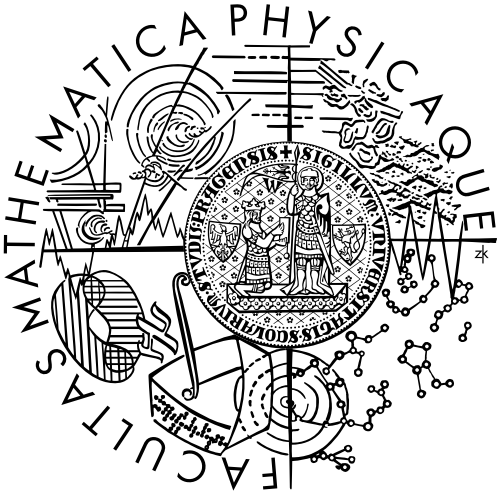
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Kohan

Presenting Results of Software Model Checker via Debugging Interface

Department of Software Engineering

Supervisor of the master thesis: RNDr. Ondřej Šerý, Ph.D.

Study programme: Computer Science
Specialization: Software Systems

Prague 2012

I would like above all to thank my supervisors, RNDr. Ondřej Šerý, Ph.D. and RNDr. Tomáš Poch, Ph.D., for their guidance in shaping my research, their continuous encouragement and major advices.
I am very grateful to all the people who supported my work by their help and advice too.

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

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, April 11, 2012 Tomáš Kohan

Název práce: Prezentace výsledků kontroly softwarového modelu skrz ladící rozhraní
Autor: Tomáš Kohan
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Ondřej Šerý, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je navrhnout a implementovat nové ladící rozhraní programu Java PathFinder. Vhodným prostředím pro toto rozhraní byl zvolen vývojový nástroj Eclipse. Vytvořené rozhraní graficky vizualizuje výstupy programu JPF a jednotlivé detaily stavu pozastaveného virtuálního stroje (JVM), zvláště pak seznam proměnných a jejich hodnot. Za tímto účelem jsou vytvořeny dva podprojekty, a to debug4jpg a JPFDeb.core. Projekt debug4jpf kontroluje a komunikuje s instancí JPF. JPFDeb.core pak ve formě zásuvného modulu pro Eclipse poskytuje takové uživatelské rozhraní, které je podobné standardnímu rozhraní ladícího programu pro Javu. Oba projekty mezi sebou komunikují přes ad hoc komunikační protokol, který byl navržen pro tento účel.

Klíčová slova: Java, verifikace, kontrola modelu, JPF, ladící rozhraní

Title: Presenting results of software model checker via debugging interface
Author: Tomáš Kohan
Department: Department of Software Engineering
Supervisor of the master thesis: RNDr. Ondřej Šerý, Ph.D., Department of Distributed and Dependable Systems

Abstract: This thesis is devoted to design and implementation of the new debugging interface of the Java PathFinder application. As a suitable inteface container was selected the Eclipse development environment. The created interface should visualize results of JPF and details of paused JVM state, especially a list of variables and their values. Two subprojects were created, i.e. debug4jpf and JPFDeb.core. The first one is responsible for controlling and communication with the JPF instance. The latter one is an Eclipse plugin and provides user interface which is similar to the interface of standard Java debugger. These two components communicate with each other by using the ad-hoc communication protocol created for this purpose.

Keywords: Java, verification, model checker, JPF, debugging interface

Contents

Introduction

Nowadays, a variety of applications control important machines and processes in our lives, e.g. surgical robots, spacecrafts, finance, army etc. Any, even small, error can cause incalculable losses. Therefore, testing and, especially, verification and model checking is being important part of the application development process.

Java PathFinder (JPF) is one of the well-known model checking software. It is predominantly used in academic environment, but it is gradually spreading to the commercial sector as well.

This thesis is devoted to design and implementation of a new debugging interface to JPF in order to improve and clarify the output of the test for the user. The current version of JPF provides a textual output which cannot be easily interpreted by the user and also not all details which are available in JPF internal data structure are present.

