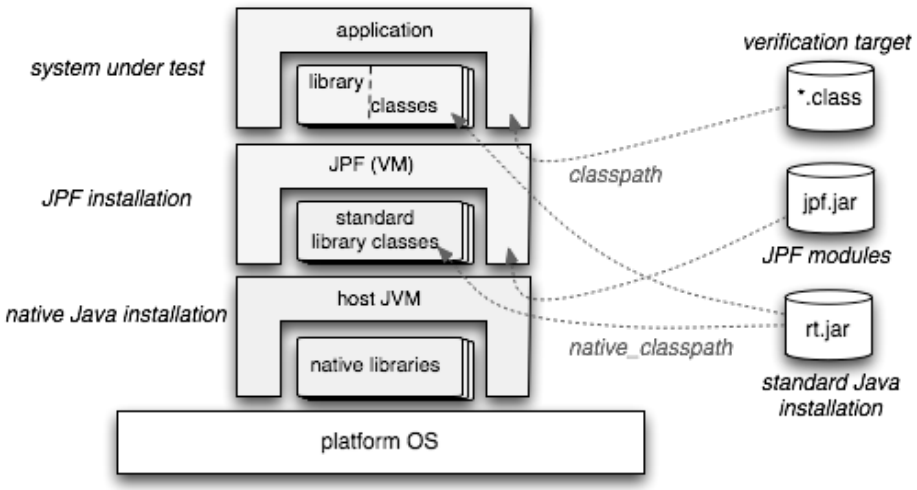

Figure 7 JPF stack [12]

JPF acts like a JVM but implements the Java Virtual Machine specification [13] in a unique way in order to become the SuT's perfect observer [14]. 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 [15]. JPF starts with

reading the bytecode of the SuT and builds up an execution environment for the verification. Each bytecode operation (defined in [13], Chapter 6: The Java Virtual Machine Instruction Set) is modeled by a bytecode class. When JPF so called executes a bytecode the *execute()* function is called. The function simulates the bytecode execution effects by modifying the JPF virtual machine's state. For example, opcode with the mnemonic *ALOAD* pushes a Java reference onto the JPF stack.

3.1.1 PathFinder modules

The software has a modular architecture which makes the core project extendable. PathFinder enables developers to override its behavior by adding custom JPF modules to it. Jpf-net-iocache and jpf-net-master-slave are JPF modules which can be plugged into jpf-core. There are several ways to manipulate JPF by a module; some of the ways are described in Chapter 3 in detail.

Over the years since the JPF has become open source a significant community has formed around it with active developers and end-users from academia and from industry as well. 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 choices

This feature of dealing with non-determinism makes PathFinder able to verify software which utilize some kind of nondeterministic data choice. The example below [16] 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 [16] where the property gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty is violated. The verification stops as soon as a property is violated 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 choice generator. The choice generator is created according to the required data choice. In this example the first random call returns with an integer between 0 and 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 the state space when the software tries to read from a nondeterministic 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 choices

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. Each choice is represented by a thread choice generator which enumerates these threads of *RUNNABLE* [17] 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*. The change in a *RUNNABLE* thread's state may be a cause of the bytecode it executed like entering a 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 non-runnable thread to exit the object's monitor and become *RUNNABLE* again.

3.2.2.1 Partial order reduction

Most formal verification technique has 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 (hereinafter POR) is utilized.

PathFinder provides this feature on-the-fly. When POR is enabled state space exploration collapses a thread's concurrency irrelevant instructions into an atomic transition. By doing so it excludes unnecessary states from the state space by eliminating concurrency irrelevant thread interleaving. POR reduces the state space by approximately more than 70% according to the official calculations [18].

3.3 Model Java Interface

A standard Java Virtual Machine provides an interface called Java Native Interface (JNI) [19] which enables virtual machine interaction with non-Java software. Native methods implement JVM functionalities. These platform dependent modules usually are written in C and compiled according to the actual JVM's processor architecture.

JPF has an API similar to the JNI called Model Java Interface (hereinafter MJI) [20]. It enables verification engineers to intercept native calls, apply any kind of outer World abstraction they intend to and create choice generators according to the intercepted method call.

3.3.1 Model class

Java software usually utilize classes provided by the JVM 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.

JPF has its own classloader and class loading mechanism. 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. 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 it continues the lookup on the classpath of the host JVM (remember Figure 7).

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 (eg. the class 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 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 JVM.

The class java.io.OutputStream requires the following lines to be present:

```
…
public void write(byte[] b, int off, int len) throws IOException {
     …
     native_write(b, off, len);
     …
}
…
private native int native_write(byte[] b, int off, int len) throws
IOException;
…
```

The native call is matched to a native peer class according to MJI name mangling [21] 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 level. The interceptors are Java classes which follow a special naming convention called MJI name mangling [21] similar to the JNI name mangling [22]. These classes are referred to as native peers; they are executed by the JVM as part of JPF.

