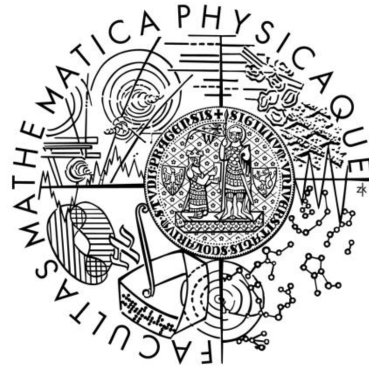
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Štěpán Vávra

Debugger interface for Java PathFinder model checker

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Jančík

Study programme: Informatics
Specialization: Software Systems

Prague 2013

Acknowledgements

First and foremost, I offer my sincerest gratitude to my supervisor, Pavel Jančík who guided me throughout this thesis. His friendliness, wisdom and personal availability were the key thing that allowed me to enjoy the whole time. Without his patience at first and then dedication and support, I would not be able to complete this work.

Peter Mehlitz, from NASA Ames Research Center, was always supporting me and encouraging me in every email. It really helped me especially at the beginnings.

At work, I have been blessed with many friendly colleagues. My boss, Tomáš Vala, has never showed any unpleasantness with my regular unavailability at work. He has understood school has had sometimes a higher priority than work and yet he put trust in me and he was giving me more responsibility.

I am also grateful to Chris Pierson, my dear friend, who was willing to refuse comfort and peace at home and came here to Czech to serve God. He is helping me continuously with my English skills even though he does not know that sometimes.

I would like to express gratitude to my parents in law. They accepted me as a member of their family and they provided me and my wife with a roof over our heads. Without them, I would not be able to continue in my studies.

My parents raised me with love and care. They created a foundation of who I am today.

Finally, I thank my beloved wife for who she is; for her endless love, patience and support. Without her, I would not be able to go through my studies nor to finish this thesis. She and our baby daughter make my life complete.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional resources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, November 23, 2013 ....

Štěpán Vávra

Název práce: Debugger interface pro Java PathFinder model checker

Autor: Štěpán Vávra

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Pavel Jančík, Katedra distribuovaných a spolehlivých systémů

Abstrakt:

Cílem této práce je začlenit Java PathFinder do architektury Java Platform Debugger Architecture, tedy umožnit debuggování Java Pathfinderu z jakéhokoliv moderního Java vývojového prostředí se všemi výhodami s tím spojenými stejně jako u běžných Java virtuálních strojů. Těmi jsou kupříkladu různé typy breakpointů, krokování v otevřených zdrojových souborech a inspekce zásobníku volání a objektů v programu.

Výsledná práce dává uživatelům k dispozici plnohodnotnou škálu možností debugování jako u běžných Java programů, a to bez nutnosti použití dalších nástrojů, editorů a zejména bez nutnosti složitého nastavování prostředí pro debugování. Díky tomu mohou uživatelé zkoumat, debugovat a pochopit stav programu při procházení error trace vyhodnocené Java PathFinderem.

Klíčovou části práce je implementace Java Debug Wire Protocol agenta jako doplněk Java PathFinderu. V důsledku tohoto se Java PathFinder stává kompletnějším Java virtuálním strojem jak v očích komunity tak i celé obce Java uživatelů.

Klíčová slova:

debugování, Java PathFinder, Java Debug Wire Protocol agent, Java Platform Debugger Architecture

Title: Debugger interface for Java PathFinder model checker

Author: Štěpán Vávra

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Jančík, Department of Distributed and Dependable Systems

Abstract:

The aim of this work is to integrate Java PathFinder into Java Platform Debugger Architecture. That is, to allow using Java PathFinder instead of a common Java Virtual Machine for the purpose of debugging Java applications in any modern Java Integrated Development Environment with all its advantages such as various kinds of breakpoints, direct stepping in opened source files, and call stack and object inspection.

The resulting work provides users with all the features they are used to while debugging Java applications. None of this requires any external tools, editors or a complicated setup. Therefore, users are able to view, debug and understand the program state while replaying an error trace in Java PathFinder.

The key part of the study is an implementation of the Java Debug Wire Protocol Agent as an extension for Java PathFinder. That makes JPF more complete as a Virtual Machine in the eyes of the community and the Java users in general.

Keywords:

debugging, Java PathFinder, Java Debug Wire Protocol Agent, Java Platform Debugger Architecture

Introduction 1
Motivation 1
Goals 2
Contributions 2
Thesis outline 2
1 Background 4
1.1 Java Platform Debugger Architecture 4
1.1.1 Debugging Interfaces 5
1.1.2 JDWP Specification 6
1.2 GNU Classpath JDWP 10
1.3 Java PathFinder 10
1.3.1 State space traversal 11
1.3.2 JPF internals 12
1.3.3 Extending JPF 13
2 High-level analysis and comparison 14
2.1 Direct JDI to JPF solution 14
2.1.1 Realization Idea 14
2.1.2 Rejection and Reasoning 14
2.2 JVM TI for JPF solution 15
2.3 JDWP for JPF solution 15
2.3.1 GNU Classpath JDWP based solution 16
2.3.2 Acceptance and Reasoning 16
3 JDWP for JPF analysis 17
3.1 JDWP Specification ambiguity 17
3.1.1 Modifiers 18
3.1.2 Data hierarchy 19
3.2 State space traversal implications 19
3.2.1 Method associated vm state change 20
3.2.2 Thread associated vm state change 20
3.3 JDWP Agent 21

3.3.1 Communication layer 24
3.3.2 Commands, Event Requests and Events 25
3.3.3 Execution and Suspension 26
3.3.4 ID management 27
3.3.5 Integration of JDWP Agent into JPF 28
4 JPF JDWP Agent architecture and design 31
4.1 Processes interaction 31
4.2 Control flows 32
4.3 Execution and initialization 34
4.4 JPF integration 35
4.5 Class model 36
4.5.1 Commands 36
4.5.2 Event Requests 36
4.5.3 Events 37
4.5.4 Values 38
4.5.5 Identifiers 38
4.5.6 Locations 41
4.5.7 Exceptions 41
4.6 Configuration 42
4.6.1 Logging 42
4.6.2 Testing 43
5 Critical assessment 44
5.1 Reusability 45
5.2 Performance 45
5.3 Related work 45
5.3.1 JPF Inspector 46
5.3.2 Debug 4 JPF 46
5.3.3 Eclipse JPF 47
5.3.4 Java Debug Trace 47
6 Further work 48

6.1 Enhancing JPDA 48
6.2 Other work 48
Summary and Conclusion 50
List of Figures 51
List of Tables 52
List of Abbreviations 53
Appendix 54
I Links 54
II User manual 54
II.I JPF JDWP Agent invocation 54
II.II General use 55
II.III Using Eclipse IDE and Eclipse JPF 56
III Parallel work 57
III.I Eclipse JPF project 57
III.II JPF JDWP Integration Tests project 58
IV Running example 59
V CD content 59
VI Code statistics 60
Bibliography 61

Introduction

The intention of this thesis is to make Java PathFinder compliant with the Java Platform Debugger Architecture (JPDA); that is, to analyze whether it is possible to integrate Java PathFinder into JPDA; to investigate, to propose and to defend the best solution; and finally, to implement it. The results of this work should provide JPF users with the same debugging experience of the JPF verification as a standard Java debugging use case does.

As the objectives of this study became clearer, the title of this study evolved to *JDWP for JPF* and then further to *JPDA for JPF*, which are another titles this study is addressed by.

Motivation

Java PathFinder even as a standard VM contains a lot of valuable information about the program state. Such an information is even more valuable when the state, or precisely the sequence of states, leads to an error, which is so rare in its occurrence, so hard to reproduce and, finally, so important to solve. This is exactly what JPF is meant for. NASA started this project in order to invent verification methods for software where a failure is not an option. However, without a user-friendly UI, the majority of Java aware users is not able to profit from the details of such information. It is too much to ask them to learn JPF internals and to gain an ability to understand what the state of JPF internal entities says about the program they are verifying.

The question is what is the right UI to inspect a Java program state? A reasonable and an expected answer would be *a debugger of a particular Java IDE*. The ability of all the debuggers to work with the same JVM relies on the debugging architecture, called JPDA, proposed by Sun engineers. It was available since the second version[1] of Sun JVM [1] and yet it proved to be good enough for today's requirements of debugging as of the Java SE 7 [2].

Therefore, introducing JPF to the JPDA provides a solid, well-specified and durable foundation for an inspection of a program being verified in JPF from any debugging tool[2]. Moreover, it provides a foundation for the development of new features for manipulation and observation of JPF verification.

[1] The Java versioning convention changed in time [55] . The version 1.2.2 is nonetheless referred as the second version of Java [56].

[2] The tool has to be however JPDA compliant.

Goals

The goals of the presented thesis are to

- Investigate suitable possibilities for an integration of JPF into JPDA
- Search for options to reuse already existing projects so that the study can focus on JPF specific problems
- Implement the solution
- Verify its usefulness by a tighter integration of the results with one common IDE

As the formal verification method to determine whether the objectives of this work were met, we established implementation coverage of the JPDA specification. However, as we found in this study, the specification is not always precise as it sometimes neglects information. Therefore, the interpretation of the specification needs to be verified too. As a consequence, we also prepared integration tests to support the outcome of the formal verification.

Contributions

So far, Java PathFinder model checker has not been integrated into JPDA; therefore, the most significant contribution of this thesis is making Java PathFinder JPDA compliant.

With JPF fitting into this Java standard debugging architecture, users benefit from the ability of using modern Java IDEs for a graphical presentation of a program state, while it is being verified in JPF.

Finally, this study opens new possibilities for the development of UI extensions for the control of the JPF verification.

Thesis outline

In chapter 1, we describe this study more in depth as well as other relevant projects and we introduce the technologies this work is related to. There is also JDWP specification explained which is necessary in order to understand the main chapters.

The high-level analysis of possibilities of integration of JPF into JPDA and the decision to implement the JDWP specification is shown in the chapter 2.

In the JDWP for JPF analysis, chapter 3, we elaborate JDWP specification ambiguities that are important for a complex understanding of the specification. We also introduce the JDWP agent concept and we decompose this concept when it is being confronted with JPF. The purpose of this chapter is to understand what the requirements on the JPF JDWP agent are.

The architecture and design we chose; implementation details we had to solve; the synthesis of the JPF JDWP agent problem is described in the chapter 4.

In the critical assessment, chapter 5, we evaluate the detailed results and achievements of this work. We also compare them with related projects.

Finally, we propose further work in the chapter 6.

Two other projects we made in parallel to support this study are listed in the appendix. A user manual is there in the section Appendix II.

1 Background

This chapter demonstrates wider appreciation of this work and provides further motivation. It also provides a high-level description of technologies, specifications and projects that are important for the comprehension of this study.

1.1 Java Platform Debugger Architecture

The Java Platform Debugger Architecture [2], abbreviated as JPDA, describes an architecture and the interaction of components that are involved in debugging of Java programs. The JPDA is structured as follows in Figure 1.

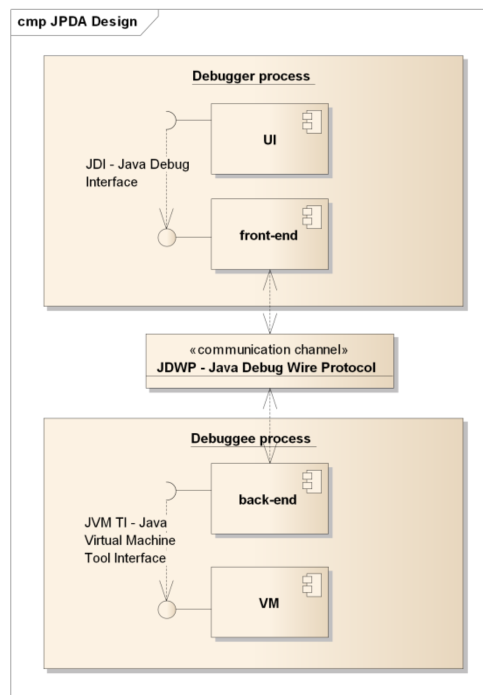


Figure 1. The JPDA structure

The intent of JPDA is to provide standard interfaces and a protocol in order to allow easy development of Java debugging tools, regardless of the platforms specifics such as operating system, hardware, or native implementations of the VM. Additionally, the goal is to provide a complete architecture that targets debugging between different platforms and/or with different versions of JVM.

1.1.1 Debugging Interfaces

The JPDA is a multi-tiered architecture that consists of two interfaces (JDI, JVM TI), a wire protocol (JDWP) and two components that connects them together (front-end and back-end). The Java Debug Interface (JDI) is a high level Java interface providing an abstraction over VM entities and the way to manipulate with them. The Java Debug Wire Protocol (JDWP) defines a communication protocol between the JPDA front-end and back-end. Finally, the Java Virtual Machine Tool Interface (JVM TI) (a successor of Java Virtual Machine Debug Interface (JVMDI)) defines the API the VM must provide for debugging.

The back-end of a debugger runs in the same process as the target VM. It is responsible for a bidirectional translation of the information being transmitted from the debugger (e.g., over JDWP) to the VM (e.g., through JVM TI).

The debugger front-end runs in the same process as the debugger. It implements the high-level Java Debug Interface (JDI) through which it provides the information from the target VM (e.g., from the low-level JDWP).

1.1.1.1 Java Debugger Interface (JDI)

The Java Debugger Interface [3] is 100% pure Java language API that defines an abstraction of the target VM and requests to manipulate with them as well as with the VM itself at the user code level.

It is designed to be used by debuggers as it greatly facilitates the JDWP information, which is abstract to some extent, and translate into a Java class model for an easy of use in Java development environments. The top Java IDEs [4] Eclipse, Netbeans, IntelliJ Idea use JDI [5][6][7] to debug java programs. Nevertheless, the debugger implementations may also use JDWP or even JVM TI directly.

1.1.1.2 Java Debug Wire Protocol (JDWP)

The Java Debug Wire Protocol defines the communication protocol between the debugger (the front-end) and the target VM, the debuggee process (the back-end). Precisely, it is the format and the semantics of the serialized bit stream being transmitted over the communication channel. It does not however specify a transport mechanism (e.g., shared memory, sockets, etc.) to be used for the communication.

JDWP defines convenient constants (e.g., *type tag* constant for long primitive type) and the data format (e.g., an executable *location*, which is determined by: *type tag*, *class ID*, *method ID* and eight-byte *index*). Furthermore, there are commands to query the target VM (e.g., to set a breakpoint at particular executable *location*) and the notification mechanism the VM informs about its current state (e.g., breakpoint hit).

The specification of the JDWP does not restrict the debuggee and the debugger to run in the same OS. Furthermore, they can run on different, hence separated

platforms. Finally, they can be implemented in different languages regardless whether they are native or non-native (e.g., Java).

Information and commands roughly correlate with what the JVM TI provides. However, JDWP is optimized for use over a network and as it targets associated problems by filtering and batching (described in section 1.1.2.1).

1.1.1.3 Java Virtual Machine Tool Interface (JVM TI)

JVM TI is a native interface provided by the VM. It defines the API the VM must provide to support a wide a range of tools that need an access to VM state, such as for profiling, debugging, monitoring, thread analysis, and coverage analysis. The provided functionality includes, but is not limited to, information retrieval (e.g., current top frame of a call stack) and modification of such information (change a value of a variable at a given frame), actions (e.g., breakpoint set) and notifications (e.g., breakpoint hit).

VM implementations, which do not implement this interface, can still provide an interface based on the JDWP[3]. There are, however, specialized Java debuggers that use JVM TI directly such as Takipi Server debugger [8] for debugging and monitoring of Java application deployed in production environment.

The specification of such VM interface allows VM implementators to create VM extensions that are logically the same regardless of the OS platform and its specifics given by the native environment. The VM extension would however require conforming to each platform it is designed for.

1.1.2 JDWP Specification

The JDWP Specification is available as web pages at Oracle website [9]. The specification is versioned the same way as the Java VM specification as it introduces new features that come only with new major releases of the Oracle JVM.

For every new JVM Specification, new version of the JDWP Specification is published. The versioning is therefore three numbers up to 1.5.0 [10] and a single number from Java SE version 6 [11]. However, the internal versioning remains as the major and minor numbers, which is the version format the debuggee advertises to a debugger.

The specification does not change it only adds new features. Version 1.5.0 added support for generics, version 6 for monitor events and version 7 does not introduce any changes.

Besides the general overview, the JDWP Specification consists of:

1. JDWP Start Up section where the handshake is described
2. JDWP Packets section with the packet structure

[3] This idea is crucial for this work. As we analyzed, this is the ultimate answer for the problem, this thesis is solving.

3. Detailed Command data types section (section 1.1.2.3)
4. and, finally, with the Protocol details within a dedicated page; it can be divided to
    a. Commands details (section 1.1.2.4)
    b. Constants (section 1.1.2.5)

1.1.2.1 Packets and packet structure

There are two defined kind of packets in the JDWP specification: the Command packet and the Reply packet.