Our approach is to integrate JPF into widely used graphical development environment (Eclipse) and visualize all runtime items which are available in JPF – threads, stack frames and variables.

The first chapter describes the testing process in general. It is a short introduction to testing topics with greater emphasis on verification. Verification can be considered to be a type of testing or separate phase of project. In this thesis is verification considered as a type of testing. Moreover, there are the difficulties with states described as the cornestone of the verification.

JPF, its internals and usage are described in the second chapter. The most of the chapter is devoted to the JPF architecture and the data model, which are used in the proposed approach.

The third chapter lists the used components of the Eclipse Debug framework. The graphical part of our approach is making use of this framework to integrate into Eclipse development environment. The launch framework is used for starting and inicialization of the implementation, while debug model and variables components visualize all threads, stack frames and variables.

The actual application design is described in the fourth chapter. The application is divided into two main components, JPFDeb.core and debug4jpf. The JPFDeb.core component is an Eclipse plugin which is responsible for communication with the Eclipse instance. In addition to this, the debug4jpf component communicates with and controls the JPF instance. There is an ad-hoc communication protocol designed and it is used for the communication between JPFDeb.core and debug4jpf.

The fifth chapter lists some of the implementation details and the user

guide. There are several sample projects included in the attached CD. The user guide is a step-by-step instructions list of how to run these sample projects, or other Java projects whose model should be checked by JPF.

To summarize, the main goals of this thesis are:

- Design and implement new debugging interface of JPF as a part of an integrated development environment.
- Visualize paused JVM state and JPF results in order to help the user to inspect the error state.
- Add variables and their values as a part of this visualization.

Chapter 1

Java Software Testing and Verification

Based on the [1], *Java* (generally *Software*) *Testing* is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software faults.

Testing can never completely establish the correctness of computer software. Instead, it furnishes a criticism or comparison that compares the state and behaviour of the product against oracles – principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Over its existence, computer software has continued to grow in complexity and size. Every software product has a target audience. For example, the audience for a video game software is completely different from a banking software. Therefore, when an organization develops or otherwise invests in a software product, it presumably must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment.

1.1 Java (Software) Testing Topics

Scope – Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do.

Defects and failures – Software faults occur through the following proc-

esses. A programmer makes an error, which results in a defect in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A single defect may result in a wide range of failure symptoms.

Compatibility – A frequent cause of software failure is compatibility with another application, a new operating system, or, increasingly, web browser version. In the case of lack of backward compatibility, this can occur because the programmers have only considered coding their programs for, or testing the software on, the latest version of an operating system. These version differences, whatever they might be, may have resulted in (unintended) software failures.

Input combinations and preconditions – A very fundamental problem with software testing is that testing under all combinations of inputs and preconditions (initial state) is not feasible, even with a simple product. This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing.

Static vs. dynamic testing – There are many approaches to software testing. Reviews, walkthroughs or inspections are considered as *static testing*, whereas actually executing programmed code with a given set of test cases is referred to as *dynamic testing*. The former can be omitted, whereas the latter takes place when programs begin to be used for the first time. This may actually begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions).

Software verification and validation – In software project management, software testing, and software engineering, *Verification* and *Validation* is the process of checking that a software system meets specifications and that it fulfils its intended purpose. It is normally part of the software testing process of a project.

Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

Verification is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

1.2 Verification

Programs often contain fatal errors despite the existence of careful designs. Many deadlocks and critical section violations, for example, are introduced

at a level of detail which designs typically do not deal with. This was for example demonstrated in the analysis of NASA's Remote Agent spacecraft control system written in the LISP programming language, and analyzed using the Spin model checker [2]. Here several classical multi-threading errors were found that were not really design errors, but rather programming mistakes such as forgetting to enclose code in critical sections.