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. 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 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);
…
```

The example native invocation discussed in section 3.3.1 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.

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 are to subscribe to 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

JPF modules are able to subscribe to jpf-core's internal JVM events. VM listener may handle choice generator, concurrency detection, bytecode execution, and thread scheduling related events declared by the interface *VMListener* by implementing it.

3.5 Search object

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

As soon as JPF initialized its simulated JVM it instantiates a search object based on the configuration and calls its *search()* method. During Jointstates' development an undocumented limitation has been identified and it has been confirmed by Peter Mehlitz [23] (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 the certain parameters (bytecode arguments) if the equivalent opcode has been recognized as the next one in the class file. Model class instances are interlinked as if they were located in a class file, so it preserves the original bytecode sequence.

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

3.7 JPF attribute

JPF objects such as bytecode instances have a special variable called attribute. Attributes enable module developers to extend JPF objects with custom information. The type of information is uncertain so attributes have the most general type (Object).

4 Transparent client-server co-verification

4.1 Concept

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 presumption about the client and the server software.

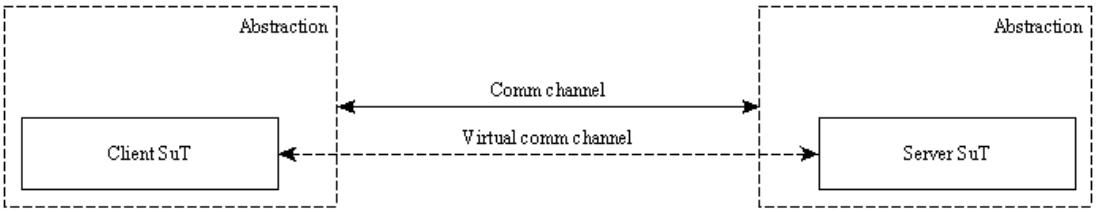

Figure 8 Applying networking verification abstraction between client and server

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 by an exhaustive method. The proper abstraction logic may alter this state space to a finite form introducing a verification presumption. 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 alter the correctness of the verification.

The open-source software product of the research this thesis documents is the JPF module called jpf-jointstates [24] (hereinafter Jointstates). Jointstates is hereby able to observe causality relations between networking interactions. The figure below demonstrates the kind of causality relationship between message A and B.

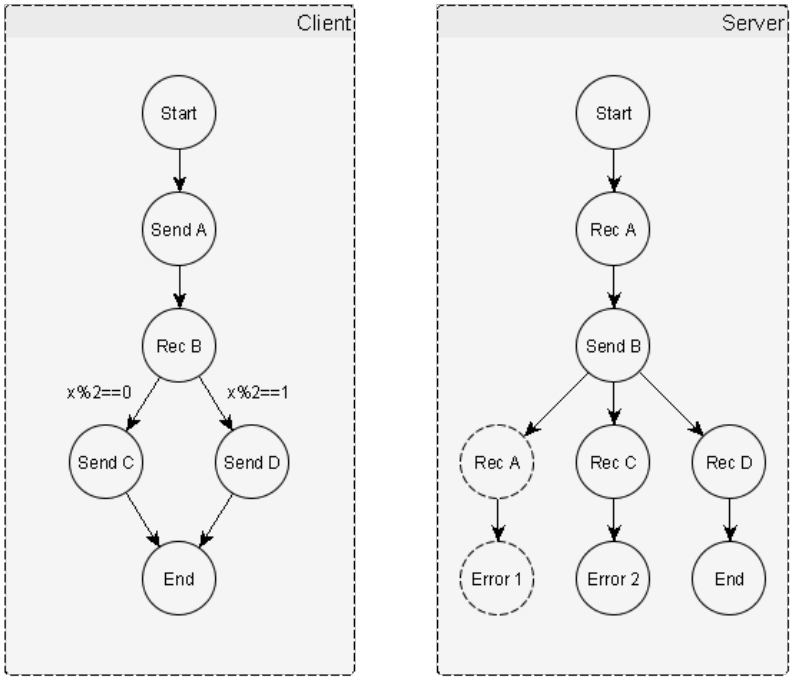

Figure 9 Interaction causality on the client side

Variable *x* on the client side represents a random value like System.currentTimeMillis() function's return value. The 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 allow 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.

The necessity of proper network abstraction is depicted in the figure below.

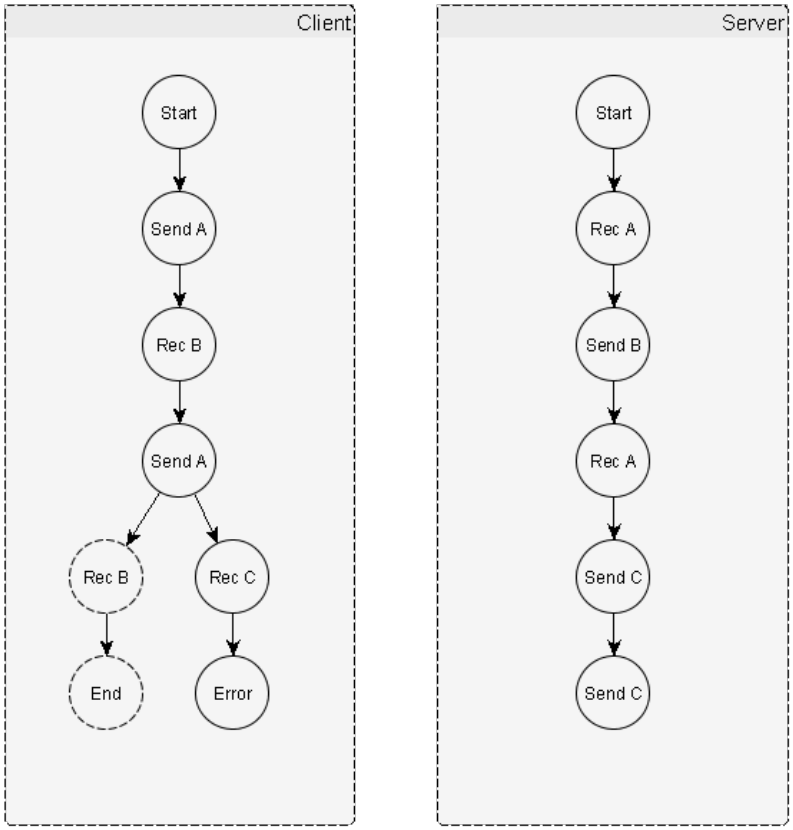

Figure 10 Interaction causality on the server side

The client sends message A and awaits a response B. The server does not return to the state before receiving message A (eg. Increments a variable). The client sends the message A again to the server. Jpf-net-iocache would return the response B to the client but the server actually responds C. Net-iocache would falsely finish the verification showing no sign of any error. Jpf-jointstates tends to find 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

4.2.1 The big picture

The simplified verification topology is depicted in the figure below.

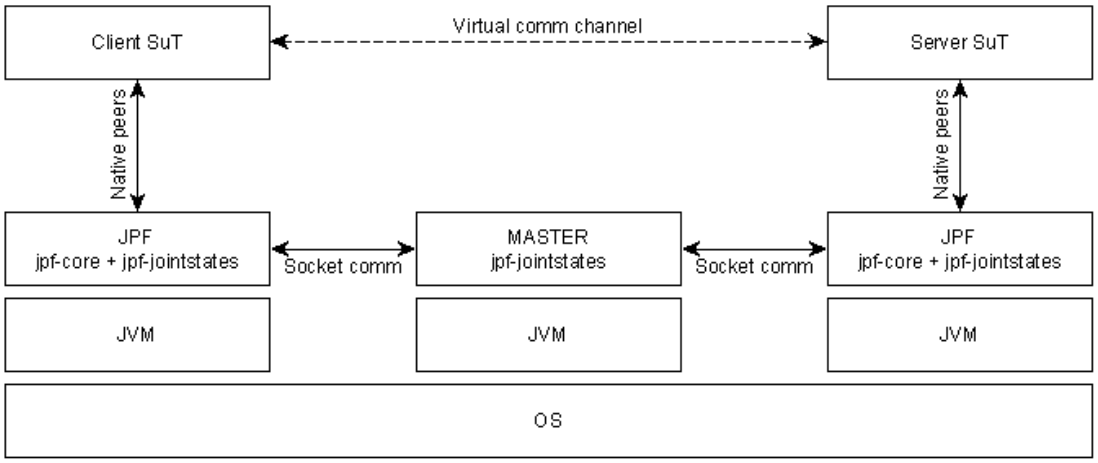

Figure 11 Jointstates architecture

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

4.2.2 Verification algorithm

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

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

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

Figure 13 Client pseudo code