The debugger sends command packet when executing a particular command and then receives the reply packet to get the result of the command. The JDWP agent uses command packets when an event occurs in the target VM. In this case, the debugger is notified through the command packet.

The both packet types consist of a *header* and a *data*. The shared header fields are *length*, *id*, *flags*. The command packet additionally contains *command set ID*, *command ID*, whereas the reply packet contains *error code ID*. Both, the command packet and the reply packet can include data. Depending on the command (given by a command set ID and a command ID as explained in the section 1.1.2.4), which determines the expected reply, the size and the content of the data differ.

1.1.2.2 Filtering and batching

The filtering and batching is an important aspect of the JDWP back-end as it is the way to optimize the amount of data being sent from a debuggee to a debugger.

Batching stands for an aggregation of similar events into one composite event so that only one command packet is used.

Filtering is an event post-processing when event request associated modifiers are applied in order to minimize the number of not desired events. JDWP specifies various modifiers: *Conditional* (an expression is evaluated in order to match an event), *ThreadOnly* (the event must have associated specific thread), *ClassMatch* (only events with an associated specific class are matched), *LocationOnly* (the event must occur only at specific executable location), etc.

1.1.2.3 Common Data types

The common data types table specifies the data types the specification refers to. The data types are divided into

Primitive types

The primitive types have fixed size. They are used by the protocol only to provide the information about the state of the communication. These primitive types do not correspond with primitive variables and their values in the target VM[4].

Identifiers

The semantics of identifiers correspond with the native identifiers used in JNI and so their size. The specification clearly states whether an ID can or cannot be reused; moreover, whether it must be unique or not.

Identifier identify

1. *objects* (i.e., class instances [12]). Additionally, objects with special purpose from the JDWP perspective, specifically: *threads*, *thread groups*, *strings*, *class loaders* and *class objects*, are identified using a dedicated special object ID that must not be interchanged with general object IDs;
2. *reference types*. If a reference type corresponds with a particular Java class it is referenced as *class ID*. Array and interface types are referenced respectively by *array ID* and *interface ID*. The specification does not use any dedicated types for enums, annotations nor throwables nor any other remaining Java types.
3. Other VM entities: *methods*, *fields* and *frames*

Location

The location identifies an executable location. The location is a composite of other common data types in following order: *type tag* constant (see 1.1.2.5), *class identifier*, *method identifier* and an *index* of the instruction within the method.

Values of variables

Variables are divided as:

1. *strings*; that are conveniently used to transfer the information represented by a Java String. The Java String objects are however represented as standard objects as stated above.
2. *values* with preceding type tag (see 1.1.2.5) that can be further divided into
   a. *primitive values* that are represented the same way as JNI primitive types
   b. *objects* that are represented by their object identifier (described above)
3. *untagged value*; that is, a value without the preceding type tag
4. *array region*; a convenient representation of arrays

[4] The information how to transfer primitive values is completely missing. The specification is referring to a table as it states "More details about each value type can be found in the next table" [9] but the table is does not exist.

1.1.2.4 Commands

The main part of the detailed specification consists of command definitions that are divided into 18 command sets. There are precisely 90 commands in the JDWP Specification version 6. A *command* represents a request from a debugger, except for the *Composite* command (see more below) which uses a debuggee to notify a debugger of an occurred event.

Every command definition consists of a *Description*, input contract named *Out Data*, output contract named *Reply Data* and, finally, a list of possible error values named *Error Data*.

The *Out Data* and *Reply Data* of command specification clearly references the *common data types* (section 1.1.2.3) and also *constants* (section 1.1.2.5). However, the possible error values are not always clearly listed in the *Error Data* lists. It happens, the command *description* or the data descriptions make reference to additional errors.

There are two command sets, which are crucial to understand the data flow: *EventRequest* and *Event* command sets. The intention of the remaining command sets (e.g., *VirtualMachine*, *ThreadReference*, *Method*, etc.) is straightforward and does not require detailed explanation.

Event Request Command Set

The event request command set provides a debugger with special commands to control the number of generated events. Event requests need to be managed by a *RequestManager* who decides whether an event should be sent back to the debugger.

Event Command Set

The event command set contains a single *Composite* command. This is the only command triggered by a debuggee. Its purpose is to notify a debugger of an internal state of a debuggee (e.g., breakpoint hit). An event is sent if it matches against an event request, otherwise it is discarded.

1.1.2.5 Constants

The JDWP specification defines multiple sets of constants. These constants are sent across the JDWP protocol as various JDWP primitive data types depending on the context in which they are used. A description of important constant types follows (there is exactly 11 constant types in the specification):

Error Constants

The error constants are used to notify a debugger of an error that occurred during an execution of a command.

EventKind Constants

These constants determine a kind of an event. There are different kinds such as: breakpoint (i.e., breakpoint hit), exception (i.e., an exception occurred), thread started, thread ended, etc. These constants are used to pair an event request with an event.

TypeTag Constants

The TypeTag constant determines whether a referenced type is either *class*, *interface* or *array*.

Tag Constants

The tag constants are used to identify either a specific specialization of an object or a primitive type. It is crucial to identify specialized objects correctly during the communication.

1.2 GNU Classpath JDWP

The GNU Classpath JDWP project provides an implementation of the JDWP back-end in the Java language. It is designed to be used by a running program as a runtime library instead of the underlying VM. The subset of functionality that is not available to the program (e.g., class redefinition) is required to be implemented in the underlying VM through a native interface. Additionally, it uses standard Java runtime objects from the `java.lang` package such as `Thread`, `ThreadGroup`, `Class`, `Object`, etc. to represent the VM entities.

There is no concise information about this subproject of the GNU Classpath project [13], although it is possible to get some information from different mailing lists where bugs and improvements in this project are recorded. The only complete information is in its sources [14] as *javadoc* comments.

This project is also used by a number of open-source JVM implementations such as Jikes RVM [15] [16] or SableVM [17] [18].

In this study, we considered GNU Classpath JDWP subproject as a suitable candidate for JPF JDWP back-end implementation. The analysis regarding the reuse of this project is further discussed in the section 2.3.

1.3 Java PathFinder

Java PathFinder is an explicit-state model checker [19] for software verification. It is a Java VM that runs Java bytecode and during the execution it verifies whether certain properties are true. It is itself written in Java language, which means, it requires underlying JVM to be able to run. However, JPF, as a VM, is not optimized for speed as

standard JVMs are; its primary purpose, as it was said, is to verify software and also to be configurable, extensible and observable.

In the middle of all is the JPF Core as shown by the Figure 2. JPF Core supports [20] backtracking, state matching, and non-determinism in both data and scheduling decisions. As JPF runs the program, the *state space* is being generated on-the-fly during the execution.

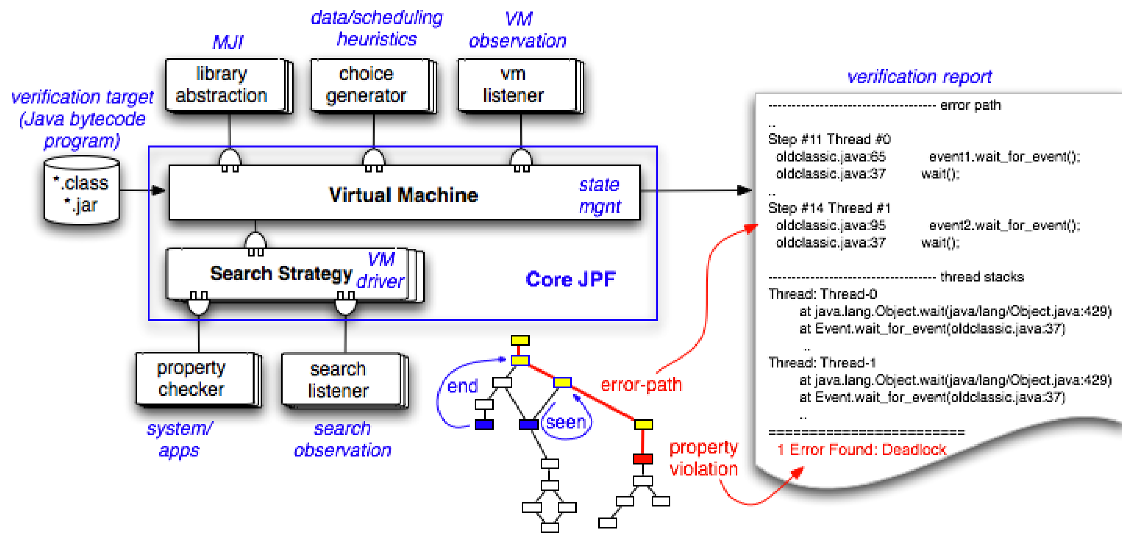


Figure 2. JPF architecture [21]

A *state* is determined by all heap objects, stacks of all threads and all static data. If a state is accessible from another one, they are connected by a *transition*, that is, a single thread executed sequence of instructions that are invisible to other threads. JPF stores every state for the purpose of backtracking and state matching. Every transition from a state to another one is determined by a *choice* that is produced by a *choice generator*. The choice is usually an action to reschedule a thread or to use a different value of a random variable. It can be however anything as it user configurable.

The *search strategy* determines the transitions produced by choice generators and the order of the states to visit.

1.3.1 State space traversal

As it was described before, JPF constructs the state space representation during the verification of the code. The act of transitioning from a state to another is referred as *State space traversal*. For this study sake, we explain more in depth information relevant to this process. After all, this is the main difference between JPF and standard JVMs that JPDA is intended for.

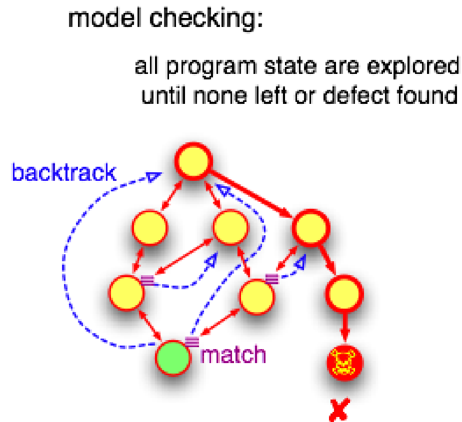


Figure 3. Traversing across the state space. [22]

To be able to traverse the state space, JPF implements an ability to store and to restore a state. To optimize the traversal, JPF also compares the state it reaches with the states it already visited. As a result, a particular path is always traversed only once.

State space explosion

The state space can easily explode to such extent; it is not possible to traverse it with existing computer resources. The computer science discipline called *model checking* pursues an investigation of methods that would shrink the state space.

With a respect to the state space JPF traverses and its JVM character, it features [19] deadlock and race-condition analysis, all runtime errors check, assertion violations, symbolic execution, different types of state search methods including heuristic ones and many more. To optimize the state space traversal, it implements dynamic partial-order reduction, heap symmetry detection and many more.

1.3.2 JPF internals

The resulting work of this thesis highly adheres with JPF internals, as its purpose is to expose them through the UI of a debugger to the user. Therefore, a brief introduction to JPF internals follows. The standard JPF setup is assumed because it is possible to reconfigure JPF to such extent that even the internal objects are represented differently.

JPF itself is represented by a `JPF` instance which has an associated a `VM` instance that incorporates virtual machine functionality. Heap is represented by a `Heap` instance.

Every object (i.e., a class instance in the program running in JPF) stored in the Heap has associated *SGOID*, an integer that is supposed to always represent the same object even along different paths. Its calculation is determined from a program context, which is represented by `AllocationContext` class. Objects themselves are represented by

`DynamicElementInfo` instances; thus, there is always 1:1 relation between `Object` subtype instances in the program and the `DynamicElementInfo` instances.

Several specialized program instances have also associated info objects where the virtual machine stores more detailed information required to work properly. The most important objects are threads, classloaders and exceptions represented by `ThreadInfo`, `ClassloaderInfo` and `ExceptionInfo`, respectively.

The classes `ClassInfo`, `FieldInfo` and `MethodInfo`, respectively represent information about the types (i.e., arrays, classes and interfaces), fields and methods. The static data relevant to the type is stored in the `StaticElementInfo`.

1.3.3 Extending JPF

The Java PathFinder is designed to be easily extensible. The most important extension points are *listeners*, *native peers* (MJI) and *bytecode factories* [23]. Every extension is supposed to meet the requirements described in the JPF project layout wiki page [24]. In this work, only listeners are used hence a brief explanation follows.

Listeners are based on an observer pattern. Each instance can register to receive various notifications about certain events of the *VM* or *search* type. These notifications begin with low-level events such as instruction executed and ends with high-level events such as state backtracked.

To use the *listeners* feature, one should implement `VMListener` or `SearchListener` interfaces. JPF can be configured to use such listener through the *listener* JPF property. It is also required to extend JPF classpath so that it can locate the implementation of the listener.

Listeners are also referenced as *Listener API* in this work.

2 High-level analysis and comparison

This chapter describes possible high-level solutions that were relevant to the JPDA implementation assignment of this thesis.

There are precisely three possibilities where to plug either the debugger or the debuggee in the JPDA concept; therefore, the same applies for JPF as a debuggee. The list is as follows:

1. Java Debug Interface (JDI) implementation
2. Java VM Tool Interface (JVM TI) implementation
3. Java Debug Wire Protocol (JDWP) agent implementation

2.1 Direct JDI to JPF solution

The idea of this solution was to take an advantage of Java based debugger and debuggee communicating over pure Java API.

The motivation of this solution was to do as simple as possible implementation where the debugger is able to work directly with the JPF internal objects that represent various JPF entities without a need for their translation or an abstraction.

2.1.1 Realization Idea

Since standard Java debuggers use the JDI API as the standard way of communication with a debuggee, it was very convenient to take an advantage of this fact. That is, to substitute the standard JDI implementation with the JPF specific one.

Debuggers use an entry point to the JDI, which is either a boostrap or a launching connector class depending on the debugger implementation. The idea is to inject the JPF JDI implementation into the debugger so that it is used instead of the original one. The injection must be implemented for each debugger it is intended for. As a result, the debugger queries JPF through JDI objects, which serve as facades[5], as they wrap the JPF internal objects.

2.1.2 Rejection and Reasoning

This idea proved to be straightforward from the implementation perspective as far as no remoting was included (i.e., support for inter process communication). We implemented a working proof of concept in Eclipse IDE that allowed a user to start a debugging, to set a breakpoint and to pause at this breakpoint. To include remoting

[5] For example StackFrame interface from the com.sun.jdi package is implemented as a facade of StackFrame class from the gov.nasa.jpf.jvm package.

capabilities into this concept, we chose Java RMI. The motivation was to allow JPF to run in a different process than a debugger. We found, such a step requires to divide logic between the debugger and a debuggee and also to define what exactly should be communicated between them.

The requirement to separate the logic into two processes led to a different solution, which was to implement the JDWP protocol in JPF (see section 2.3). JDWP protocol specifies how to separate the logic as it defines the format and semantics of data being sent between a debugger and a debuggee.

2.2 JVM TI for JPF solution

The JVM TI solution (general concept described in the section 1.1.1.3) requires JPF to implement a native interface [25] for each platform it is used on and intended for (e.g., Windows and Linux platforms at least).

The main advantage of this solution is a realization of wider interface which also provides functionality for profiling (i.e., attaching of profiling agents)[6], monitoring, thread analysis and even more. Additionally, JPF would reflect standard pattern implemented in common Java VMs.

To the contrary, common Java VMs are natively implemented and hence it makes a sense for them to expose any functionality using a native interface. This solution requires a design of a native library, which provides JVM TI on one side and queries JPF through JNI on the other side. This library would be written in a native language; hence, it would be platform specific. On the contrary, JPF is written in Java language, which is not native; therefore, it can run on any platform. Moreover, the entire JPF infrastructure and the community are Java oriented.

Considering the disadvantages of this concept as well as the advantages of the JDWP solution, which are discussed below, this solution was considered and found inappropriate.

2.3 JDWP for JPF solution

The idea of this approach is to plug JPF at the JDWP level. That is, to create a JDWP agent. The purpose of the agent is to mediate debugger requests into JPF and vice versa by implementing the JDWP protocol. Further details of this solution are described later in this thesis as this solution was chosen as the most suitable.

The motivation for this solution is that common Java debuggers, which are part of IDEs, can connect to the JPF debuggee as if they connect to common JVMs. Moreover, it requires no modifications at the debugger side. On the other hand, this

[6] This is what the JTG project (described in 0) desired to implement.

solution requires an understanding of JDWP specification. It also introduces higher abstraction of JPF internals since the data sent across the JDWP protocol are raw bytes.

2.3.1 GNU Classpath JDWP based solution

Thanks to the GNU Classpath JDWP project (see 1.2), it was possible to create a working proof of concept that even had a potential to be a base of this work. However, this project has certain flaws and we decided to remove them entirely.