One can argue that since modern programming languages are the result of decades of research, they are the result of good language design principles. Hence, they may be good design/modeling languages. This idea is already applied within UML where statechart transitions (between control states) can be annotated with code fragments in any favourite programming language. In fact, the distinction between design and program gets blurred since final code may get generated from the UML designs. An additional observation is that some program development methods suggest a prototyping approach where the system is incrementally constructed using a real programming language, rather than being derived from a pre-constructed design. Furthermore, any research result on programming languages can benefit design verification since designs are typically less complex.

It is well known that concurrent programs are non-trivial to construct, and with Java essentially giving the capability for anyone to write concurrent programs a model checker[1] for Java might have a bright future. In fact, one area where it can have an immediate impact is in environments where Java is taught. In the rest of this section some of the most important issues in the model checking of programming languages are mentioned. Specifically, the major reasons why model checking programs is considered hard are highlighted there.

Following sections highlight main topics of verification of Java program as described in [4].

1.2.1 Complexity of Language Constructs

Input languages for model checkers are often kept relatively simple to allow efficient processing during model checking. General programming languages, however, contain many new features almost never seen in model checking input languages, for example, classes, dynamic memory allocation, exceptions, floating point numbers, method calls, etc. Three solutions were pursued by different groups trying to model check Java: one can translate the new features to existing ones, one can create a model checker that can handle these new features, or, one can use a combination of translation and a new/extended model checker.

[1]Model checking, as defined in [3], is a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. if the design contains an error, model checking will produce a counterexample that can be used to pinpoint the source of the error.

Translation

The first version of JPF [5] was based on a translation from Java to Promela. Although both these systems were successful in model checking some interesting Java programs, such source-to-source translations suffer from one serious drawback – language coverage. Each language feature of the source language must have a corresponding feature in the destination language. This is not true of Java and Promela, since Promela, for example, does not support dynamic allocations.

Custom-made Model Checker

In order to overcome the language coverage problem it is obvious that either the current model checkers need to be extended, or a new custom-made model checker must be developed. Some work was done on extending the Spin model checker to handle dynamic memory allocation [6, 7], but again in terms of Java this only covers a part of the language and much more is required before full Java language coverage will be achieved this way. JPF took the other route, it has its own custom-made model checker that can execute all the bytecode instructions, and hence allow the whole of Java to be model checked. The model checker consists of its own Java Virtual Machine (JVM) that executes the bytecodes and a search component that guides the execution. Note that the model checker is therefore an explicit state model checker, similar to Spin, rather than a symbolic one based on Binary Decision Diagrams such as SMV [8]. A major design decision for JPF was to make it as modular and understandable to others as possible, but it sacrificed speed in the process.

Language and Properties Supported

The JVM implemented in JPF supports all Java bytecodes, hence any program written in pure Java can be analyzed. Unfortunately, not all Java programs consist of pure Java code - one often finds that certain methods are defined as being native to the operating system. When a Java program calls methods that have no corresponding bytecodes, then JPF cannot determine what the state of these code fragments will be and hence cannot handle programs that, for example, access the file system (user-defined class-loaders, file I/O operations, etc.), or communicate over a network, contains GUI code, etc. Fortunately, many native methods do not have side-effects and hence simple wrapper-methods can be written that translate the inputs and outputs to the native method, which then allow the original method to be called and all state changes to happen after returning from the call.

1.2.2 Complex States

In order to reduce a number of states during explicit state model checking one must know when a state is revisited. It is common for a hashtable to be used to store states, which means an efficient hash function is required as well as fast state comparison.

This insight also convinced the authors of JPF that it cannot tie model checking algorithm with an existing JVM, which is in general highly optimized for speed, but will not allow the memory to be encoded easily.