```
var msg1 = RECEIVE
PRINT msg1

var msg2 = RECEIVE
PRINT msg2
```

Figure 12 Server pseudo code

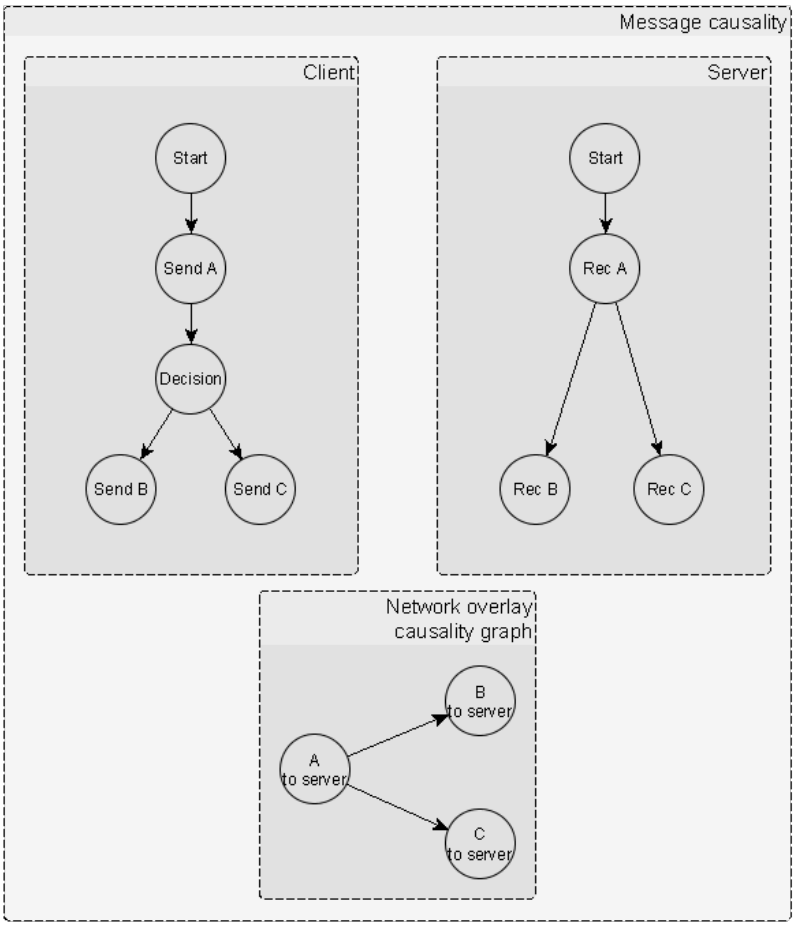

Figure 14 Message causality overlay’s connection to the SuT’s state space

Be aware that Jointstates does not verify the system’s correctly-timed nature defined by Bondavalli-Simoncini [14] due to the verification mechanism’s start time-synchronization; it checks correctly-valued aspects only.

4.2.2.1 Joint states

Joint states are utilized for message causality management during model checking. It consists of two process states; the client SuT verification state and the server SuT verification state. The two process states are managed by two separate Java PathFinder instances as seen in section 4.2.1. The SuT verification states are linked to

each other by a joint state identifier (hereinafter JSID). Joint state creation occurs when the verification algorithm identifies a new message type on the actual joint state depth (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. 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. A data choice generator is created and both scenarios are checked by Jointstates.

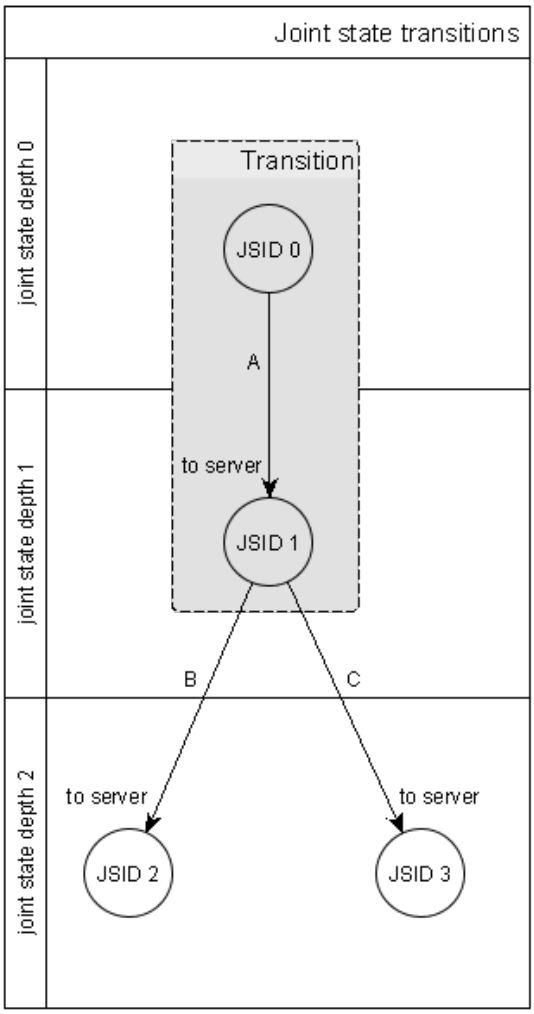

Figure 15 Joint state transition of a joint state

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 27) while slaves run the networking capable JPF verification algorithm stepped