Firstly, the GNU Classpath JDWP agent does not take an advantage of a polymorphism [26] and extensively uses switch statements despite the fact the class model is designed hierarchically. Secondly, it is designed to run as a part of the program that is being debugged; and therefore, the integration with JPF required a complete redesign.

Given these facts, we decided to reuse only the communication layer although with a respect to the redesign made at the other layers. The communication layer is further described in the section 3.3.1.

2.3.2 Acceptance and Reasoning

This solution has reasonable advantages such as a clean integration into JPDA that is by default supported by debuggers, optimization for network communication and many more. In an addition, this solution is precisely what the JPF community presented years ago as a nice to have feature for JPF [27]. Despite the complexity of this solution, given by the extensive JDWP Specification, we chose this option, to implement the JDWP, as the most suitable one. Another important argument was also the working proof of concept based on GNU Classpath JDWP.

3 JDWP for JPF analysis

In this chapter, we present the concept of Java Debug Wire Protocol agent for Java PathFinder and we elaborate the analysis of this solution.

It must be noted, the JDWP specification is not precise in numerous aspects; as such, the specification can be interpreted in different ways. Nevertheless, there are already multiple implementations of different parts of the JPDA working together. Therefore, the interpretation of the JDWP specification is restricted to work with existing debuggers. In an addition, several open-source implementations of the JDWP specification are freely available; thus, the interpretation may be consulted with the existing ones.

For this study sake, we chose to use OpenJDK [28] and Apache Harmony [29] projects as reference implementations of the JDWP agent. As a reference debugger, we chose Eclipse IDE [30] and its JDT debugger. For the JDI implementation, we chose Eclipse JDT JDI project [5]. The GNU Classpath JDWP implementation [13] turned to be wrong in several aspects. For that reason, we could not use this project as a reference implementation in many cases.

3.1 JDWP Specification ambiguity

In this subchapter, we elaborate major ambiguities of the specification and the way to solve them considering the reference projects we chose as mentioned above.

Values of primitive types

JDWP does not specify how to transfer values of primitive types. Nevertheless, as the specification extensively refers to JNI, the values of primitive types should be handled in accordance with the JNI Types and Data Structures Specification [31] too. That is, to expect such a format when reading an input as well as to use this format when writing them to the communication channel.

Error constants

JDWP specify a large number of error constants; however, more than a half is not referenced by the specification altogether. Therefore, the use of constants from the Eclipse JDT `JdwpReplyPacket` class implementation [32] should be used as a guide for JDWP agent commands implementation. The commands shall return the same error code set as the Eclipse JDT implementation expects.

Line Table

The specification is not clear in the terms how the line table of a method should be constructed. The OpenJDK implementation constructs the line table as follows. Given the sequence of instructions, for each of them, the line table should contain a new record whenever a line number of a code index sequence is changed. The line number, for an increasing sequence of instructions, can therefore also decrease. As a result, the line table may contain non-monotonic line number sequence.

It was also needed to fix a bug in JPF Core, as it did not expect non-monotonic line number tables in the class files.

3.1.1 Modifiers

Count modifier

The use of count modifier (modifiers are described in the section 1.1.2.1) is not clearly described in the specification. However, reference agent implementations realize its functionality as follows.

The order, in which the modifiers for an event request came, matters. All the modifiers should be evaluated in this order and the same applies for the count modifier. Whenever a modifier, which precedes the count modifier, does not match a given event, the count modifier should remain untouched. When the count modifier reaches zero value, subsequent events shall not be reported for this event; therefore, this event request should be removed.

ClassMatch modifier

Even though the JDWP specification is talking about regular expressions for a class modifier, the pattern is actually very limited. The only special character it supports is an asterisk, which stays for any character sequence. In fact, it is not a regular expression altogether. As a result, the agent shall accept any pattern, as well as the reference agent implementations do. Therefore, since no pattern is invalid, the command *Set* from the *event request* command set never returns invalid string error.

InstanceOnly modifier

The specification is not clear what exactly an *instance* this modifier should match is. The reference agent implementations however use the instance to match the object of the field that is being accessed and for the rest of allowed events, it is always the object referenced at the top of the frame of instance methods. The null instance value of this modifier matches the static fields and methods.

3.1.2 Data hierarchy

Surprisingly, JDWP specifies all the data as if they were flatly structured. However, a deep investigation reveals that *errors*, *identifiers*, *values*, *events* and *modifiers* are sophistically hierarchically structured. Moreover, the hierarchy is applied the same way throughout the specification. It is a key thing to understand, for a good design of the JDWP agent, this implied, but never mentioned, hierarchy. Again, as a good guide the JDI implementation shall be used as it partially reflects the hierarchy in its class design. The exact design of this hierarchy is shown in diagrams later in this thesis.

3.2 State space traversal implications

The spate space traversal (described in the section 1.3.1) is the main difference between JPF and standards JVMs. In this subchapter, we discuss implications of this process from the JPDA perspective. The exact implementation of the state space traversal process is unimportant. Important is the relation between a *jpf state* change and a *vm state* change. The *jpf state* refers to what JPF traverses. The *vm state* refers to a state of VM the debugger is interested about.

The debugger uses event requests to receive notifications (i.e., events) about *vm state* changes. The problem starts when JPF moves from one *jpf state* to another and the *vm state*, in which the debugger expects the VM to be, changes too. Particularly, the *vm state* changes the debugger may[7] keep track of are interesting. The Table 1 shows events that may be a source of problems.

Event	Resolution
VMStart	Sent only once
SingleStep	May cause an incomplete step
MethodEntry	Described in the section 3.2.1
MethodExit	Described in the section 3.2.1
MethodExitWithReturnValue	Described in the section 3.2.1
ThreadStart	Described in the section 3.2.2
ThreadDeath	Described in the section 3.2.2
ClassPrepare	Sent only once
ClassUnload	Never sent
VMDeath	Sent only once

Table 1. Events and resolution

The three event kinds for which the event is sent only once should work with any debugger. To pretend the class unloading is not supported in JPF should work too. The

[7] This is debugger specific. It is not our intention to explore all possible Java debugger implementations.

step action in a debugger that expects a step event may appear as an incomplete step provided JPF changes to a *jpf state* where the particular thread is not running. Method and thread events are described below. The stepping mechanism also depends on the algorithm that is used for a thread suspension, which is analyzed in the section 3.3.3. The rest of the events: *Breakpoint*, *Exception*, *FieldAccess* and *FieldModification* are considered as stateless and reentrant; therefore, they are safe.

3.2.1 Method associated vm state change

Entering and exiting a method can cause problems with debuggers provided they keep track of the entered method (e.g., for call stack purposes) and the jpf state change violates what the debugger expects. The same applies for method exits. As a possible workaround to achieve consistency with debuggers, the JDWP back-end can reflect the *jpf state* change with a set of relevant method exit events, i.e., to unwind the call stack. And also to send a set of appropriate method entry events, i.e., to fill the call stack to match the current *jpf state*.

We found, the Eclipse JDT debugger does not suffer with this problem since it uses these events statelessly. The only use of these events is for method entry and exit breakpoints. Therefore, the method entry and exit events are respectively sent before JPF executes the first instruction or it executed the last instruction of a method.

3.2.2 Thread associated vm state change

Thread starts and deaths are a problem since debuggers continuously keep the set of running threads. The reason is to show current thread list dynamically as well as to be able to request for a specific thread relevant events.

A possible workaround is to let the debugger to update the thread list it keeps through thread start and death events whenever *jpf state* is changed. This solution assumes the debugger implementation does not expect to receive the thread start and death events more than once. This expectation is however in accordance with the JVM specification [33] since a `Thread` object can be started, as well as exited, only once.

Besides a race condition that we found [34] in Eclipse IDE, it can receive multiple thread events for the same thread as long as the thread death event is processed after its matching thread start event. If not, the thread death event can be paired with incorrect thread start event since they are handled asynchronously [35]. As a result a thread may disappear from the thread list that Eclipse keeps.

In this work, we refer to this problem[8] as to a problem of *Appearing and disappearing threads*.

[8] This problem is the reason why the debugging of the JPF verification cannot be fully supported as further evaluated in the chapter 5.

3.3 JDWP Agent

The JDWP Agent [25] concept comes from the JVM TI Specification and it is defined as a client of JVM TI (described in the section 1.1.1.3). It queries the application running in VM through many function JVM TI provides, either in response to events, the agent registers for, or independent of them.

The purpose of various agents might be different; however, they still use the exact same JVM API. Common purpose of such agent is to provide a debugging functionality (usually through the JDWP protocol[9]) or a profiling functionality[10].

The concept of the JDWP agent is described by Junjie and Wise [36]. The idea decomposes the agent into modules shown in the Table 2.

Module	Purpose
TransportManager	A provider of a communication between a debugger and a debuggee
PacketDispatcher	Processes the packets the transport layer receives
CommandDispatcher	Responsible for a command lookup
CommandHandler	Command execution
RequestManager	Manages all event requests and is responsible for pairing them with a callback which results in an event creation
JVM integration (JVMTI)	The virtual machine providing callback mechanism as well as an interface to observe its state
EventComposer (EventCallback)	Creates events
ObjectManager	Manages IDs of VM entities (object) that need to be presented to the debugger
EventDispatcher	Aggregates similar events and forwards them to the transport manager
ThreadManager	Suspends or resumes threads

Table 2. JDWP Agent modules

The way the modules interact between each other is shown in the Figure 4. There are illustrated two control flows, the agent must facilitate. The flows are *commands execution* (on the right; it goes from the debugger to the JVM) and *event processing* (on the left; it goes from the JVM to the debugger).

[9] As it was mentioned in the section 1.1.1.3, there are debuggers that do not use JDWP protocol.
[10] There is no specific protocol (JDWP analogical) for profiling by Oracle. Yourkit profiler [**59**], for example, hooks to JVM TI but does not mention a use of any other standard interfaces or protocols.

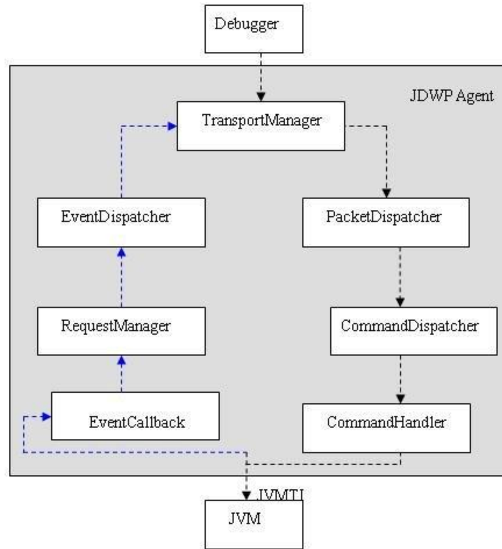


Figure 4. Standard JDWP data flow [36]

The former flow is described in detail in the Figure 5. The most important aspect is that a command handler, besides querying the VM, can also query the request manager which stores a request for an event.

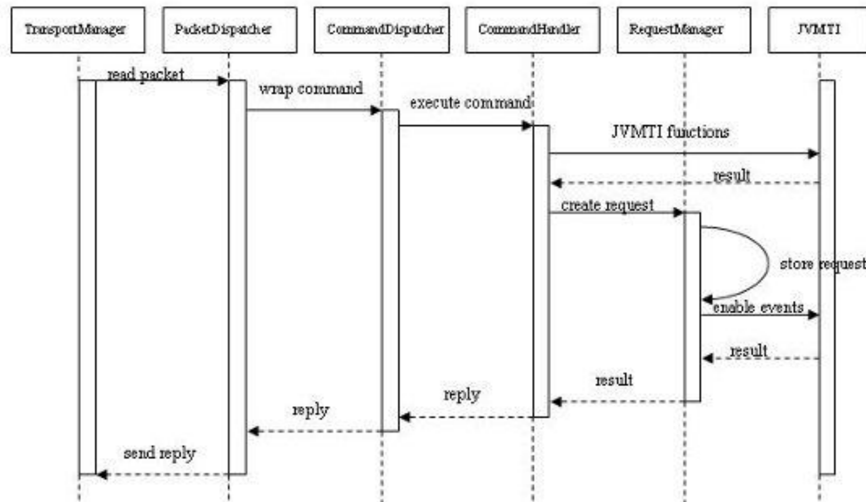


Figure 5. Command execution diagram [36]

The latter flow is described in the Figure 6. This sequence diagram describes the mechanism of event creation and its pairing with event requests. Furthermore, there is a translation of all the objects associated with the event to IDs. And finally, there is a thread suspension which any event can trigger.

A good example is a conditional breakpoint hit event. The agent must firstly evaluate the expression associated with the request for the breakpoint. Secondly, it must

translate the method and the class where the hit occurred into IDs in order to create a *location* common type. Furthermore, it sends the breakpoint hit for that location as an event to the debugger, and lastly, it suspends the thread where the hit occurred.

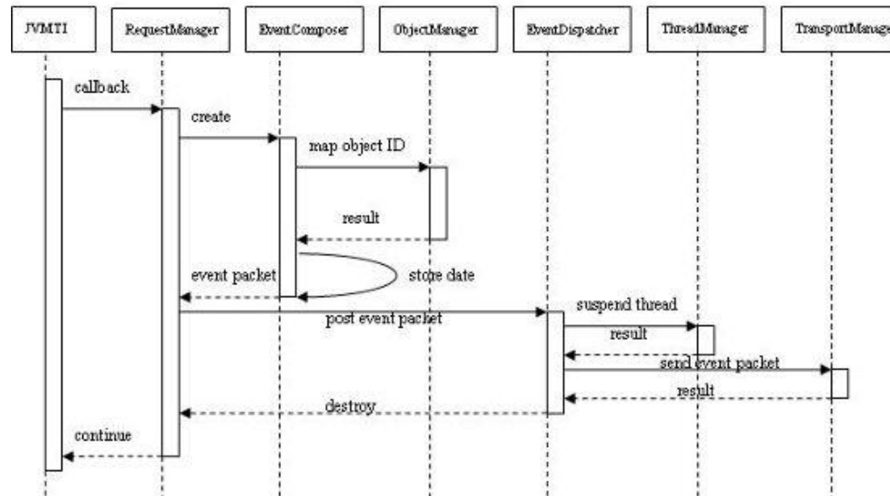


Figure 6. Event processing diagram [36]

In this work, we reuse the name JDWP Agent for an extension of Java PathFinder that communicates through its specific interface. To glue this idea together with the GNU Classpath JDWP agent and Java PathFinder, it is required to analyze, design and implement:

1. Multi-threaded communication layer for processing of JDWP packets[11]
2. Command Processor of the debugger requests
3. Event requests management for pairing with notifications from JPF
4. Event filtering facility that controls the number of generated events by the JDWP back-end
5. Suspension manager
6. Management of object instances and reference types identifiers, guaranteeing that IDs sent across the JDWP protocol meet specific JDWP requirements on the VM objects identification and on the lifetime of identifiers
7. Agent integration with JPF

[11] The communication layer is reused from the GNU Classpath JDWP implementation. The rest is however not.

3.3.1 Communication layer

In this section we introduce the details of the part we reused for JPF JDWP Agent of the GNU Classpath JDWP project (already mentioned in the section 1.2).

The communication layer includes implementation of: initialization from the *configuration property*, *transport methods*, *server* and *client mode*, *packets* and *packet processing*.

3.3.1.1 Configuration property

The initialization of the agent requires a configuration property. The format of this property is a sequence of properties with values delimited by comma. Its purpose is to specify a *transport* method. The transport method may require additional properties depending on its implementation. Furthermore, the *client* or *server* mode and *suspension* can be set.

The exact format of the configuration property and its sub-options is listed in the user manual (see Appendix II).

3.3.1.2 Transport methods

The part of the agent that provides an abstraction for a communication with the debugger is called *transport*. Any transport implementations must subclass the `Transport` interface as this is the way to dynamically initialize an arbitrary communication method. The agent comes with a socket implementation only (the `SocketTransport` class).

The socket transport requires an address parameter in a format *hostname:port* (*hostname* defaults to localhost if omitted), which together with an optional *server* parameter, determines the way the socket is created. It can either run as a TCP server and wait for a connection or a TCP client and connect to the server at the given hostname and port.

In this study, we have found the socket transport provider is sufficient and it is not necessary to implement any other providers (e.g., shared memory).

3.3.1.3 Packets and packet processing

The GNU Classpath JDWP agent implements two packet types `JdwpCommandPacket` and `JdwpReplyPacket` with a common supertype `JdwpPacket`. The former one transfers commands requested by either a debugger or a debuggee. The commands sent from a debugger are received by the `JdwpConnection` thread, which stores them in a shared queue. The `PacketProcessor` thread further processes these packets and forwards them to a command dispatcher.