JPF design philosophy is to keep the states of the JVM in a complex data-structure, but one that would allow encoding the states in an efficient fashion in order to determine if the states have been visited before. Specifically, each state consists of three components: information for each thread in the Java program, the static variables (in classes) and the dynamic variables (in objects) in the system. The information for each thread consists of a stack of frames, one for each method called, whereas the static and dynamic information consists of information about the locks for the classes/objects and the fields in the classes/objects. Each of the components mentioned above is a Java data-structure. The solution to make the storing of states more efficient, is a generalization of the Collapse method from Spin [9]: each component of the JVM state is stored separately in a table, and the index at which the component is stored is then used to represent the component. The philosophy behind the collapsing scheme is that although many states can be visited by a program the underlying components of many of these states will be the same. This actually alludes to the other optimization: only update the part of the system that changes, i.e., keep the indexes calculated for the previous state the same, only calculate the one that changed.

1.2.3 Curbing the State Space Explosion

Maybe the most challenging part of model checking is reducing the size of the state space to be explored to something that tool can handle. Since designs often contain less detail than implementations, model checking is often thought of as a technique that is best applied to designs, rather than implementations. Applying model checking by itself to programs will not scale to large or complex programs. The avenue JPF authors were pursuing is to augment model checking with information gathered from other techniques in order to handle large programs. Specifically, they are investigating the use of symmetry reductions, abstract interpretation, static analysis and runtime analysis to allow more efficient model checking of Java programs.

Symmetry Reductions

The main idea behind symmetry reductions [10, 11, 12, 13] is that symmetries induce an equivalence relation on states of the system, and while performing analysis of the state space (for example during model checking) one can discard a state if an equivalent state has already been explored. Typically a canonicalization function is used to map each state into a unique representative of the equivalence class. Various schemes have been proposed for efficiently implementing such functions [12] and the complexity of this problem is discussed in [13].

Abstraction

The basic idea underlying abstraction algorithm is that the user specifies an abstraction function for certain parts of the data-domain of a system. The model checking system then, by using decision procedures, either automatically generates, on-the-fly during model checking, a state-graph over the abstract data [14, 15, 16] or automatically generates an abstract system, that manipulates the abstract data, which can then be model checked [17, 18]. The trade-off between the two techniques is that the generation of the state-graph can be more precise, but at the price of calling the decision procedures throughout the model checking process, whereas the generation of the abstract system requires the decision procedures to be called proportionally to the size of the program. Abstractions are often defined over small parts of the program, within one class or over a small group of classes, hence JPF favors the generation of abstract programs, rather than the on-the-fly generation of abstract state-graphs.

Static Analysis

Static analysis of programs consists of analyzing programs without executing them. In general, the analysis is performed without making assumptions about the inputs of the program. The analysis results are therefore valid for any set of inputs. A wide variety of techniques fall into the static analysis group; e.g., data flow analysis, set and constraint resolution, abstract interpretation, and theorem proving can all be applied to static analysis problems (with various degrees of success). They all derive some properties about a program. These properties are then used in slicing, code optimization, code parallelization, abstract debugging, code verification, code understanding, or code re-engineering for examples.

JPF interest in static analysis lies in its potential for reducing the size of the state space generated by a program. Therefore, JPF authors have focused their efforts on three static analysis problems that can result in state space reduction: static slicing, partial evaluation, and partial order reduction. Static slicing takes a program and a slicing criterion and generates a smaller program that is functionally equivalent to the original program with regard to the criterion. Partial evaluation propagates constant values and simplifies expressions in the process. Partial order reduction focuses on identifying statements that can be safely interleaved with any statement on a different thread. The combined use of these analyses results in smaller state spaces, and therefore, helps reduce the state explosion problem.

Runtime Analysis

Runtime analysis is conceptually based on the idea of executing a program once, and observing the generated execution trace to extract various kinds of information. This information can then be used to predict whether other different execution traces may violate some properties of interest (in addition of course to demonstrating whether the generated trace violates such properties).