according to the master's synchronization commands. The slaves run a heuristic algorithm which implements the state space exploration logic. The heuristic combines breadth-first search (hereinafter BFS) and depth-first search (hereinafter DFS) algorithms.

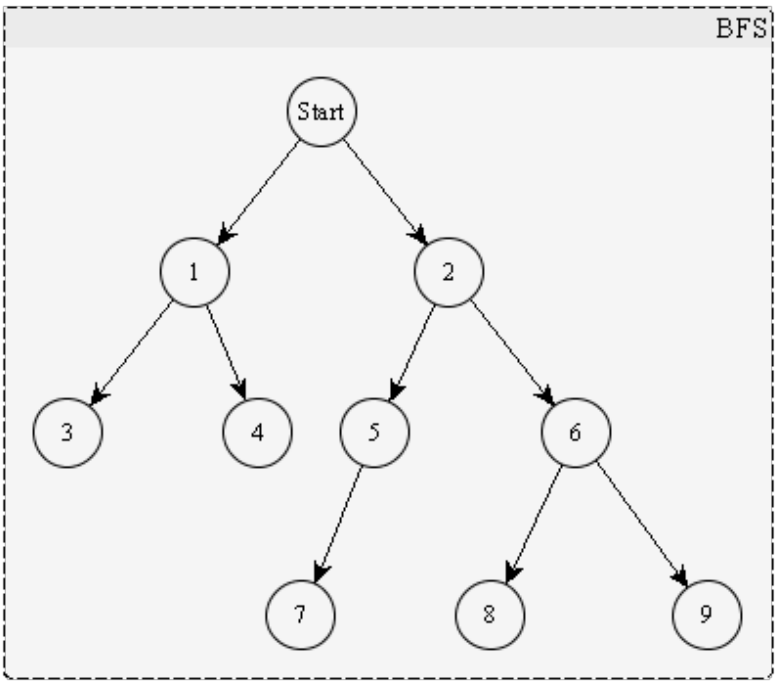

Figure 16 BFS state space exploration

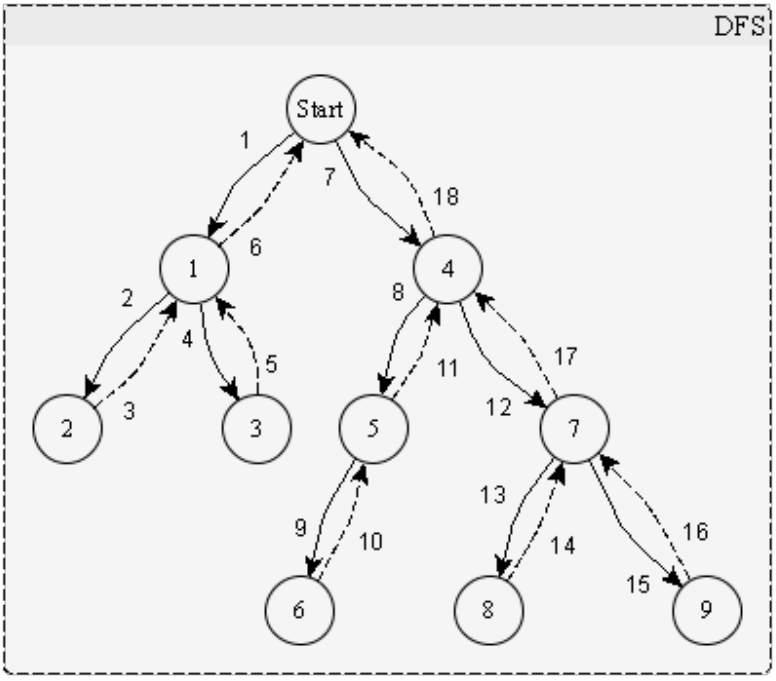

Figure 17 DFS state space exploration

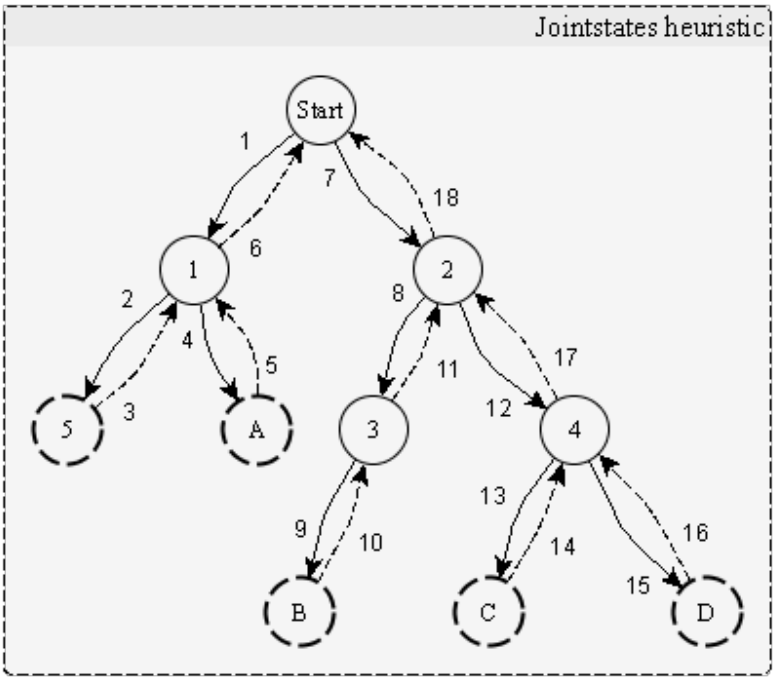

Figure 18 Jointstates state space exploration (simplified version)

The following pseudo code represents the Jointstates slave heuristic search algorithm.

```
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 18. The algorithm enqueues the joint state into a prioritized first-in-first-out (hereinafter FIFO) queue and backtracks as if DFS reached the bottom of the graph. When the DFS

ends (in the Start state at first), the BFS dequeues the next joint state and starts a DFS started on the dequeued state (marked with *5*). The DFS just started explores normal and joint states below this state. When finished the 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, receiver side model checker instance requires the possible inputs to create a data choice generator with the possible choices. Use of a prioritized BFS state 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 as an ancestor, which is state *B*.