The implementation of the command dispatcher is made in a scope of this work.

3.3.2 Commands, Event Requests and Events

The command flow (as it was already introduced in the beginning of this chapter) consists of five steps:

1. Command set mapping
2. Command mapping
3. Command execution
4. Error processing
5. Reply

The purpose of command set and command mappings, is to translate two IDs into an actual command implementation. It can be realized using switch statements, hash maps or arrays. Despite the poorest performance of the hash maps (when comparing to branch instruction or a single access to an array), we chose this option as the most elegant in a respect to the extensive use of polymorphic nature of the whole agent design. The command performance is not critical (in a contrary to event processing) as it is always what a user triggers.

The event request manager provides a simple functionality to store requests for each event kind and then to trigger matching of an event with all the requests. Each request implements the decision, whether a request matches an event. The mechanism of event processing is precisely as follows:

1. Event request lookup
2. For each found request, filtering needs to be applied. As a result, some requests may be discarded.
3. For each remaining request, all the events paired with the request ID are put together into one composite event.
4. The suspension policy for this composite event is calculated according to the higher status wins rule[12].
5. The composite event is sent to the debugger.
6. Finally, the JPF execution is suspended in accordance with the calculated suspension status.

The design of events, event requests; their hierarchy; and the way to allow an association of a request with an event according to the specification is further described in the design chapter 4.

[12] The specification is not clear on this matter; however, the GNU Classpath project implements the functionality the same way.

3.3.3 Execution and Suspension

While the program being verified is typically multi-threaded, JPF executes only one thread at a time since it runs single-threaded. The parallel thread execution is entirely simulated and the reason is that JPF needs complete control over the thread scheduling.

The agent must provide a functionality to suspend a thread and the whole VM. We analyzed, JPF thread suspension can be designed as follows:

1. To allow a debugger to explicitly suspend a thread while some other threads are still running in JPF
2. To always suspend the whole VM with the *SuspendPolicy* constant *ALL*
3. To pause the whole JPF verification even if only one thread is explicitly suspended by the debugger

3.3.3.1 Explicit suspension of a thread

The first option, to allow explicit suspension of a thread, is closer to the standard JDWP agent concept (i.e., conforming to the JVM TI specification). Apparently, this is what a user expects from the debugger while debugging a Java application. Ultimately, this is what a user will intuitively expect while debugging JPF verification. However, as it was already elaborated, JPF only simulates parallel thread execution. Following observation explains consequences of explicit thread suspension.

When JPF traverses along a particular path in a state space, the request from the debugger to suspend a thread may cause JPF to follow a different path. In other words, the original path is not debugged anymore and moreover, the verification itself can be violated because of the new path the suspension causes. This statement applies for both a single-path JPF run as well as a general JPF verification.

As a conclusion, this alternative, due to its complexity and proneness to violation of the verification itself, was found to be only sub-optimal.

3.3.3.2 Suspend always the whole VM

This option has two sub-options. The former one is to always send events with the *ALL SuspendPolicy* constant and expect a debugger to reflect this information even if the request was sent with *EVENT_THREAD* constant. Even though the specification theoretically allows such a behavior, we found it does not work with Eclipse IDE JDT debugger. Therefore, this option is inacceptable[13].

The latter sub-option is to configure the debugger to always request a suspension of the whole VM which is possible to a certain extent (for instance, Eclipse IDE has

[13] There might be however a chance to convince people in Eclipse community that the JDT debugger implementation needs a modification. As it is shown later, this option turned to be unnecessary.

such an option in its preferences [37]) but still, this option creates unnecessary constraints on the debugger.

3.3.3.3 Pause JPF verification

The idea is as follows. Whenever at least one thread is suspended, JPF verification is paused. If the debugger suspends another thread, it will appear as suspended in the debugger. JPF verification will continue only and only if all suspended threads are resumed. As a result, if JPF verification is paused because of a thread suspension any repetitive resumption and suspension of another thread will not affect it from being stopped at the same instruction for the whole time JPF is paused.

We chose this option as the most convenient for the JPF JDWP Agent because of its relative simplicity and a guarantee of no interference with the JPF verification. Finally, it is easy to understand for a user and additionally, similar approach is used in JPF Inspector [38].

This decision also provides an option to use a carefree stepping algorithm. That is, to let JPF schedule the threads any way it needs while the thread step is executed.

3.3.4 ID management

The state space traversal problem (introduced in the section 1.3.1) puts specific requirements on ID management. ID management provides methods to map a JPF entity to a number, i.e., ID, and vice versa. It also must reflect specification requirements on the ID lifetime regardless of the JPF state.

The most complicated is a mapping of objects (i.e., class instances of the program being verified) as they can disappear and then appear again during the state space traversal. The same object can be instantiated as well as collected multiple times; however, the ID must remain the same.

The static elements are easier to map since JPF guarantees no class redefinition and, moreover, multiple class loads of the same class results always in the same `ClassInfo` instance.

The following Table 3 shows mapping of the JPF entities to identifiers.

Identifier	JPF mapping	Note
objectID	SGOID from the `ElementInfo`	Should not be reused at all unless explicitly disposed
referenceTypeID	`ClassInfo` hash	No class redefinition is guaranteed [39]
frameID	The position from the top together with the *threadID*	Must uniquely identify a frame within the whole VM although only for a time the thread is suspended.
methodID	global ID	No class morphing is guaranteed [39]

fieldID	`FieldInfo` hash	No class morphing is guaranteed [39]

Table 3. JPF entities to JDWP IDs mapping

JDWP identifiers of the JPF entities can be managed as either short-live or long-live objects. The advantage of the former option is that it cannot cause memory leaks and it does not significantly increase the minimal amount of memory required by the agent. On the other hand, a frequent creation of the identifiers requires more processor time. The latter option has opposite advantages; however, if the implementation uses weak references[14] and it is well tested, memory leaks should not be a threat.

The JPF JDWP Agent uses both approaches of identifier management (as it is further described in the design chapter 4).

3.3.5 Integration of JDWP Agent into JPF

The integration of the JDWP agent into JPF has to cover multiple aspects:

1. a mechanism for event generation
2. parallel processing of requests from the transport layer
3. an ability to perform operations with VM entities
4. and other aspects: on demand code execution, object garbage collection enabling and disabling, and threads and VM execution control

The event generation mechanism is the first criteria we wanted to look at since it influences the architecture of the agent. We analyzed the rest of the criteria only for the solution that we found to be optimal.

As we listed in the background of this thesis, JPF provides three extension points (see 1.3.3). The two of these options *native peers* and *bytecode factories* we found to be only sub-optimal. The former option does not bring any advantages, as its purpose is to provide native implementation for the program being verified in JPF.

The *bytecode factories* mechanism may be theoretically used the same way as some debuggers and standard profiling tools [40] do, to perform bytecode instrumentation [41]. This option provides great debugging possibilities and many more. JPF does not provide nonetheless dynamic instruction substitution during runtime. Additionally, this solution brings complexity and unnecessary requirements for design and development. Finally, the JPDA requirements can be fully satisfied with the listener extension point as it is further discussed below.

[14] That is functionality provided by `java.lang.ref`.

3.3.5.1 Listener based

We found the *listeners* extension point, comparing to other possibilities, as the most suitable integration of the JDWP Agent into JPF. It can fully substitute the standard JVM TI callback mechanism that is used by standard agent implementations [25].

JDWP Event	JPF listener API notification
VMStart	VMListener.vmInitialized
SingleStep	VMListener.executeInstruction
Breakpoint	VMListener.executeInstruction
MethodEntry	VMListener.methodEntered
MethodExit	VMListener.methodExited
MethodExitWithReturnValue	VMListener.methodExited
Exception	VMListener.exceptionThrown
ThreadStart	VMListener.threadStarted
ThreadDeath	VMListener.threadTerminated
ClassPrepare	VMListener.classLoaded
FieldAccess	VMListener.executeInstruction
FieldModification	VMListener.executeInstruction
VMDeath	SearchListener.searchFinished

Table 4. JDWP Events to JPF listeners mapping

The mapping of the JPF listener notifications to the JDWP events describes the Table 4. The majority of events have matching notification methods and therefore these events do not need a further discussion.

Breakpoints and stepping events need to be hooked to every instruction and therefore it is required to implement this functionality for every possible instruction.

For the field events (e.g., *FieldAccess*), which are hooked to every instructions in the table, new listener methods in the `VMListener` interface could be proposed. However, the field instructions provide visitor pattern methods, which greatly facilitate the needs of this work. After all, this approach is also used by JPF Inspector.

The second aspect, to process debugger requests in parallel can be implemented in separated threads that run in parallel with the JPF main thread. The rest of the criteria are easily met since the purpose of a JPF extension is to be able to observe everything inside of JPF.

3.3.5.2 Non-standard options discussion

There is an option to implement the JDWP agent by extending the code of JPF core. That is, to hardwire the event triggers directly to JPF execution mechanism. This option has a chance to perform slightly better than when triggering the events from listeners. However, it is entirely in a contradiction with the whole JPF extension design.

Finally, the purpose of JPF Core is to do the core of the verification, not to provide well performing debugging capabilities at the expense of the verification itself. Therefore, it is meaningless to analyze this option any further.

Another possible solution is to implement the agent the same way as what GNU Classpath JDWP offers (explained in the section 1.2). This idea is obviously inacceptable, as the agent would require running in parallel to the rest of the application (i.e., the main thread and all other threads it spawns) but JPF only emulates the parallel execution. As a result, it will not be able to process requests from the debugger in parallel.

The implementation of JVM TI interface natively was discussed and denied in the section 2.2. However, the JDWP agent could use an implementation of JVM TI like interface in Java. Additionally, it would provide an interface for other extensions (e.g., for profiling, etc.). Nevertheless, there is no benefit of such solution, as standard agents that are implemented natively would not be able use this interface. Finally, implementation of such a complex interface would exceed a scope of this work.

4 JPF JDWP Agent architecture and design

In this chapter, we introduce the agent architecture, JDWP entities design and we elaborate several implementation details.

The architecture of the JPF JDWP Agent is shown in the Figure 7.

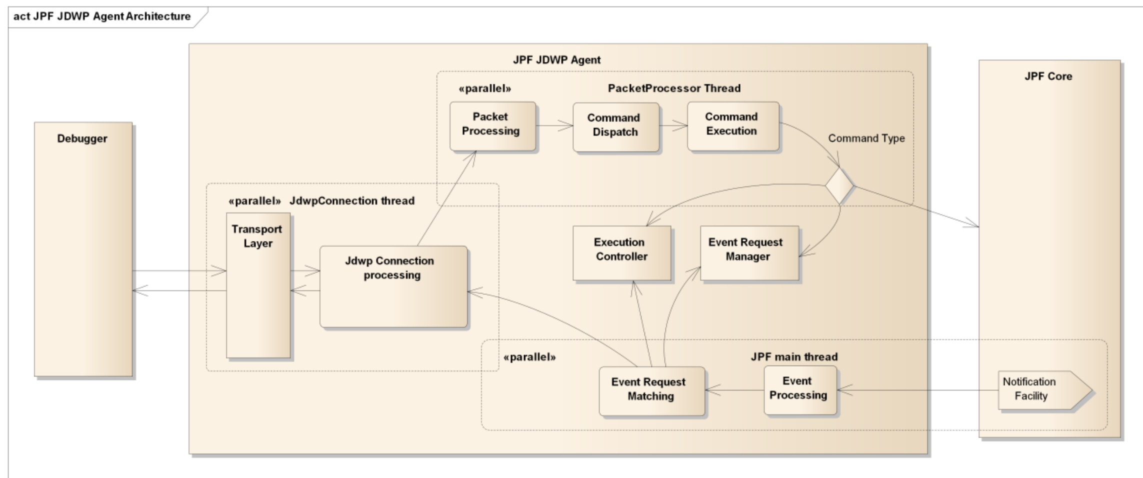


Figure 7. JPF JDWP Agent Architecture

The agent consists of three[15] threads. The way they interact is described in the first three subchapters. The communication (i.e., the integration) between the agent and the JPF Core reveals the subchapter 4.4. The design and the hierarchy of the JDWP entities are elaborated in the section 4.5. At the end of this chapter, the agent configuration is described.

4.1 Processes interaction

JDWP adds three threads on top of the main JPF thread, namely `Jdwp`, `JdwpConnection` and `PacketProcessor`. The `Jdwp` thread from a package `gnu.classpath.jdwp` is a short-live thread as it is for initialization purposes only (for initialization details refer to the section 4.3). The interaction of the other two threads with JPF shows Figure 8. All the JDWP threads are created as *daemon* threads to let JPF terminate regardless of whether the other threads are blocked by any type of operation (e.g., an I/O operation).

[15] There are precisely four threads involved as it is revealed later. The three threads shown in the Figure 7 are nonetheless crucial for an understanding of the agent architecture.

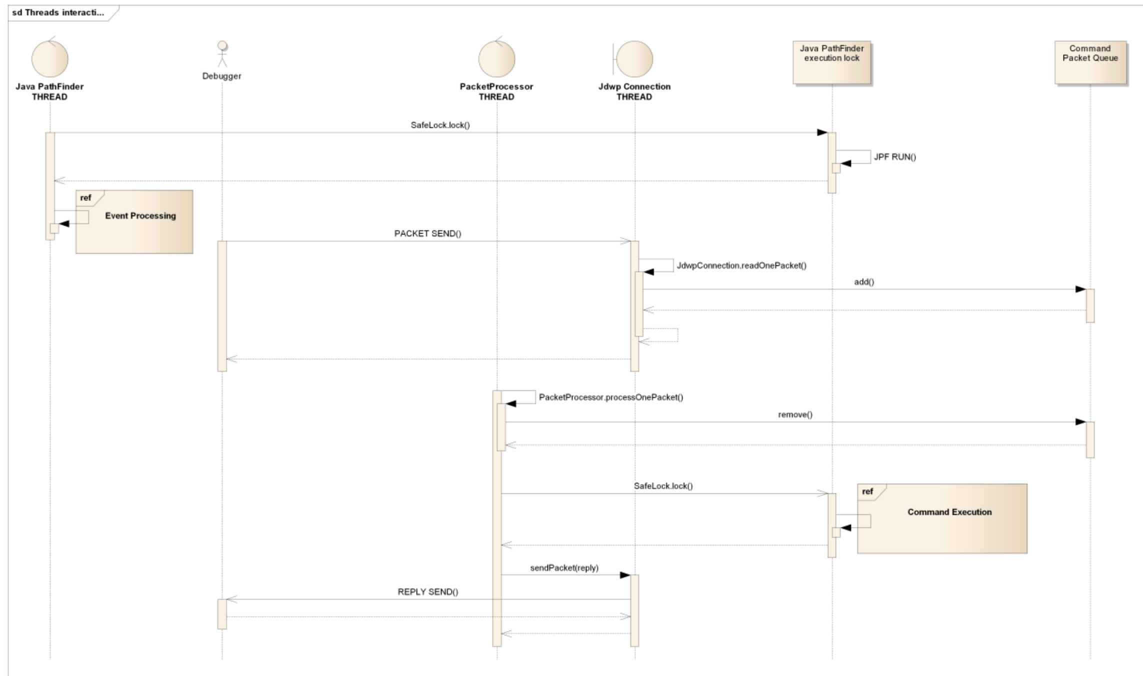


Figure 8. Threads interaction diagram

The diagram shows three entities representing Java threads while a debugger sends a command packet to the `JdwpConnection` thread and two shared objects as lifelines the threads synchronize on.

The first lifeline represents a global lock that limits JPF execution to one thread at time. It is the JPF `main` thread and the `PacketProcessor` thread that are synchronizing on this global lock. The JPF main thread owns the lock whenever it performs the verification (i.e., runs the program). It yields the lock in notify methods of the `JDWPListener` class so that the `PacketProcessor` may execute commands it received from the debugger. The reason to disallow parallel execution of JPF entirely is that JPF is by design single threaded and therefore there is no guarantee that JPF entities are designed for a parallel access.

The second lifeline represents a shared queue of command packets. The `JdwpConnection` thread receives the command packets through the transport layer (analyzed in detail in the section 3.3.1) and stores them in the shared queue and then, it processes a next incoming packet. The `PacketProcessor` thread consumes the packets and executes the commands as explained in the paragraph above. No other type of packets JDWP Agent receives.

4.2 Control flows

JDWP agent implements two crucial control flows called *Command execution* and *Event processing* flows.