The important observation here is that the generated execution trace itself does not have to violate these properties in order for their potential violation in other traces to be detected. Runtime analysis algorithms will typically not guarantee that errors are found since they work on a single arbitrary trace. They also may yield false positives in the sense that analysis results indicate warnings rather than hard error messages. What is attractive about such algorithms is, however, that they scale very well, and that they often catch the problems they are designed to catch. That is, the randomness in the choice of run does not seem to imply a similar randomness in the analysis results. In practice runtime analysis algorithms will not store the entire execution trace, but will maintain some selected information about the past, and either do analysis of this information on-the-fly, or after program termination.

Data race detection – A concrete data race occurs when two concurrent threads simultaneously access a shared variable and when at least one access is a write; hence the threads use no explicit mechanism to prevent the accesses from being simultaneous. The program is guaranteed data race free if for every variable there is a nonempty set of locks that all threads own when they access the variable. The Eraser algorithm [19] can detect that a data race on a variable is possible (potential) even though no concrete data races have occurred, by observing and remembering which locks are active whenever it is accessed.

Deadlock detection A classical deadlock situation can occur where two threads share two locks and attempt to take the locks in different order. An algorithm that detects such lock cycles must in addition take into account that a third lock may protect against a deadlock like the one above, if this lock is taken as the first thing by both threads, before any of the other two locks are taken. In this situation no warnings should be emitted. Such a protecting third lock is called a *gate lock*.

The algorithm for detecting this situation is based on the idea of recording the locking pattern for each thread during runtime as a lock tree, and then, when the program is terminated, comparing the trees for each pair of threads. The lock tree that is recorded for a thread represents the nested pattern in which locks are taken by the thread.

Using runtime analysis to guide model checking The runtime analysis algorithms described in the previous two list items can provide useful information to a programmer as stand alone tools. In this item, there is described how runtime analysis can be furthermore used to guide a model checker. The basic idea is to run the program in simulation mode first, using the JVM (implemented in JPF) simulator, with all the runtime analysis options turned on, thereby obtaining a set of warnings about data races and lock order conflicts. The threads causing the warnings are stored in a race window. When the simulation is terminated, forced or according to the program logic, the resulting race window will then be fed into the model checker, which will now search the state space, but

now only focusing its attention on the threads in the window. That is, the model checker only schedules threads that are in the window.

Chapter 2

Java PathFinder

Based on [20], Java PathFinder (JPF) is a system to verify executable Java bytecode programs. In its basic form, it is a Java Virtual Machine (JVM) that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. Unlike traditional debuggers, JPF reports the entire execution path that leads to a defect. JPF is especially well-suited for finding hard-to-test concurrency defects in multithreaded programs.

While software model checking in theory sounds like a safe and robust verification method, reality shows that it does not scale well. To make it practical, a model checker has to employ flexible heuristics and state abstractions. JPF is unique in terms of its configurability and extensibility, and hence is a good platform to explore new ways to improve scalability.

JPF is a pure Java application that can be run either as a standalone command line tool, or embedded into systems like development environments. It was mostly developed - and is still used - at the NASA Ames Research Center. Started in 1999 as a feasibility study for software model checking, JPF has found its way into academia and industry, and has even helped detect defects in real spacecraft.

The [20] describes the JPF as follows:

> *"The answer used to be simple: JPF is an explicit state software model checker for Java bytecode. Today, JPF is a swiss army knife for all sort of runtime based verification purposes."*

This basically means that JPF is a Java Virtual Machine that executes the program not just once (like a normal VM), but theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths. If it finds an error, JPF reports the whole execution that leads to it. Unlike a normal debugger, JPF keeps track of every step how it got to the defect.

2.1 JPF Capabilities

Out of the box, JPF can search for deadlocks and unhandled exceptions (e.g. `NullPointerExceptions` and `AssertionErrors`), but the user can provide own property classes, or write listener-extensions to implement other property checks (like race conditions).