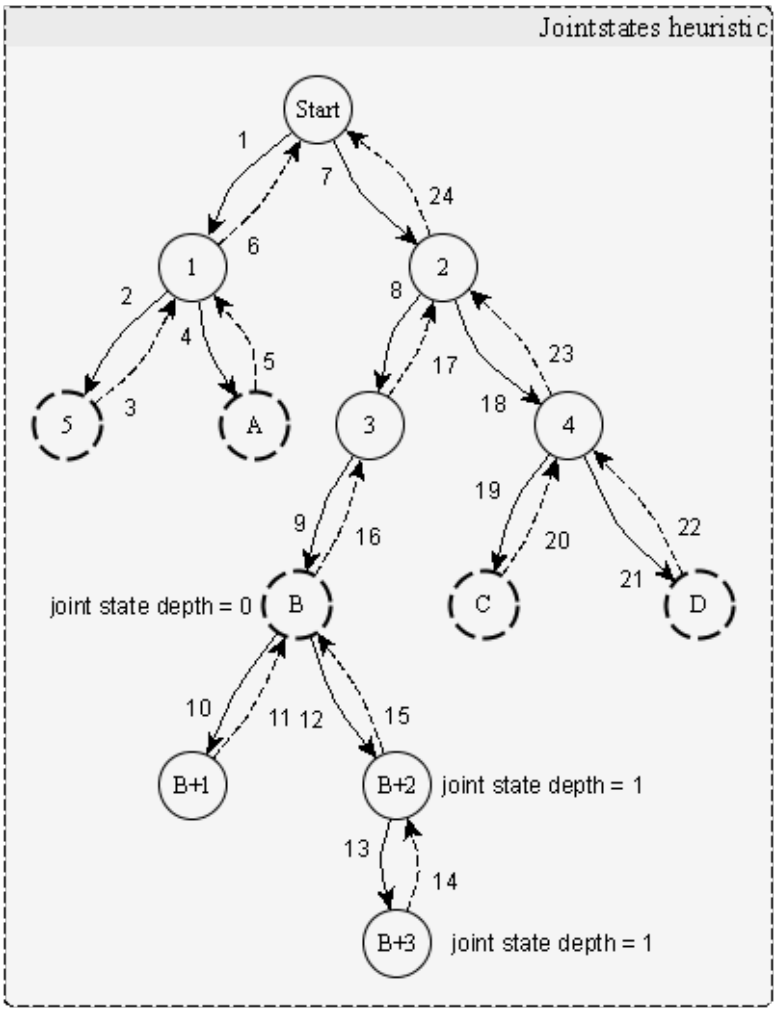

Figure 19 Joint state depth

The algorithm enqueues states with the same priority if has the same joint state depth. According to BFS, the higher is the joint state depth, the lower is the priority. States enqueued into the same joint state depth are separated into two sub-priority groups; a write state group and a read state group. Because of write operations does not, but read operations does require information from the outside World, write state groups have higher sub-priorities compared to read state groups found on the same joint state level.

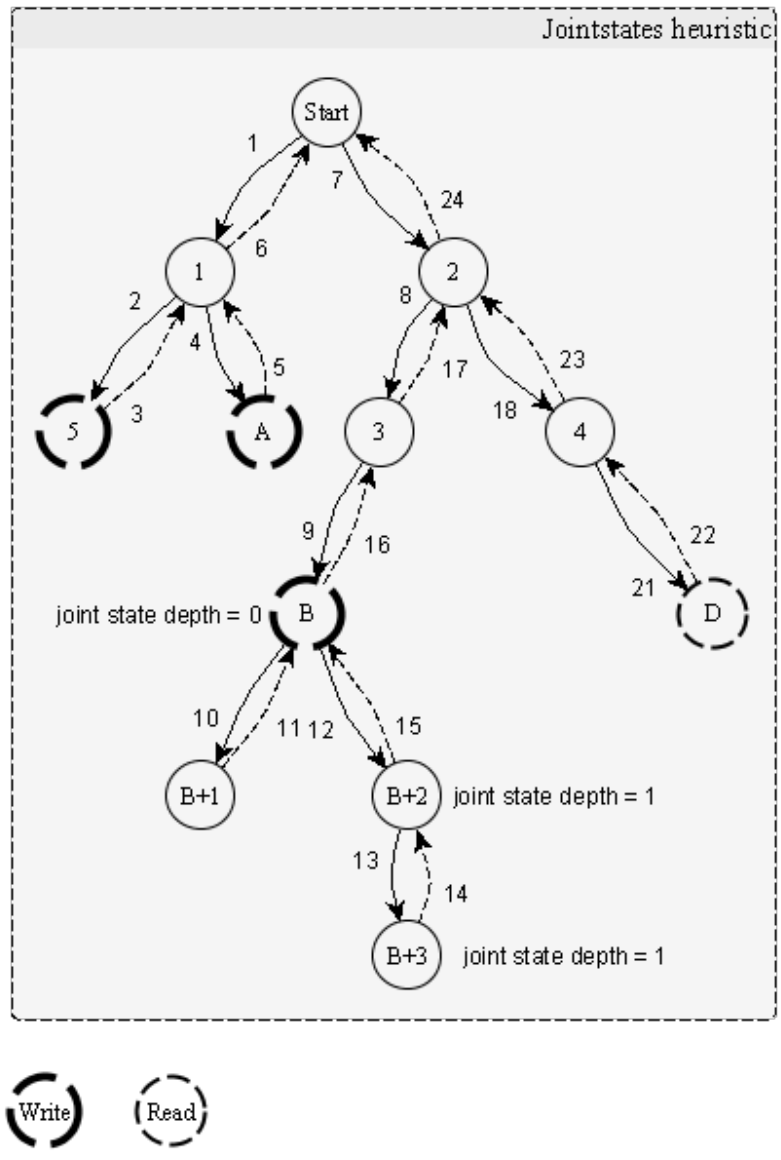

Figure 20 Write and read state groups

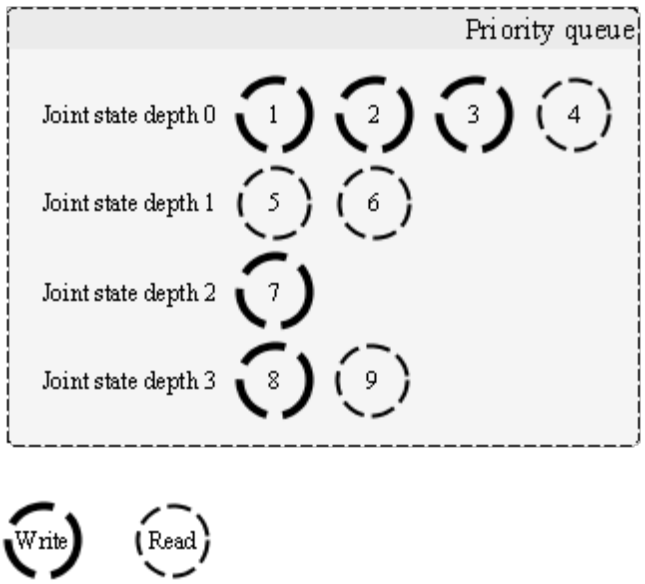

Figure 21 Priority queue's dequeued state order

4.3 Implementation

4.3.1 Jointstates' modules

Figure 11 shows only a glimpse about the architectural concept. The figures below extend it with general and JPF specific implementation data about Jointstates' architecture.