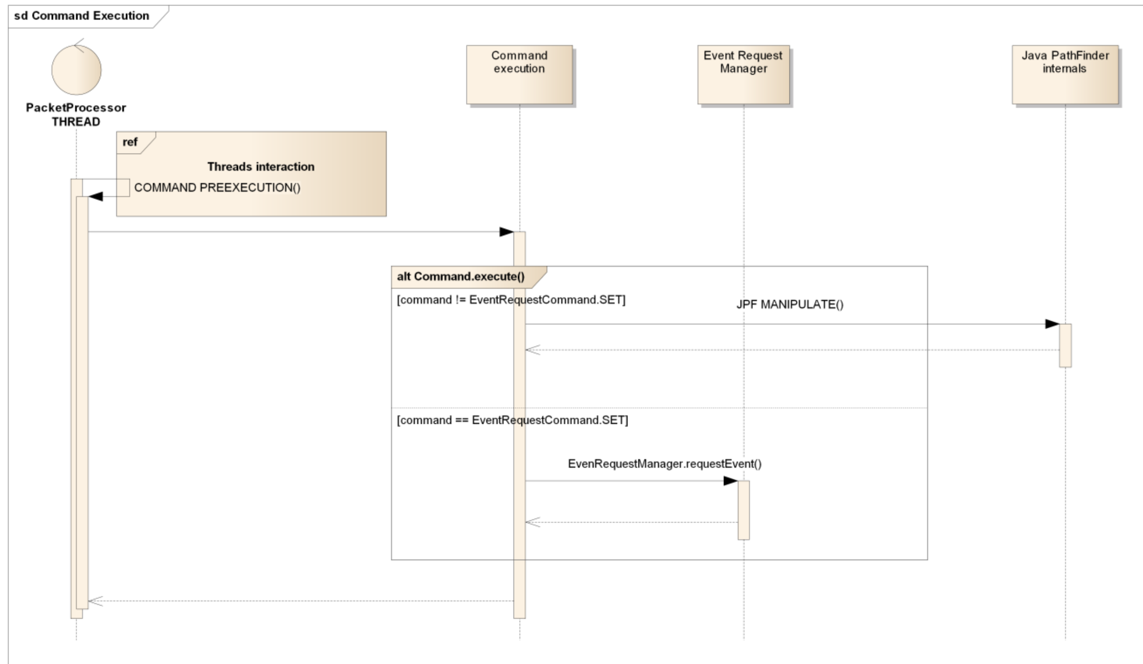


Figure 9. Command execution diagram

The *command execution* flow (Figure 9) describes the only way a debugger controls JPF, which is done through *commands*. This diagram omits communication performed by `PacketProcessor`, which is already described in the threads interaction diagram Figure 8. Commands are described more in detail in a dedicated section 4.5.1.

The diagram emphasizes one alternative command *Set* from the *EventRequest CommandSet*. The purpose of this command is to register an event request; hence, the way a debugger controls a number and kind of events generated by the JPF. It is the `EventRequestManager` class who registers these event requests and who is queried by the event processing facility as shown in the Figure 10.

The *event processing* flow, as shown in Figure 10, describes the way events are processed before they are sent under several constraints to the debugger. Events are created when JPF notifies JDWP Agent through the `JDWPListener`. It is not known; however, whether the debugger requested them; hence a filtering must be performed. The `EventRequestManager` is queried with each event and it executes all filters through `EventRequest.matches` method to detect whether a given request matches the event. Events have implemented associated list of matching requests and for each of those, a pair, the event itself with a request event ID, is put into a *Command packet* (method `EventBase.toPacket`) which is sent to the debugger. Every event obtains a suspension policy integer from the request it is paired with. The suspension (i.e., suspension of a thread or a whole JPF, or no action) is implemented in the `SuspensionStatus` class. JPF JDWP Agent implements suspension in a unique way, which was analyzed in the 3.3.3 section.

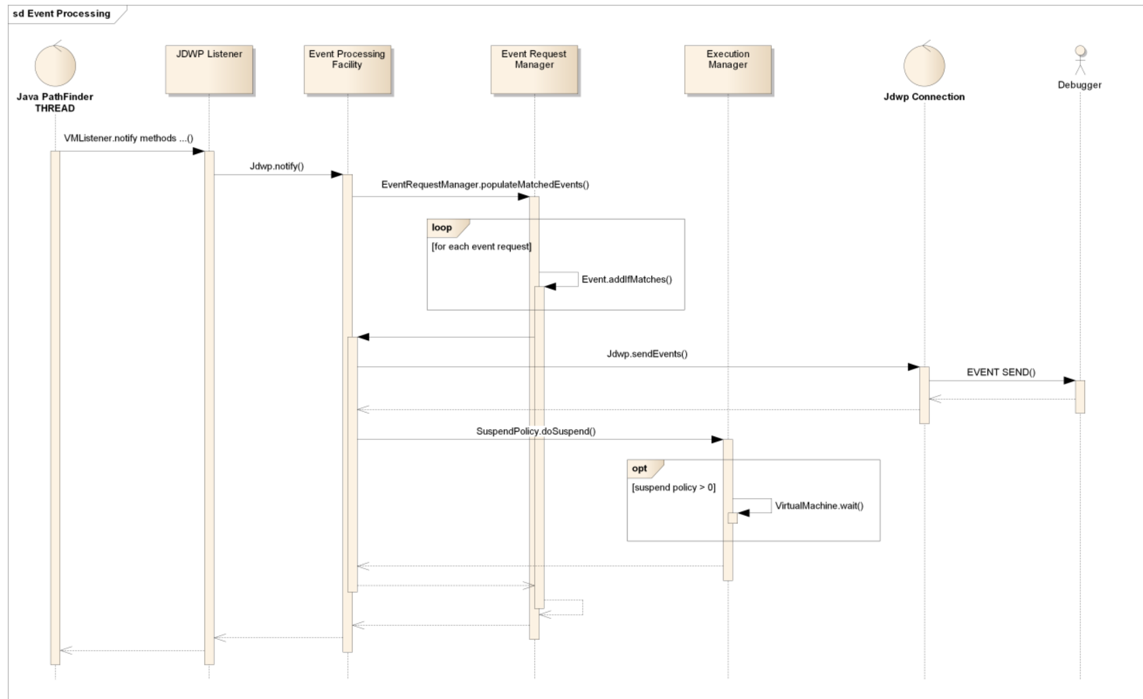


Figure 10. Event processing diagram

4.3 Execution and initialization

JPF JDWP agent supports two modes of execution. The most preferred, as well as the most convenient, way to enable the agent is called the *Listener* mode. There is also a possibility to run the agent in a *Standalone* mode. The initialization of the JDWP back-end is logically the same regardless of the execution mode.

The *Listener* mode requires user only to enable the `JDWPListener` for the verification. The initialization is performed in a default constructor of this listener. The initial suspension of the JPF verification is implemented there as well. This setup requires *jpf-jdwp* extension to be available through the JPF site properties file.

The *Standalone* mode is less preferred as it adds additional requirements on the configuration of the JPF verification. The verification starts from the `JDWPRunner` class, which implements the main method, from where the JDWP back-end is initialized and JPF is programmatically started. The minimal requirements are to correctly specify *classpaths* for both the JPF running in a standard JVM and the program running in JPF. Finally, it is required to specify the target class of the verification, which is consumed by JPF.

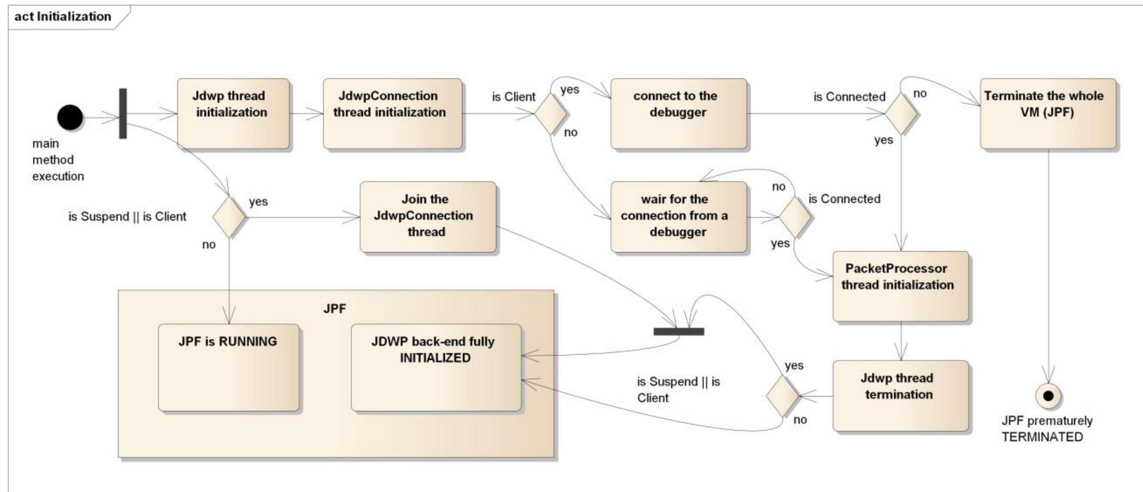


Figure 11. Jdwp back-end initialization

Before the agent is ready to process the first request (i.e., the first command packet), it needs to run an initialization sequence (Figure 11) which is executed by the `Jdwp` thread. The purpose of this thread is to prepare the JDWP back-end asynchronously without blocking the JPF itself from running. To the contrary, the value *y* (i.e., *yes*) of the *suspend* parameter of the configuration *jdwp* property makes the JPF to wait (i.e., to join the `Jdwp` thread) until the initialization sequence is completed in order to prevent the JPF from running before the debugger is attached. The initialization of the `JdwpConnection` thread includes also establishment of the connection with the debugger as the figure shows.

The configuration property that controls the JDWP agent initialization was described in the section 3.3.1.1.

4.4 JPF integration

JDWP Agent integrates with JPF Core in two aspects as implied by the overall design diagram in Figure 7. It is through the commands, which is how a debugger accesses JPF entities, and through the listener API, which is how the events are triggered.

The JDWP integration with JPF through commands is not separated by any kind of abstraction. JDWP commands directly manipulate with JPF entities and therefore commands implementation and subsequently the whole agent is tightly bounded with the JPF class model. Since the commands are executed by a different thread than by which the JPF verification is executed, a locking mechanism, which guarantees mutual exclusion (described in section 4.1) of these threads, is implemented in the agent.

The integration at the events level is implemented in the `JDWPListener` class. This class implements `VMListener` and `SearchListener` interfaces that are part of the

Listener API. As we analyzed in the section 3.3.5.1, this API provides sufficient notification methods to generate all the events.

4.5 Class model

In this subchapter, we show the design of all JDWP entities that are important: *commands*, *event requests*, *events*, *values*, *identifiers*, *ID managers* and *exceptions*. The relation between these entities was in detail analyzed in the chapter 4. All of these entities are involved in the command execution and event processing flows (section 4.2).

4.5.1 Commands

The *command sets* and the *commands* from the package `gov.nasa.jpf.jdwp.command` are designed as enums. The motivation to use enums is that Java enumerations are well suited for singleton pattern use [42] and due to their polymorphic nature; they are eligible for strategy pattern. Both properties are optimal for command dispatch and execution design.

When a command packet is received, the packet processor looks up the requested command from the given command set in two steps. At first, the command set ID is mapped to a specific `CommandSet` enum instance and then the command ID is mapped to a specific `Command` enum instance. Both of the enumerations follow a strategy pattern and the mapping of the IDs to the enum instances is enforced by a `ConvertibleEnum` interface. The command execution is realized by `Command.execute` method. For convenience reasons, `Command` enums overload this method by adding command set specific default parameters.

It is not possible; however, to execute multiple commands at once. Additionally, JPF program verification stops during the command execution (as discussed in the section 4.1). Finally, error handling is explained in a detail in the section 4.5.7.

4.5.2 Event Requests

All event requests are designed to be instances of `EventRequest` class from the `gov.nasa.jpf.jdwp.event` package. The fact that only specific modifiers can be bound to certain event requests makes the request model complex. These constraints are reflected through the `Filterable` subtypes of the `Event` interface. The check whether a modifier can be applied for certain event request is determined by

1. an interface parameter in the declaration of subtypes of the `Filter` generic class
2. an interface a particular event implements

As a result, if an event of a given type implements a particular `Event` interface, which is declared as a parameter of a certain `Filter` subclass; this filter can be used to

modify the request for this event type. This check is implemented in the `EventKind.isFilterableBy` method.

Even though a performance of event request processing is not crucial for JPF verification, the data that came with event request modifiers are uniformly stored as JDWP IDs and they are not translated to JPF entities. This is exactly opposite to the behavior of events that is described later.

4.5.3 Events

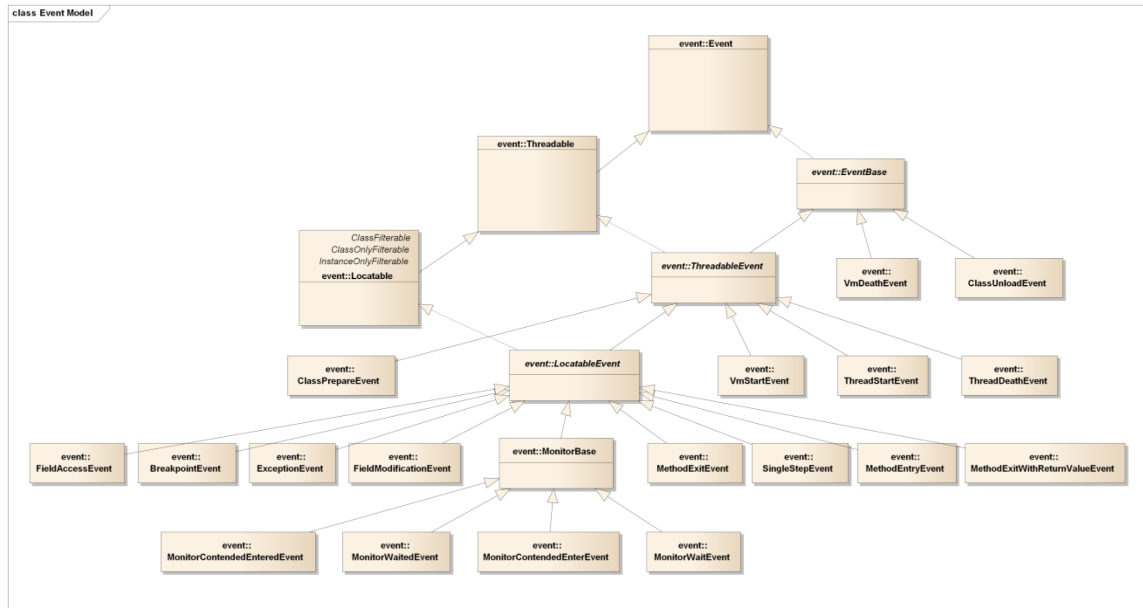


Figure 12. The Event class model

The events are designed hierarchically as subtypes of the `Event` interface. As it is shown in the Figure 12, events are organized hierarchically with three[16] base interfaces (`Event`, `Threadable` and `Locatable`). The events are subclasses of the `EventBase` class. They located in the `gov.nasa.jpf.jdwp.event` package. This inheritance reflects the main aspects that are common for certain event types; precisely, whether an event is associated with a thread, a location or, finally, with monitor related entities.

The event class model with all relevant element relationships is substantially more complex as it reflects also relationship with event request modifiers.

In order to speed up event filtering, the whole event processing is optimized to not execute unnecessary code until a matching event request is found. For this reason, all event subtypes are referencing JPF entities directly and the translation to the JDWP identifiers is performed right before it is sent to the debugger.

[16] There are precisely 11 interfaces the events implement. The diagram would be too complex if all of them were included, and therefore, only the three most important interfaces are shown.

4.5.4 Values

Values are located in the `gov.nasa.jpf.jdwp.value` package. The root interface `Value` declares functionality for an alternation of associated JPF entities regardless of their internal representation (i.e., a stack, a heap or an array). The purpose of this design is to handle specific requirements on entity manipulation through a polymorphism. Also the fact that a *value* is either a *primitive value* or an *object ID* is reflected in the hierarchy as both `PrimitiveValue` and `ObjectId` implement the `Value` interface. This hierarchy without object IDs is shown in Figure 13 (the *object ID* hierarchy is shown in detail in the section 4.5.5.1).

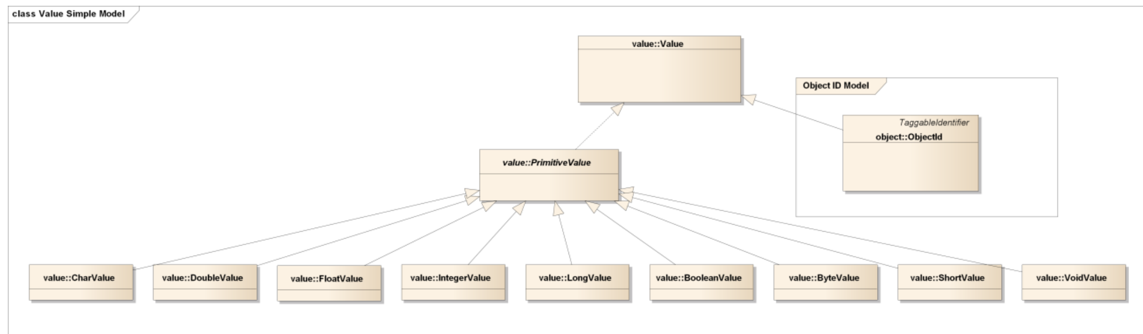


Figure 13. The simplified Value class model

4.5.5 Identifiers

Identifiers represent entities that are sent across the JDWP. They mediate a translation of the VM entities to the JDI mirrors. JDI mirror is a proxy used by a debugger to work with entities in the debugged VM as stated in JDI Specification.