In general, JPF is capable of checking every Java program that does not depend on unsupported native methods. The JPF JVM cannot execute platform specific, native code. This especially imposes a restriction as to what standard libraries can be used from within the application under test. While it is possible to write these library versions, especially by using the Model Java Interface (MJI) mechanism of JPF, there is currently no support for `java.awt`, `java.net`, and only limited support for `java.io`. Another restriction is given by JPF's state storage requirements, which effectively limits the size of checkable applications to approximately 10,000 lines of code (depending on their internal structure) if no application and property specific abstractions are used. Because of these library and size limitations, JPF so far has been mainly used for applications that are models, but require a full procedural programming language. JPF is especially useful to verify concurrent Java programs, due to its systematic exploration of scheduling sequences.

2.2 Model Checking vs. Testing

Unlike the testing, JPF can simulate non-determinism. Certain aspects like scheduling sequences cannot be controlled by a test driver, and require help from the execution environment (JVM). Other sources of non-determinism like random input data are supported with special APIs which can significantly ease the creation of test drivers. Simulating non-determinism requires more than just the systematic generation of all non-deterministic choices. Two capabilities come into play to make this work: backtracking and state matching.

Backtracking means that JPF can restore previous execution states, to see if there are unexplored choices left. For instance, if JPF reaches a program end state, it can walk backwards to find different possible scheduling sequences that have not been executed yet. While this theoretically can be achieved by re-executing the program from the beginning, backtracking is a much more efficient mechanism if state storage is optimized.

State Matching is another key mechanism to avoid unnecessary work. The execution state of a program mainly consists of heap and thread-stack snapshots. While JPF executes, it checks every new state if it already has seen an equal one, in which case there is no use to continue along the current execution path, and JPF can backtrack to the nearest non-explored non-deterministic choice.

In theory, explicit state model checking is a rigorous method - all choices are explored, if there is any defect, it will be found. Unfortunately, software model checking can only provide this rigor for reasonably small programs (usually less than 10,000 lines of code), since the number of states rapidly

exceeds computational limits for complex programs. This problem is known as state space explosion, and can be easily illustrated by the number of possible scheduling sequences for a given number of processes consisting of atomic sections.

JPF addresses this scalability problem in three ways:

Configurable search strategies try to solve the problem that the whole state space cannot be searched by directing the search so that defects are found quicker, i.e. with less computational resources. This basically means to use the model checker not as a "proofing", but as a "debugging" tool, which is mostly achieved by using heuristics to order and filter the set of potential follow-on states according to some property related relevance. Computation of heuristic values is delegated to a user configured class, i.e. is not hardcoded in the JPF core.

Reducing the number of states that have to be stored is the preferred way to improve scalability, and is supported by a number of mechanisms

- *Heuristic choice generator* means the set of choices in a certain state does not have to be complete. Consider a non-deterministic input float value with a threshold behavior. The float type makes it impossible to generate all possible values anyways, but in terms of checking the system behavior it might be sufficient to try only three choices: less than, equal, and greater than the threshold. The important capability is to make these heuristics configurable so that they can be easily extended or adapted to specific application needs.
- *Partial order reduction* is the most important mechanism to reduce the state space in concurrent programs. The goal is to only consider context switches at operations that can have effects across thread boundaries, like PUTFIELD instructions on objects that are accessible from different threads. The challenge is to do this on-the-fly, without requiring error-prone user instrumentation. JPFs partial order reduction makes use of the Java bytecodes, and reachability information obtained from the garbage collector, to achieve this
- *Host VM execution* - JPF is a JVM that is written in Java, i.e. it runs on top of a host VM. For components that are not property-relevant, it makes sense to delegate the execution from the state-tracked JPF into the non-state tracked host VM. The corresponding Model Java Interface (MJI) mechanism is especially suitable to handle IO simulaion and other standard library functionality.
- *State abstraction* - Per default, JPF stores all heap, stack and thread changes, which is sometimes a huge overhead if it comes to deciding whether two execution states differ from the perspective of a certain application. For example, state matching based on shape analysis of data structures can yield significant state reduction, and has been successfully used in recent JPF applications.