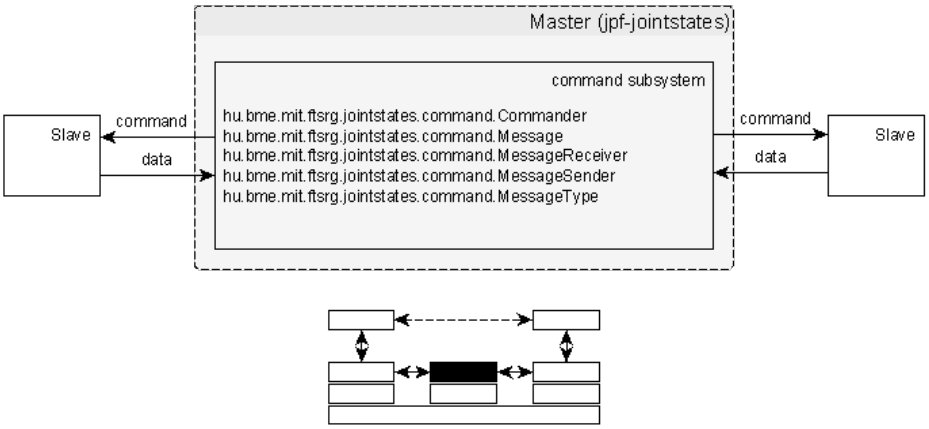

Figure 22 Master module

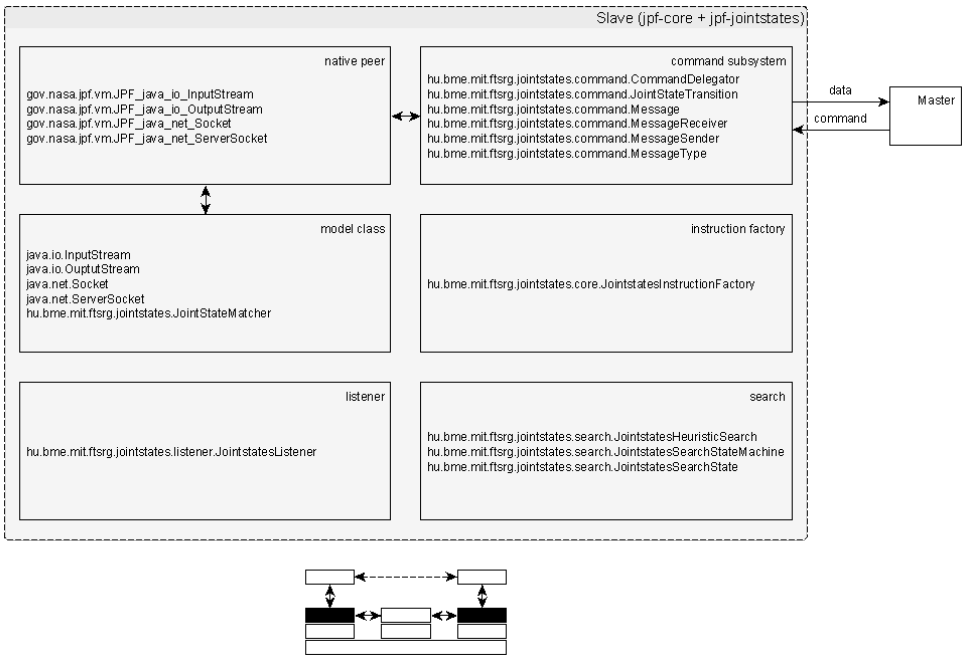

Figure 23 Slave modules

4.3.2 JPF components

As described in Chapter 3 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 classes

Model classes override the networking related classes provided by the JRE. 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 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 [25]. Stores the 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 the 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`

Handles socket creation.