The main purpose of the Identifiers model is to provide well structured and easy to understand association of object IDs to JPF entities through inheritance. The identifiers are referenced throughout the whole JDWP agent implementation. After all, they are propagated to the user through the JDI. The root package for all identifiers is `gov.nasa.jpf.jdwp.id`. As the shows, only three JPF entity types, namely *fields*, *frames* and *methods* are at the top of the hierarchy. The rest is further specialized as taggable identifiers which are even further divided to *object IDs* and *reference type IDs*.

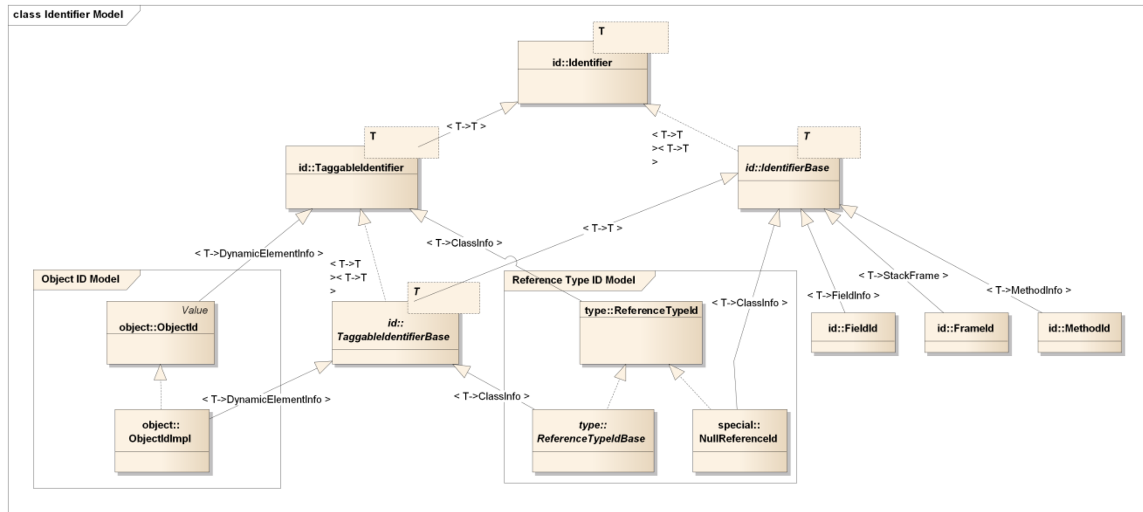


Figure 14. The simplified Identifier class model

4.5.5.1 Object ID

The interface `ObjectId` represents the corresponding *object ID* common data type from the JDWP Specification. It stands for all objects that are allocated by the program. Therefore, they are always associated with a `DynamicElementInfo` JPF entity through the `TaggableIdentifier` generic parameter as the Figure 14 shows.

The subtypes of this interface (as shown in the Figure 15) represent specialized objects in SuT (i.e., the System under Test), namely arrays, class loaders, strings, threads, class objects and thread groups. The polymorphism provides a convenient way to manipulate with all the specialized objects as if they were normal objects. The main motivation for such a design is that the natural relation of all the objects is preserved as they all implement `ObjectId` interface. Finally, this structure is also required by JPDA as the specialized objects may be passed in some cases to commands with declared *object ID* as a parameter.

In this work, we had to consider the fact that `ElementInfo` implementation overrides `hashCode` and `equals` methods of the `Object` class and moreover, multiple `ElementInfo` objects can represent the same object in the program running in JPF which is supposed to be translated to the same mirror using the same object ID throughout its life. Therefore, indexes of `ElementInfo` called *SGOIDs* (i.e., the heap identifier) are used to identify objects in the JDWP protocol. *SGOIDs* are supposed to be search global, more precisely, they are guaranteed to not change even when JPF traverses different paths in the state space. In older versions of JPF, `ElementInfo` instances were reused to represent different objects, but since JPF version 7, this is not an issue anymore. However, this approach cannot prevent a use of a different Heap implementation where this can be implemented differently.

The translation of `ElementInfo` instances to object IDs is provided by `ObjectIdManager` which is instantiated as `JdwpIdManager` singleton (shown later in the Figure 17).

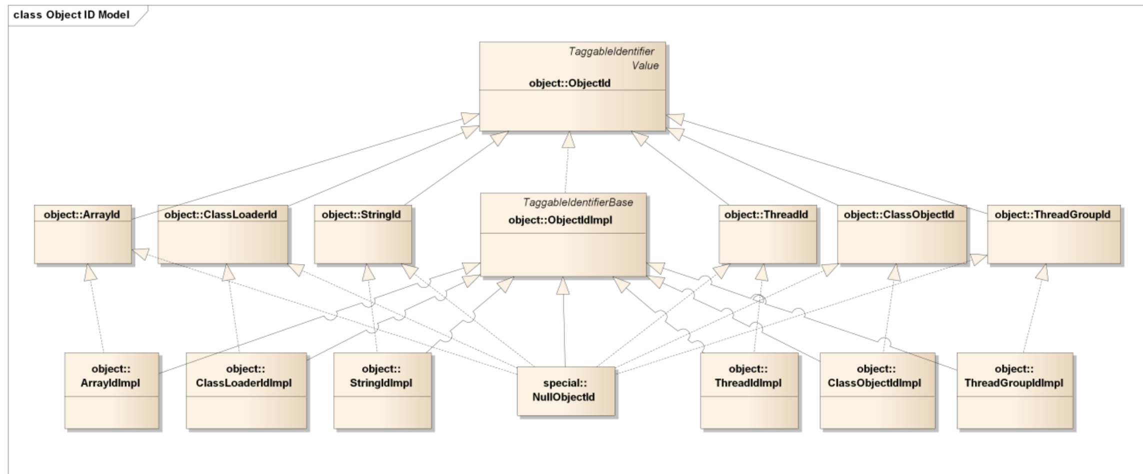


Figure 15. Object ID class model

4.5.5.2 Reference type ID

Reference types IDs are represented by the interface `RererenceTypeId` as shown in Figure 16. The type of the reference is denoted by an enum `TypeTag`. These classes and remaining implementation of reference type subtypes are located in the package `gov.nasa.jpf.jdwp.id.type`. The main purpose of these classes is to associate a particular instance of `ClassInfo` with a `long` number. This association is managed by dedicated ID managers that are subtypes of `IdManager` class (see Figure 17).

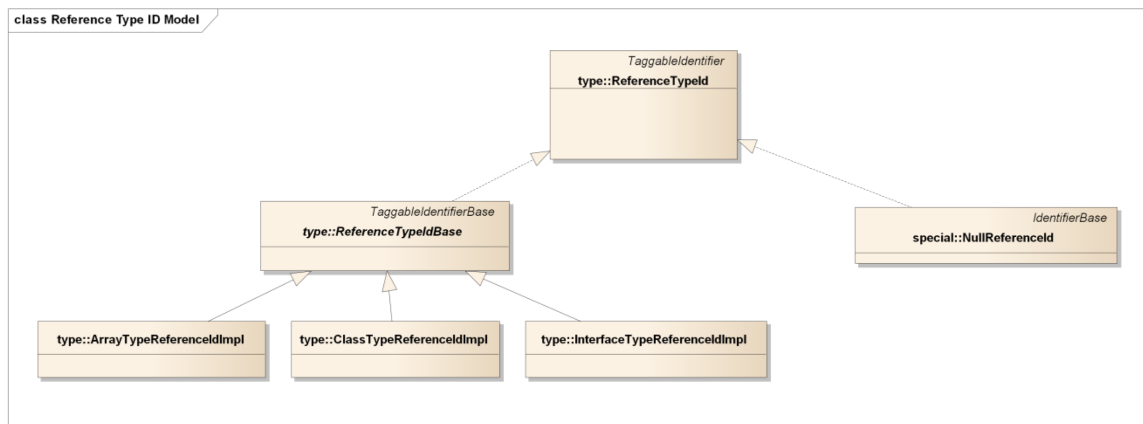


Figure 16. Reference Type ID class model

The JDWP Specification has varying requirements on the lifetime of the IDs represented as numbers of sizes from 1 byte to 8 bytes. Since JPF provides neither homogeneous identification of its internal entities nor there is a guarantee of backwards

compatibility, dedicated ID managers (Figure 17) had to be implemented in the JDWP agent.

4.5.5.3 ID managers

The ID managers do a management of all identifiers. They are hierarchically structured, as shown in the Figure 17, so that specific requirements on different entities from the JDWP model can be effectively satisfied through a polymorphism. The `JdwpIdManager` singleton is the only point of access to all functions of all the ID managers. As the figure shows, this manager have associated four specialized managers and one manager for a translation of object IDs. The `ObjectIdManager`, as an exception, does not inherit the `IdManager` abstract class because of a different approach that was used for a management of `ElementInfo` objects.

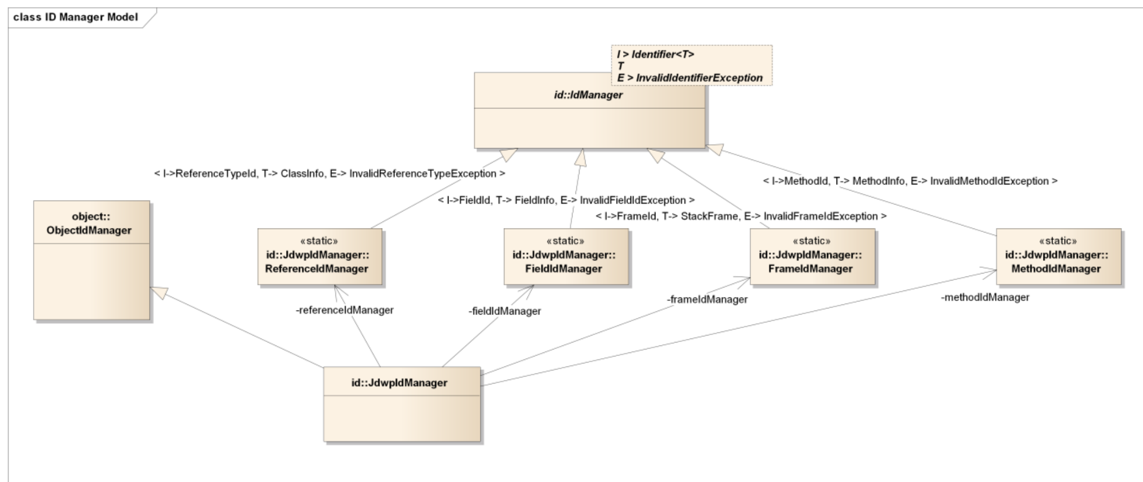


Figure 17. Identifier Manager class model

4.5.6 Locations

Location data types from the JDWP specification are implemented in the `Location` class, package `gov.nasa.jpf.jdwp.type`. Their purpose is to wrap an instruction (i.e., subcluss of the JPF `Instruction` class), which already contains all relevant information the debugger requires.

4.5.7 Exceptions

JDWP agent implements a hierarchical set of checked exceptions that conforms to the flat JDWP error constants table [9]. Due to a lack of information regarding many of the error constants (as discussed in the section 3.1), the set of exceptions is limited to the error codes that are either explicitly mentioned by the specification or their purpose is obvious from the description.

The motivation for the hierarchical structure, as shown in the Figure 18, is to take an advantage of polymorphism where associated model is hierarchical as well (i.e., object and type identifiers, values, ID managers and events).

The purpose of these exceptions is to specifically identify the type of the problem at the moment of the exceptional state. Additionally, the hierarchical model allows compile time check of the possible exceptions in the command declaration with the exceptions that are thrown anywhere in the command implementation.

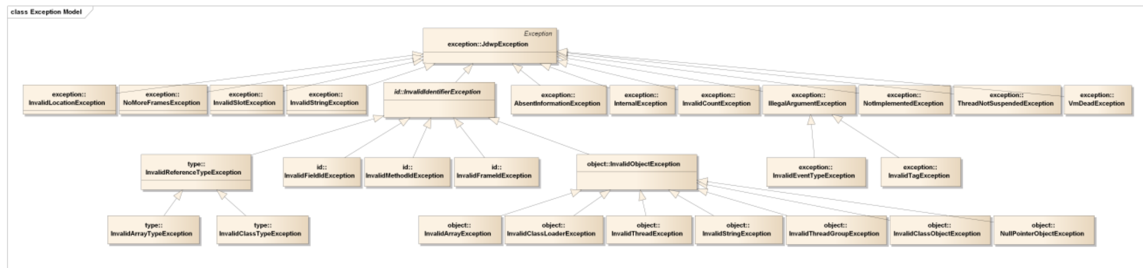


Figure 18. Exception class model

4.6 Configuration

The JPF JDWP Agent project follows standard guidelines for JPF extension [24]; and therefore, it features:

- ant-based build system
- three source directories for *main*, *test* and *example* classes
- Mercurial version control system
- configuration files for a project import into Eclipse and Netbeans IDEs
- properties file *jpf.properties* with JPF specific configuration (classpaths and default values of possible properties)

4.6.1 Logging

Even though JPF provides its standard logging facility, JDWP uses Simple Logging Facade for Java (i.e., slf4j) in a favor of better performance [43] and an ability to use any underlying logging framework. By default, JDWP agent is configured for use of log4j logging framework for which the configuration file *log4j.xml* is located in a root of main sources directory. The log4j framework is configured to use asynchronous logger in order to prevent premature truncation of log files in case of sudden agent death. If logging level higher than info is enabled, it can significantly affect JDWP agent ability to process all requests and it may impact a performance of agent.

4.6.2 Testing

JPF JDWP agent comes with a test framework for unit testing so that various parts of the JDWP back-end can be continuously tested. The JDWP test framework is built on top of standard JPF unit framework that gives a control over the code that is executed in the underlying VM as well as what is executed directly in JPF. This idea is even further extended in the JDWP test framework so that it is possible to conveniently execute arbitrary code in the underlying VM while the program is verified in JPF. The motivation is to enter a program state in JPF programmatically, to switch to the underlying VM and to execute JDWP commands there, as the debugger would do. Finally, the expected results are verified in both, the underlying VM as well as JPF.

To use this JDWP test framework, a junit class must subclass a `TestJdwp` class. The core functionality is done by the `JdwpTestListener`, which redirects a call to `JdwpVerifier.verify` method from the JPF to the underlying VM where `JdwpVerifier.verifyOutsideOfSuT` method is executed. All the arguments are translated to their respective representation in JPF as `ElementInfo` instances. However, primitive types are not supported. The algorithm of the translation is based on a reflection and thus, it is not designed for excessive testing.

JPF JDWP extension also takes an advantage of existing Eclipse JDT JDI tests. They are modified in order to test JPF and the JPF JDWP Agent together with the Eclipse JDT debugger. These classes are located in the dedicated project JPF JDWP Integration Tests project, which is further explained in the Appendix III.II section.

5 Critical assessment

For this thesis, we decided to integrate JPF into JPDA at the JDWP level instead of the two remaining standard options, which are JDI and JVM TI levels. The former one brings too many constraints on the debugger side and does not benefit from what the architecture targets. The latter option provides broader functionality than what JPDA requires and, moreover, it is the standard interface for the communication with JVM. However, it is intended to be a native interface. As a consequence, we rejected this option. Other possible non-standard solutions were also rejected in the favor of the JDWP solution.

The JDWP level integration is suggested by JPDA for those JVMs that do not implement the JVM TI. The functionality, which is required by the JDWP Specification, is referred as the JDWP agent. The agent, which is called JPF JDWP Agent, is based on the GNU Classpath JDWP project (see 1.2). Despite of its original purpose to serve at the program level, we incorporated it at the JPF (i.e., VM) level. Its native interface intended for an integration with a VM and insufficient design of internal objects led us, however, to reuse only a minor part of the project. The resulting JPF JDWP Agent allows JDWP compliant Java debuggers to debug a program being verified in JPF.

To determine the quality of the agent design and implementation, its compliance with the JDWP Specification and its overall usability for debugging, we used sources of existing native implementations of JDWP agents, the Eclipse JDT Java debugger sources and the source of its JDI implementation. Furthermore, we created unit tests to prevent a regression while JPF evolves. Moreover, we adopted more than two hundred tests from the Eclipse JDT project for JDI implementation testing (see Appendix III.II), which serve as integration tests of the JPF core and the JPF JDWP agent together with the Eclipse JDT debugger. At last, we also extensively user tested the running agent from the Eclipse IDE with its JDT Java debugger.

Whereas the agent solution works with any JDWP compliant Java debuggers, we decided to extend the Eclipse JPF project to demonstrate the wide potential of JPF debugging. The main features are debugging of JPF verification and JPF itself, error trace store and reply, and an ability to stop on a property violation (see Appendix III.I).

The main imperfection of the JPF JDWP Agent is caused by the problem with reappearing threads (see 3.2.2) that was solved as a workaround, yet in accordance with the JDWP specification. However, we cannot expect all the JDWP debuggers to be compatible with such a behavior. The same applies for the Eclipse JDT debugger where a race condition we discovered in the JDI thread management prevents it from working well. As a result, only the debugging of a JPF error trace replay is fully supported.

Considering a standard use case of the JPF verification, which either requires the best possible performance or is trying to make a contribution to the model checking field of science, the limitation of an error trace only debugging is acceptable.

Another imperfection is that we implemented only the JDWP capabilities required and used by the Eclipse JDT debugger. Therefore, the agent implementation does not feature monitor events, particularly because of the inability to test them with this debugger.

5.1 Reusability

Because of the tight integration of the agent with JPF, the JDWP agent is not easily reusable in different VM implementations. In order to create independent JDWP agent, JPF would first need to provide some interface. It could be API similar to JVM TI (although in Java) or to provide better encapsulation of JPF internals. In this work, we did not have ambitions to add an abstraction layer between JPF extensions and JPF core.

It is also not possible to merge the results of this work back to the GNU Classpath JDWP project due to the extensive direct association of the JDWP classes with the JPF classes.

5.2 Performance

Even though performance is not the primary target of the JPF JDWP agent implementation[17], we still saw performance as an important objective while designing the agent. The most critical is event-processing flow, which is optimized to delay the processor demanding tasks to its end when an event is known not to be discarded.

The reappearing threads (see 3.2.2) problem causes the main performance degradation during the debugging of a state space traversal. The excessive traffic of thread start and death events may even cause debugger instability. We also found the use of conditional breakpoints lead to poor performance as well. However, this is also an issue with standard Java debuggers while debugging a program in a common JVM.

We also did profiling of the agent and then applied the outcome in order to prevent unnecessary bottlenecks in the implementation of the application.

5.3 Related work

This subchapter discusses work that has already been done on a similar subject though it was never related directly to JPDA. After all, the assignment for this study developed from the following related work.

[17] An extensive use of switch statements have a potential to perform better than a use of hash maps.

5.3.1 JPF Inspector

JPF Inspector [44], also referred as *jpf-inspector*, is a JPF extension that together with JPF Shell [45] provides CLI to manipulate and inspect JPF verification. Its main features are [46] breakpoints, program state inspection and also its limited modification, forward and backward single step traversal of program state and more.

Apparently, JPF Inspector is a mature project that has a great potential for advanced debugging and inspection of a program state during JPF verification. There were even plans for creating a GUI or even integration with common Java IDEs (suggestion on this topic is made in the chapter 6). The original idea of this study considered *jpf-inspector* as a possible foundation for the JPDA interface implementation. More specifically, it was to use it as an abstraction provider for direct access to the JPF internals through its internal API.

The programming interface, which JPF Inspector provides, is a command based API used between its client and server parts. We decided; however, to implement the solution for this study from scratch since the JPF Inspector API is too closely designed for the CLI needs it provides. JPF Inspector implements 18 commands, whereas JDWP specifies 88 commands and, moreover, it does not cover all JPDA functionalities at the granularity this thesis requires.

JPF Inspector was nonetheless used as a source of inspiration and a reference implementation of JPF internals manipulation.

5.3.2 Debug 4 JPF

The *debug4jpf* Java PathFinder extension is a part of a thesis named *Presenting results of software model checker via debugging interface* [47]. This extension, together with the second part of this thesis named *JPFDeb.core,* provides a subset of capabilities that are specified by JPDA. This thesis aims at just one IDE, in particular, Eclipse.

Apparently, the author of this study is aware of JPDA as well as his advisor, since it states the original proposal was to use JPDA for its realization. However, Kohan declares: "Disadvantages of using JPDA would be complexity and cumbersomeness of design and implementation" [47] and continues: "the implementation would be inadequately difficult or even not possible". On the other hand, he also points out several advantages of the JPDA solution such as remote debugging or a the compatibility with all JPDA debuggers. These advantages are nonetheless neglected since he decided to design and to use a proprietary string-based protocol.

In this study, we proved it is possible to integrate JPF with JPDA in a scope of single master thesis. We are not, however, infirming the statement regarding the complexity as the codebase of this work ended quite large (see VI Code statistics).

In this thesis we also reveal it is possible, contrary to Kohan's reasoning [47], to implement a stepping (3.2) of a program being verified in JPF.

5.3.3 Eclipse JPF

Eclipse JPF [48] and Netbeans JPF [49] plug-ins are extensions that provide very simple functionality for running JPF verification based on an application property file, as stated in the cited sources, from an IDE.

These extensions are perfectly compatible with the results of this study as the JDWP back-end can be enabled by a declaration of another listener in the application property file (section 4.3) as well as by a command line argument.

We also extended the Eclipse JPF plug-in (precisely, it was rewritten) to demonstrate the capabilities of JPDA in JPF through a user-friendly GUI [50]. The realization of this extension was done aside of this study so that the integration of JPF to the JPDA is not disguised by another project that is complex enough on its own.

5.3.4 Java Debug Trace

An assignment for a project named *Java Debug Trace* (abbreviated as *JTG*), was published at Charles University around 2009 [18] as a *Software Project* [51] course assignment [52]. Besides the implementation of JPDA for JPF, this assignment also suggests to implement JVM TI in JPF for the sake of profiling (to reveal certain metrics to IDEs such as time and number of invocations of methods) and finally, to introduce additional interface for a navigation along a trace (e.g., step back functionally, which is not present in JPDA).

This project was never completed as it was cancelled [53].

This study fulfills only the implementation of JPDA for JPF; however, a discussion on JVM TI topic is done in section 2.2. Finally, an extension of the JDWP protocol to support JPF specific functionality (e.g., step back command) is proposed in chapter 6.

[18] This is a date of last modification of the document. It is nonetheless marginal information since the project was never completed.

6 Further work

With a complete implementation of the JDWP protocol for Java PathFinder, many doors are opening as JPF became compliant with a global Java debugging standard, the JPDA. It is, however, required to keep JPF JDWP Agent up to date with the changes made to JPF itself and also with future JDWP standards new Java releases will bring.

6.1 Enhancing JPDA

The first thing is obviously to extend the JDWP specification in order to support JPF specific behavior and its needs. All of this must be; however, backward compatible so that standard debuggers may still be used for JPF debugging. It is equally important to cautiously take into consideration possible growth of the specification with new versions of Java language.

We suggest proposing an extension of the JDWP protocol specification with new commands and event types for JPF state space traversal control and its ability to notify about it.

Apparently, the JDI specification as well as its implementation (i.e., the JPDA front-end) must incorporate the JDWP changes, which is the basic cause of its complexity. The optimal solution is to enrich the JDI model and to coexist with any existing JDI implementations[19] in a way that the same extension implementation can be reused by all Java debuggers. In addition, licensing issues may arise, as Oracle is the owner of the JPDA as a whole, as well as of the standard JDI implementations provided in JDKs.

Finally, the debugger itself (i.e., the UI) must reflect the additional functionality and present it to the user.

We believe, in case of Eclipse platform, the suggested change would be possible to build on top of the current Eclipse JPF plug-in [50] without a modification of the Eclipse JDT plug-in. Additionally, Eclipse community is opened to improvements hence API changes could be accepted.

6.2 Other work

The changes to the JPDA stack should also go hand in hand with the JPF Inspector project. We believe that extended JDWP and JDI specifications are the right

[19] Besides the standard JDI implementation provided by Oracle in every JDK release, some debuggers come with their own implementation (e.g., Eclipse JDT).

interfaces for even such advanced and tightly JPF bounded tool as JPF Inspector is. After all, to bring the functionality of JPF Inspector to Java IDEs is the vision of its creators.

We also suggest modifying JPF class model that represents the standard JVM entities in order to be analogical to the model we can see in JDI or JNI. Such a change would allow JPF JDWP implementation to map more elegantly JPF internals to the JDI and vice versa.

Summary and Conclusion

This work integrates Java PathFinder model checker with the JPDA debugging standard. From now on, JDWP compliant Java debuggers may observe and control[20] JPF verification.

The JPF JDWP Agent is a JPF extension that implements JDWP protocol on top of JPF Core. As such, it empowers JPF verification for a debugging with following features: stepping (in, over and out), various kinds of breakpoints with various modifiers (object instances, threads, exception throws, counters and also condition evaluations), object and call stack inspection, values modification, arbitrary code evaluation or execution and lot more. All of this is available through a GUI exactly the same way as when standard Java programs are debugged.

The work has fulfilled all the Goals listed in the Introduction, as elaborated in assessment chapter 5. The JDWP interface was chosen among other possibilities from the JPDA stack and its specification was implemented without any compromises. Finally, considering also Eclipse JPF extension (included in Appendix III.I) and JPF JDWP Integration Tests project (included in Appendix III.II), a whole JPDA for JPF solution is available for public use and further development.

However, several limitations remain. It was made impossible to override a thread scheduling from a debugger. JPF debugging while it backtracks is supported only partially since the implementation uses only JPDA available methods to reflect such VM state changes. Finally, JDWP for JPF is fully tested only against the Eclipse JDT debugger.

The most challenging part of this work was to find an interpretation of the JDWP specification so that the realization works with other debuggers. That involved an examination of various Oracle (former Sun Microsystem) specifications, investigation of other JDWP implementations in native languages, exploration of Java debugger implementations and finally though definitely the most frequently, debugging of a Java debugger while it debugs a debuggee running in JPF that is also being debugged.

Besides this very thesis, a project JPDA for JPF [54] was accepted by the Java PathFinder team and it was successfully completed in the program Google Summer of Code 2013.

[20] State space traversal control is not a part of the JPDA. The support of it is nonetheless proposed in the section 6.1.

List of Figures

Figure 1. The JPDA structure 4
Figure 2. JPF architecture [21] 11
Figure 3. Traversing across the state space. [22] 12
Figure 4. Standard JDWP data flow [36] 22
Figure 5. Command execution diagram [36] 22
Figure 6. Event processing diagram [36] 23
Figure 7. JPF JDWP Agent Architecture 31
Figure 8. Threads interaction diagram 32
Figure 9. Command execution diagram 33
Figure 10. Event processing diagram 34
Figure 11. Jdwp back-end initialization 35
Figure 13. The simplified Value class model 38
Figure 14. The simplified Identifier class model 39
Figure 15. Object ID class model 40
Figure 16. Reference Type ID class model 40
Figure 17. Identifier Manager class model 41
Figure 18. Exception class model 42
Figure 19. Eclipse JPF JDWP debugging experience 58
Figure 20. Eclipse JPF verification execution 58

List of Tables

Table 1. Events and resolution 19
Table 2. JDWP Agent modules 21
Table 3. JPF entities to JDWP IDs mapping 28
Table 4. JDWP Events to JPF listeners mapping 29
Table 5. List of abbreviations....................53
Table 6. JDWP configuration property suboptions [2] 55
Table 7. Code statistics....................60

List of Abbreviations

Abbreviation	Meaning	Location
JPF	(Java PathFinder) It is a explicit state model checker which works as a JVM.	s. 1.3
JPDA	(Java Platform Debugger Architecture) The architecture designed by Sun (Oracle) for debugging of Java applications.	s. 1.1
JDI	(Java Debug Interface) Pure Java interface from JPDA the debuggers may use.	s. 1.1.1.1
JDWP	(Java Debug Wire Protocol) A protocol from JPDA that is used between a debugger and a debuggee.	s. 1.1.1.2
JVM TI	(Java Virtual Machine Tool Interface) A native interface debuggers or debug agent use to communicate with the JVM and vice versa.	s. 1.1.1.3
JDT	(Java Debug Tools) Eclipse Java debugging tools project	p. 17
JVM	(Java Virtual Machine) It runs Java programs.	p. 4
JNI	(Java Native Interface) Native application may use this interface to communicate with Java application and vice versa.	p. 7
VM	(Virtual Machine) Is used as a synonym to Java VM	p. 1
UI	(User interface) An interface user interacts with when working with a program.	p. 1
IDE	(Integrated Development Environment) An application that servers for editing program sources and provides multiple tools developer needs (refactoring, debugging, testing, etc.)	p. 1
SuT	(System under Test) The program that is being verified in JPF	p. 39

Table 5. List of abbreviations

Appendix

I Links

- The attachment of this thesis: https://bitbucket.org/stepanv/mthesis

JPF JDWP Agent

- Project: https://bitbucket.org/stepanv/jpf-jdwp
- Wiki: https://bitbucket.org/stepanv/jpf-jdwp/wiki/Home
- Sources: ssh://hg@bitbucket.org/stepanv/jpf-jdwp
- Javadoc: http://stepanv.bitbucket.org/jpf-jdwp/javadoc/

Eclipse JPF

- Project: https://bitbucket.org/stepanv/eclipse-jpf
- Wiki: https://bitbucket.org/stepanv/eclipse-jpf/wiki/Home
- Sources: ssh://hg@bitbucket.org/stepanv/eclipse-jpf
- Javadoc: http://stepanv.bitbucket.org/eclipse-jpf/javadoc/
- Eclipse 4.3 Kepler update site:
http://stepanv.github.io/eclipse-jpf-updatesite/update/

JPF JDWP Integration Tests project

- Project: https://bitbucket.org/stepanv/jpf-jdwptest
- Sources: ssh://hg@bitbucket.org/stepanv/jpf-jdwptest

II User manual

In order to use JPF JDWP for debugging purposes it is required to have multiple tools and programs installed. Furthermore, JPF Core and JPF JDWP sources are required. Finally, user needs a debugger and a Java program to verify. The link for the whole documentation can be found in the I Links section of Appendix.

There is however much easier option, which is to use the Eclipse JPF plug-in that comes with embedded JPF and JDWP runtimes. With this plug-in, user can start verification from Eclipse 4.3 Kepler without any other requirements.

II.I JPF JDWP Agent invocation

The JPF JDWP Agent uses a configuration property for its initialization. The configuration property can be specified either as a System property `jdwp` or a JPF

property jpf-jdwp.jdwp. The format and options follow the standard, which is specified by Oracle in the JPDA documentation [2].

The format of the property is: <name1>[=<value1>],<name2>[=<value2>]...

The table below describes the options that can be used:

Name	Required?	Default value	Description
transport	yes	none	Name of the transport to use in connecting to debugger application. Only dt_socket is supported.
server	no	"n"	If "y", listen for a debugger application to attach; otherwise, attach to the debugger application at the specified address. If "y" and no address is specified, choose a transport address at which to listen for a debugger application, and print the address to the standard output stream.
address	yes, if server=n no, otherwise	""	Transport address for the connection. If server=n, attempt to attach to debugger application at this address. If server=y, listen for a connection at this address.
suspend	no	"y"	If "y", *VMStartEvent* has a *suspendPolicy* of *SUSPEND_ALL*. If "n", *VMStartEvent* has a *suspendPolicy* of *SUSPEND*_NONE.

Table 6. JDWP configuration property suboptions [2]

Examples:

```
+jpf-jdwp.jdwp=transport=dt_socket,server=y,suspend=y,address=8000
```

Listen for a socket connection on port 8000. Suspend the VM to wait for a debugger attach.

```
+jpf-jdwp.jdwp=transport=dt_socket,server=n,address=localhost:8000
```

Attach to a listening debugger at localhost, port 8000.

II.II General use

Required tools

- JDK7u25+
- Ant 1.8.3+
- Mercurial 2.3.2+
- Java debugger

Environment preparation

1. Obtain sources from Mercurial repositories
   - jpf-core (from http://babelfish.arc.nasa.gov/hg/jpf/jpf-core)
   - jpf-jdwp (from ssh://hg@bitbucket.org/stepanv/jpf-jdwp)
2. Setup environment *.jpf/site.properties* file with extensions: (http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/site-properties)
   - with *jpf-core* and
   - *jpf-jdwp* extensions as
3. Build both projects with Ant (http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/build)

Preparation for debugging

1. Create application property file for the Java application (http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/config)
2. Enable the listener `gov.nasa.jpf.jdwp.JDWPListener`
3. Add the `jpf-jdwp.jdwp` property with parameters that suit the user needs (i.e., `transport=dt_socket,server=<y/n>,suspend=<y/n>,address=<hostname>:<portnumber>` as described in II.I JPF JDWP Agent invocation section)

Starting the debuggee

1. Depending on the JDWP setup, user may start the JPF verification (http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/run)

Starting the debugger

1. Depending on the JDWP setup, user may attach the debugger to JPF. This is debugger specific.

II.III Using Eclipse IDE and Eclipse JPF

Requirements

- JDK7u25+
- Ant 1.8.3+
- Eclipse 4.3 Kepler

Installation

Install the Eclipse JPF plug-in from the update site (see I Links).