4.3.2.2 Native peers

Native peers capture model class native calls. Message transfer occurs between a slave and the master instance when the 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 JVM level by native peers. The 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 the 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. In order to pass the invocation to the JVM the following native method declarations are necessary in 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);
```

Please note that the return type integer has different meaning for the read and for the 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 knows this difference by the native method return type declaration located in the model class.

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. *INVOKEVIRTUAL* bytecode instances which perform socket read or socket write operations are flagged with a unique JPF attribute (described in section 3.7).

4.3.2.4 Listener

Jointstates has a JPF listener called *JointstatesListener*. The class is instantiated when jpf-core initializes the configured external JPF modules. The listener's 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 a bytecode instance (of the SuT). 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. Regarding *BreakGenerator* has no 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 support for heuristic verification search strategies 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. The exploration is performed by the method *generateChildren()* which generates the subordinate states of *[root]*. When a state is generated it is enqueued to 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..

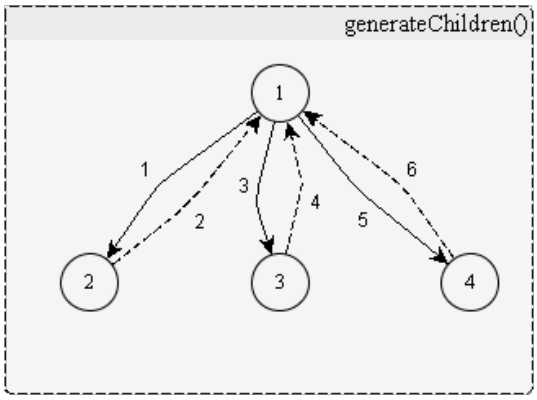

Figure 24 Heuristic state generation

GenerateChildren() is implemented by jpf-core but *JointstatesHeuristicSearch* wraps it by method override. The children generation of a read or write state 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).

0	normal states	↑
1		
…		
MAX_INT-OFFSET-1 → [root]		
MAX_INT-OFFSET	write state, joint state depth 1	↓
MAX_INT-OFFSET+1	read state, joint state depth 1	
MAX_INT-OFFSET+2	write state, joint state depth 2	
MAX_INT-OFFSET+3	read state, joint state depth 2	
…	write states & read states	
MAX_INT-1		
MAX_INT		

Figure 25 State priority assignment

4.3.3 Message transfer system

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

Figure 26 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. The synchronization algorithm is represented by the following state chart diagram.

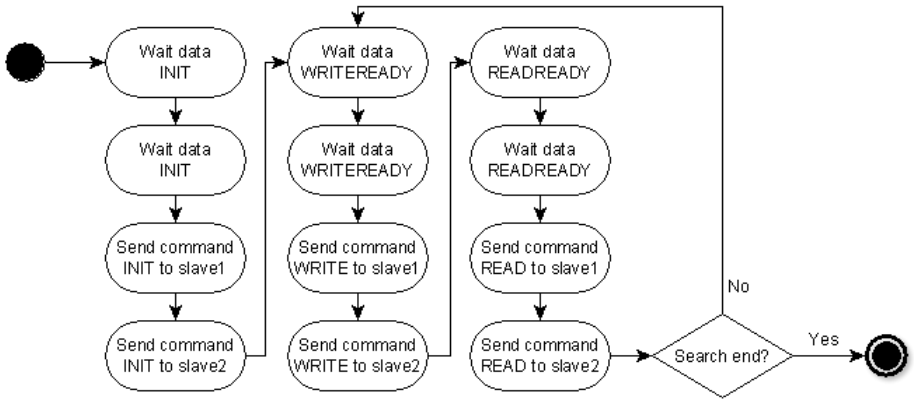

Figure 27 Synchronization algorithm

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 20 and Figure 21 by joint state depths until the distributed search ends.

The 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.

The master contains a component which is the center of provided verification data called *Aggregator*. This class collects joint state transitions on each joint state depths 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 when one SuT performs a write operation which is intercepted by *JPF_java_io_OutputStream.* The 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 joint state identifier the SuT gained by sending the message which is saved by the model class *JointStateMatcher*. Its model class behavior attaches the joint stated identifier to the JPF verification state which is backtracked, saved and restored from time to time.

QUERY occurs when the slave hits a read operation. The native peer *JPF_java_io_InputStream* starts a *QUERY* operation on the *Aggregator*. By doing so, the slave provides the current joint state identifier to the *Aggregator* which responds by 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, the next joint state identifier is extracted from the selected transition and saved into the model class *JointStateMatcher*. At the end, the read operation returns with the transition provided message.

5 Example

Jointstates' source code [24] provides three client-server pairs. One of them represents a fault-free system (*DummyClient* and *DummyServer*), the other two contains a concurrency 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 clients sends an integer, and the server returns it multiplied by ten. The other two systems are based on this one.

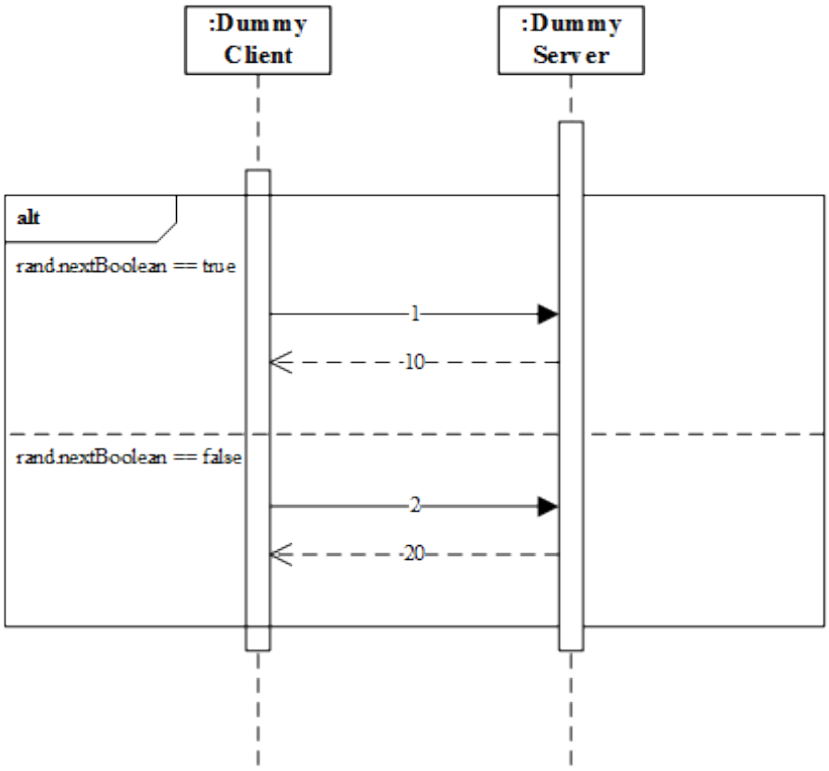

Figure 28 Dummy system behavior

The test output is the following.

```
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. On a control branch, if the client receives an unexpected integer it enters its own Java monitor by calling *this.wait()*. This should be considered a concurrency bug regarding there are no threads alive to issue *notify()* or *notifyAll()* if Jointstates 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. The *Lucky* system's behavior is depicted on the following diagram.