Debugging

1. Create application property file (`.jpf`) for the Java application that needs to be verified.
   (http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/config)
2. Open *Java PathFinder Verification* debugging launch dialog for the `.jpf` file
3. Start the debugging

III Parallel work

Besides the JDWP for JPF project, we started two projects in order to support the core work of this study. The former project is an extension of Eclipse JPF project and the latter one is a project for an integration testing of the JPF JDWP Agent.

III.I Eclipse JPF project

Eclipse JPF project extends the original work made by Sandro Badame and Peter Mehlitz [48]. The purpose of this extension was to add a debugging support into the JPF verification launcher. As a result, we completely rewrote the plug due to its poor design and incorrect use of Eclipse plug-in API.

The documentation and other links are listed in the ILinks section.

The main features are:

1. Debugging support (of the program being verified in JPF and JPF itself)
2. New ways to run the verification (through the standard launch dialogs)
3. Complex verification configuration (as a launch configuration)
4. Embedded runtimes (user does not need JDWP or JPF installed)
5. Console output with pattern trackers for links into the sources
6. (original) JPF Extension stub creator
7. (original) Legacy mode (starts the verification exactly the same way as the predecessor did)

Following figures illustrate capabilities of JPF debugging. In the Figure 19, we can see the verification is paused because of a property violation as the threads are in a deadlock. The Figure 20 shows how to start the verification.

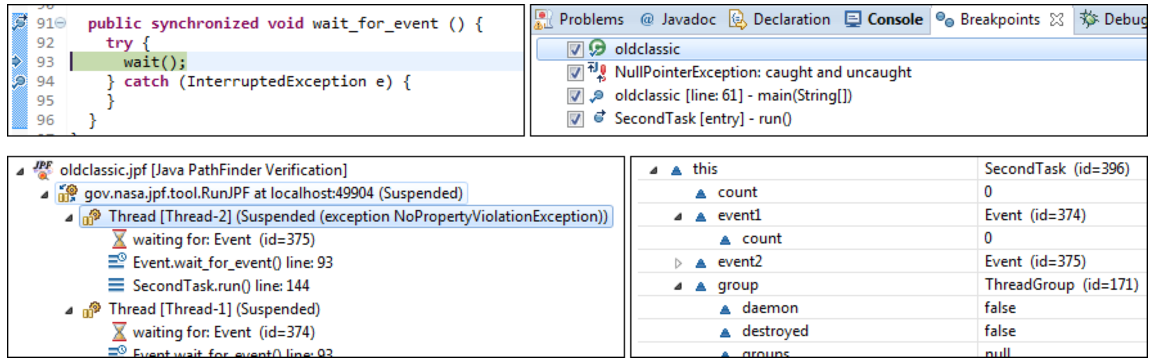


Figure 19. Eclipse JPF JDWP debugging experience

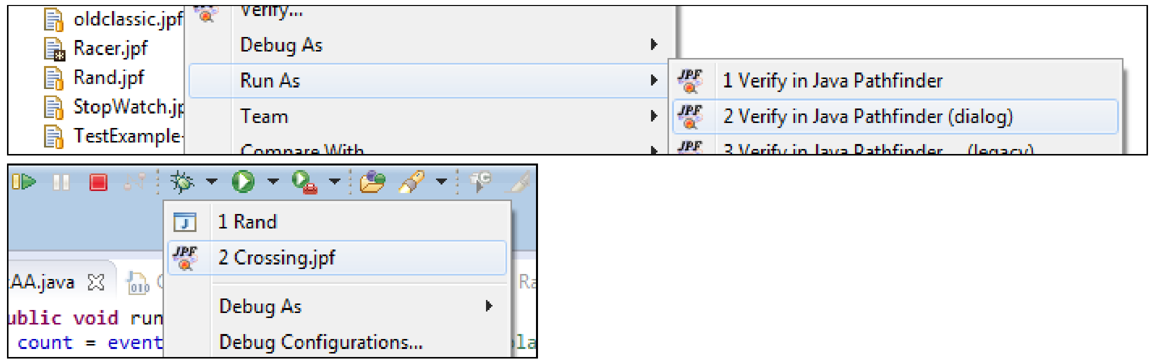


Figure 20. Eclipse JPF verification execution

III.II JPF JDWP Integration Tests project

The JPF JDWP Integration Tests project adopts a subproject of Eclipse JDT [5] that contains a test framework, which is designed for testing of the Eclipse JDI implementation.

We modified the test framework so that the JDI is tested against a program being verified in JPF. As a result, we have more than two hundred fifty tests where JPF together with the JPF JDWP agent are extensively tested. Every JDWP command is executed multiple times with different values in a different debugging context. Moreover, every event is triggered multiple times in a different program state. It tests both positive and negative scenarios.

The idea of this test framework is as follows. Every test starts a JPF verification of a program. Once the program reaches a particular state, the test begins. During the test, JPF continues the verification or pauses depending on the test specifics. When the test ends, JPF verification is terminated.

The key property of this idea is that the program, which is being verified in JPF, is always the same. It already contains parts of code the test instructs it to execute. The program runs in an infinite loop, checking a value of a variable, and receives commands that control the program execution. These commands are based on a particular JDWP command through which a debugger can change a value of a variable. The test uses

these commands to modify this variable, which is the way in which the program is controlled to execute exactly what the test is design to verify.

The location of this project is listed in the I Links section. These tests can be executed either from a command line using Ant or from Eclipse IDE.

IV Running example

This example shows how to debug in Eclipse IDE `my.packagge.MainClass` class that is included in the JDWP project. It is however assumed the *jpf-jdwp* and the *jpf-core* extensions are correctly installed.

To enable JDWP in the application

In Eclipse, create new *Debug/Run - Java Application* configuration that will run JPF and the program in it:

1. Main class: `gov.nasa.jpf.jdwp.JDWPRunner`
2. As *Program arguments*, set (do not substitute the placeholder/variable; Eclipse will do it for you automatically):
   `+target=my.packagge.MainClass`
   `+classpath=+,${workspace_loc:jpf-jdwp/build/examples}`
3. Enable JDWP by adding another property to *Program arguments*:
   `+jpf-jdwp.jdwp=transport=dt_socket,server=y,suspend=y,address=8000`
4. *Run* it (you can also *Debug* it but that means you will debug JPF itself; including JDWP implementation).
   It will stay suspended until you attach a debugger.
5. Now, your application verification is about to start.
   It will wait on port 8000 for a debugger attach.

To attach a debugger

Create new *Debug - Remote Java Application* configuration that will attach the debugger to the application that is about to start.

1. Put a breakpoint into the `my.packagge.MainClass` so that it gets suspended when the breakpoint is hit
2. *Connection Properties* stay default: Host `localhost` and Port `8000`
3. *Debug* it

V CD content

This thesis contains attached CD where attachments are. Besides a readme, where the content is explained in detail, it contains:

- Sources of *jpf-jdwp*, *eclipse-jpf* and *jpf-jdwptest* projects
- The text of this in thesis in pdf and Enterprise Architect project

- One-click example which demonstrates JDWP for JPF in Eclipse IDE
- Example project with .jpf verification files and Java classes to verify

VI Code statistics

Module	Java sources	Size	Comment Lines	Lines of Code[21]
JPF JDWP	206	867 KB	12327	10453
GNU sources	10	59 KB	905	730
genuine agent runtime[22]	174	697 KB	10552	7641
ECLIPSE JPF	39	275 KB	1843	4916
original code	13	61 KB	380	1359

Table 7. Code statistics

[21] Calculated using *cloc-1.60*

[22] Genuine agent runtime refers to the sources of the agent implementation (i.e., without test and example sources) that we created during this work (i.e., without the GNU Classpath JDWP sources).

Bibliography

[1] Peter Mikhalenko, "Debug your Java code with ease using JPDA," in *TechRepulic*, 2006. [Online]. http://www.techrepublic.com/article/debug-your-java-code-with-ease-using-jpda/

[2] Oracle. (2013) Java Platform Debugger Architecture. [Online]. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/

[3] Oracle. (2013) Java Debug Interface: Specification. [Online]. http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

[4] Edwin Torres, "Five of the Best Java IDEs," in *Yahoo Voices: voices.yahoo.com*, 2013. [Online]. http://voices.yahoo.com/five-best-java-ides-12343888.html

[5] Eclipse. (2013) Github. [Online]. https://github.com/eclipse/eclipse.jdt

[6] Oracle. (2013) Netbeans. [Online]. https://netbeans.org/

[7] Jetbrains. (2013) GitHub. [Online]. https://github.com/JetBrains/intellij-community

[8] Takipi, Inc. (2013) Takipi - Server Debugging Made Easy. [Online]. http://www.takipi.com/faq.html

[9] Oracle. (2011) JDK 6 Java Platform Debugger Architecture (JPDA)-related APIs & Developer Guides. [Online]. http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html

[10] Oracle. (© 2013) Oracle Technology Network. [Online]. http://www.oracle.com/technetwork/java/javase/versioning-naming-139433.html

[11] (© 2013) Oracle Technology Network. [Online]. http://www.oracle.com/technetwork/java/javase/namechange-140185.html

[12] Sharon Zakhour, Sowmya Kannan, and Raymond Gallardo, *The Java tutorial: a short course on the basics*, Fifth edition. ed., 2013. [Online]. http://www.oracle.com/technetwork/java/javase/java-tutorial-downloads-2005894.html

[13] (2009) GNU Classpath. [Online]. http://www.gnu.org/software/classpath/home.html

[14] Free Software Foundation, Inc. (2009) GNU Project. [Online]. http://www.gnu.org/software/classpath/downloads/downloads.html

[15] (© 2007) Jikes RVM. [Online]. http://jikesrvm.org/

[16] "Debugging the RVM," in *Home - RVM - Codehaus*, 2011. [Online]. http://docs.codehaus.org/display/RVM/Debugging+the+RVM

[17] (2007) Sable VM. [Online]. http://sablevm.org/

[18] "Package gnu.classpath.jdwp," in *sablevm-classlib Documentation*, 2007. [Online]. http://sourcecodebrowser.com/sablevm-classlib/1.13/namespacegnu_1_1classpath_1_1jdwp.html

[19] NASA. (2012) NASA. [Online]. http://ti.arc.nasa.gov/tech/rse/vandv/jpf/

[20] Corina S. Păsăreanu, Pavel Parízek, and Artem Khyzha, "Abstract Pathfinder," *Abstract Pathfinder*, no. 6068, 2012. [Online]. http://ti.arc.nasa.gov/publications/6068/download/

[21] (2003) Java PathFinder (JPF). [Online]. http://javapathfinder.sourceforge.net/

[22] The Java PathFinder team. (2009) Java Path Finder. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/testing_vs_model_checking

[23] "JPF Developer Guide," in *JPF WIKI*, 2010. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/start

[24] "JPF Runtime Modules," in *JPF WIKI*, 2009. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/modules

[25] Oracle. (2007) Java SE 7 Java Virtual Machine Tool Interface (JVMTI): related APIs and Developer Guides. [Online]. http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[26] Gil Fink, "Applying Strategy Pattern Instead of Using Switch Statements," in *Gil Fink's Blog*, 2009. [Online]. http://blogs.microsoft.co.il/gilf/2009/11/22/applying-strategy-pattern-instead-of-using-switch-statements/

[27] "JPF and Google Summer of Code 2010," in *events/start – Java Path Finder*, 2010. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/events/soc2010

[28] Oracle. (© 2013) OpenJDK. [Online]. http://openjdk.java.net/

[29] The Apache Software Foundation. (© 2003-2010) Apache Harmony: Open Source Java Platform. [Online]. http://harmony.apache.org/

[30] The Eclipse Foundation. (© 2013) Eclipse: The Eclipse Foundation open source community website. [Online]. http://www.eclipse.org/

[31] Oracle. (2013) Java Native Interface Specification. [Online]. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html

[32] IBM. (© 2000) Github. [Online]. https://github.com/eclipse/eclipse.jdt.debug/blob/master/org.eclipse.jdt.debug/jdi/org/eclipse/jdi/internal/jdwp/JdwpReplyPacket.java

[33] Oracle, *The Java virtual machine specification*, 7th ed. Upper Saddle River, NJ:

Addison-Wesley, 2013. [Online]. http://docs.oracle.com/javase/specs/jvms/se7/html/

[34] Eclipse. (2013) Bugzilla Main Page. [Online]. https://bugs.eclipse.org/bugs/show_bug.cgi?id=418746

[35] Eclipse. (2009) Bugzilla Main Page. [Online]. https://bugs.eclipse.org/bugs/show_bug.cgi?id=272494

[36] Yu Junjie and Xu Wise, "深入 Java 调试体系，第 3 部分: JDWP 协议及实现," in *Developer Works*, 2009. [Online]. http://www.ibm.com/developerworks/cn/java/j-lo-jpda3/

[37] IBM. (2013) Eclipse documentation. [Online]. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fdebug%2Fref-debug.htm

[38] Pavel Jančík. (2012) jpf/jpf-inspector. [Online]. http://babelfish.arc.nasa.gov/hg/jpf/jpf-inspector/file/3c8319a3579d/src/main/gov/nasa/jpf/inspector/server/jpf/StopHolder.java

[39] Peter Mehlitz, Re: jpf-core bugfix, documentation, 2013-08.

[40] Yourkit. (© 2003-2013) Yourkit - Java Profiler. [Online]. http://www.yourkit.com/docs/80/help/overhead.jsp

[41] Jari Aarniala, "Instrumenting Java bytecode," in *Department of Computer Science*, 2005. [Online]. http://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Aarniala-instrumenting.pdf

[42] Joshua Bloch, *Effective java*, 2nd ed. Upper Saddle River: Addison-Wesley, c2008.

[43] QOS. (2013) Simple Logging Facade for Java. [Online]. http://www.slf4j.org/faq.html#logging_performance

[44] "JPF Inspector," in *Java Path Finder*, 2011. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-inspector

[45] "JPF Shell," in *Java Path Finder*, 2009. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-shell

[46] Pavel Jančík, Kofroň Jan, and Pavel Parízek, "Advanced Debugging with JPF-Inspector," in *Advanced Debugging with JPF-Inspector*, 2011, p. 5. [Online]. http://plg.uwaterloo.ca/~pparizek/papers/jpf11-inspector.pdf

[47] Tomáš Kohan, *JPF results via debugging interface Presenting results of software model checker via debugging interface*. Prague, 2012, Master Thesis. Charles University. [Online]. https://is.cuni.cz/webapps/zzp/search

[48] the Java PathFinder team. (2013) Java Path Finder. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/eclipse-jpf

[49] The Java PathFinder team. (2009) Java Path Finder. [Online]. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/netbeans-jpf

[50] Štěpán Vávra. (2013) Eclipse JPF Wiki. [Online]. https://bitbucket.org/stepanv/eclipse-jpf/wiki/Home

[51] "Software Project," in *Student Information System*, Prague, 2013. [Online]. http://is.cuni.cz/studium/eng/predmety/index.php?do=predmet&kod=NPRG023

[52] Tomáš Poch, "Java Trace Guide," in *Stránka projektové komise*, 2009. [Online]. http://ksvi.mff.cuni.cz/~holan/SWP/zadani/jtg.pdf

[53] "Project List," in *Stránka projektové komise*, Prague, 2009. [Online]. http://ksvi.mff.cuni.cz/~holan/SWP/vedouci.htm

[54] Štěpán Vávra, "JPDA for JPF," in *Google Summer of Code 2013,* 2013. [Online]. http://www.google-melange.com/gsoc/proposal/review/google/gsoc2013/stepanv/1

[55] "Java Naming," in *java.com: Java + You*, 2005. [Online]. http://www.java.com/en/about/javanaming.jsp

[56] "Sun delivers next version of the Java platform," in *Internet Archive: Wayback machine*, 2007. [Online]. http://web.archive.org/web/20070816170028/http://www.sun.com/smi/Press/sunflash/1998-12/sunflash.981208.9.xml

[57] Oracle. (© 1995) Java SE at a Glance. [Online]. http://www.oracle.com/technetwork/java/javase/tooldescr-136044.html

[58] IBM. (2012) Eclipse Help - Kepler. [Online]. http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/misc/naming.html?cp=2_1_3_3

[59] YourKit Java Profiler. (© 2003-2013) Java Profiler. [Online]. http://www.yourkit.com/docs/80/help/profiling_j2ee_remote.jsp

[60] Oracle. (2013) The Java® Virtual Machine Specification. [Online]. http://docs.oracle.com/javase/specs/jvms/se7/html/