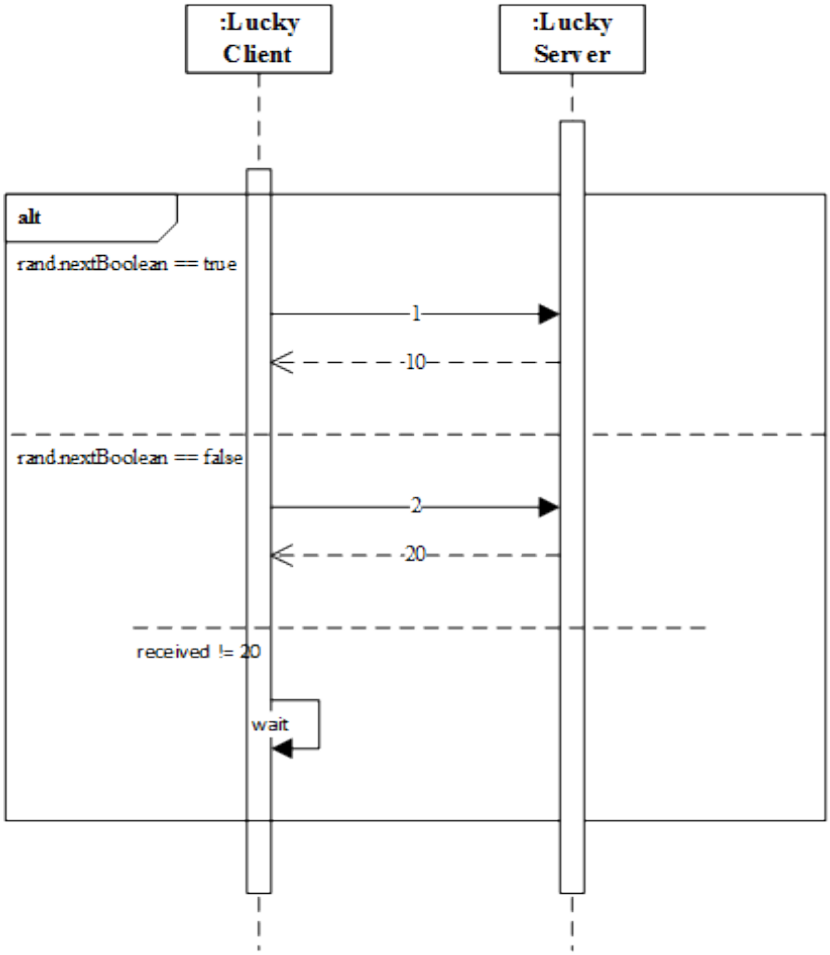

Figure 29 Lucky system behavior

When Jointstates verifies the system it finds no bugs as expected.

```
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 scenario among the examples is the case of *Buggy* system. 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 30.

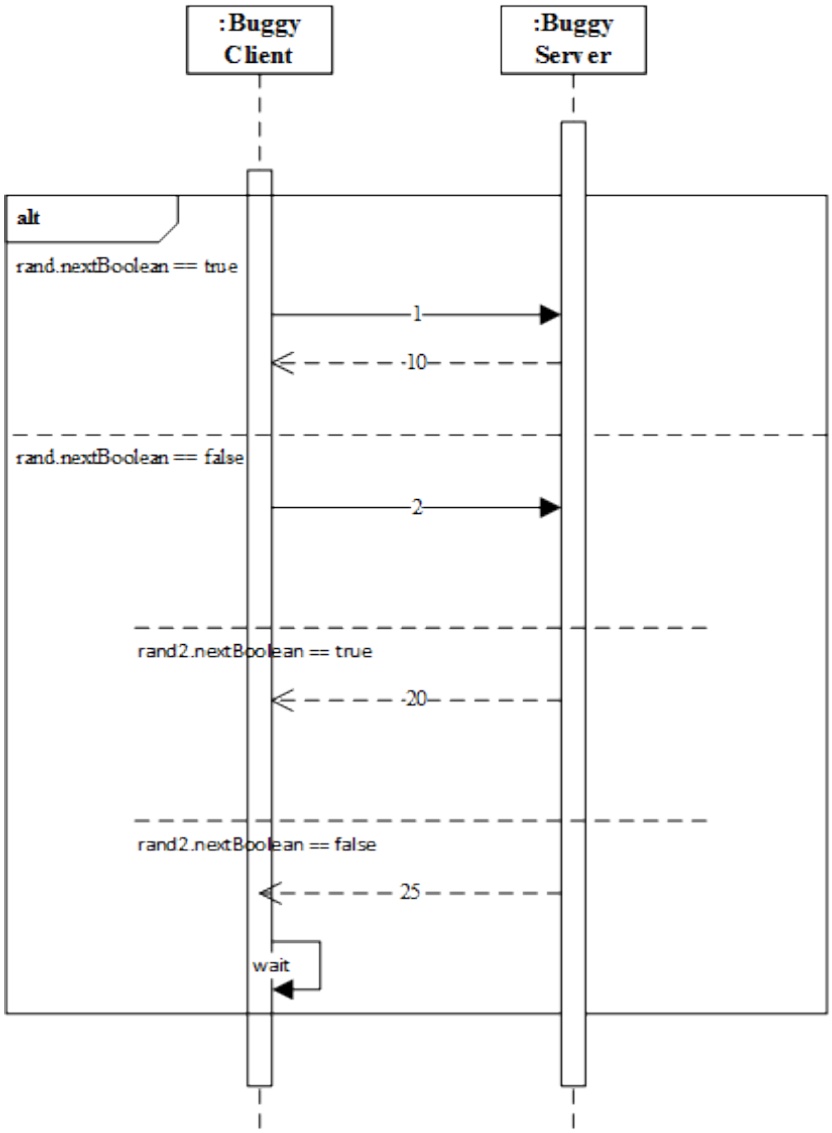

Figure 30 Buggy system behavior

Jointstates finds the bug and prints the source code lines which cause 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 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. 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 data.
- Client and server must communicate through native Java sockets and exchange integers only. This limitation can be overcome by further feature development.
- The verification is ineffective compared to Net-iocache which 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 can be the bound of the verification.
- InputStream and OutputStream must not be used for other than socket networking purposes by the SuT. 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 wait for *END* messages from both sides to stop its execution.

6.2 Strengths

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 2 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

- If Jointstates has found an error, the message trace should appear near the violated property as part of the counterexample.
- The concept is not confined to client-server verification. The implementation expects two communicating sides. Jointstates does not assume a strict client-server communication where only the client may start the communication. A generalization of the implementation should lead to a solution which is able to verify a peer-to-peer based system.

References

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

[2] 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.

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

[4] 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].

[5] 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].

[6] 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].

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

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

[9] 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.

[10] W. Leungwattanakit, “I / O Cache Implementation Guide ⋆ Table of Contents,” 2012.

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

[12] 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].

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

[14] 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.

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

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

[17] Oracle, “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].

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

[19] Oracle, “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].

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

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

[22] Oracle, “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].

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

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

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