Reducing state storage costs refers mainly to implementation features of the JPF core. While not being the primary measure to deal with state space explosion, efficient state storage is mandatory for a software model checker. Since state transitions usually result in a small amount of changes (e.g. a single stack frame), JPF uses a technique called state collapsing to bring down the per-state memory requirements by storing indexes into state-component specific pools (hash tables) instead of directly storing changed values.

To compare states, JPF extends the state collapsing mechanism by hashing the resulting pool-index vectors, using a single, consecutive number as a unique state-id, thus reducing state equality checks to single integer comparisons. The hash mechanism (state set implementation instead of hash table) is configurable, using MD5 as default. The 128 bit hash values make it much more likely to run out of state memory before ever encountering a hash collision.

2.3 Java PathFinder Architecture

JPF was designed around two major abstractions: (1) the *JVM*, and (2) the *Search* object (Figure 2.1).

1. The *JVM* object is the Java specific state generator. By executing Java bytecode instructions, the JVM generates state representations that can be

   - checked for equality (has a state been visited before)
   - queried (thread states, data values etc.)
   - stored
   - restored

   The main JVM parameterizations are classes that implement the state management (matching, storing, backtracking). Most of the execution scheme is delegated to the `SystemState`, which in turn uses a `SchedulerFactory` (a factory object for `ThreadChoiceGenerators`) to generate scheduling sequences of interest.

   There are three major JVM methods in the context of the VM-Search collaboration:

   - *Forward* - generate the next state, report if the generated state has a successor. If yes, store on a backtrack stack for efficient restoration.
   - *Backtrack* - restore the last state on the backtrack stack
   - *RestoreState* - restore an arbitrary state (not necessarily on the backtrack stack)

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

   Search objects also configure and evaluate property objects (e.g. `NotDeadlockedProperty`, `NoAssertionsViolatedProperty`). The main Search implementations include a simple depth-first search (`DFSearch`), and a priority-queue based search that can be parameterized to do various search types based on selecting the most interesting state out of the collection of all successors of a given state (`HeuristicSearch`). A Search implementation mainly provides a single search method, which includes the main loop that iterates through the relevant state space until it has been completely explored, or the search found a property violation.

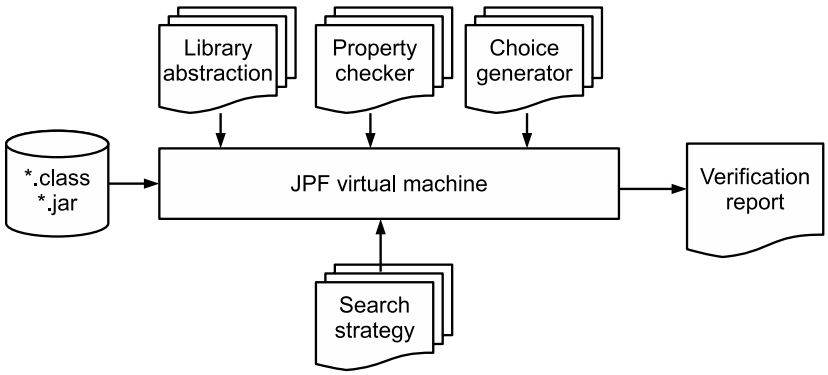


Figure 2.1: JPF Components

2.3.1 Verification Steps

For the standard depth first search (`gov.nasa.jpf.search.DFSearch`), listener implementations can assume the notification model in Figure 2.2.

2.3.2 Listeners

Beyond this basic Search-VM collaboration, there are numerous potential variations, e.g. to gather statistics, to monitor the state exploration progress, or to query details of states like field values. These are typical tasks for programs that use JPF, and add certain functionality on top of it (in this case it is a graphical user interface). The goal is to provide an extension mechanism in JPF that enables adding such functionality without modifying Search or VM implementations.

The required extensibility is achieved by means of a Listener pattern (a Observer variant with a wide, change-topic specific notification interface), i.e.

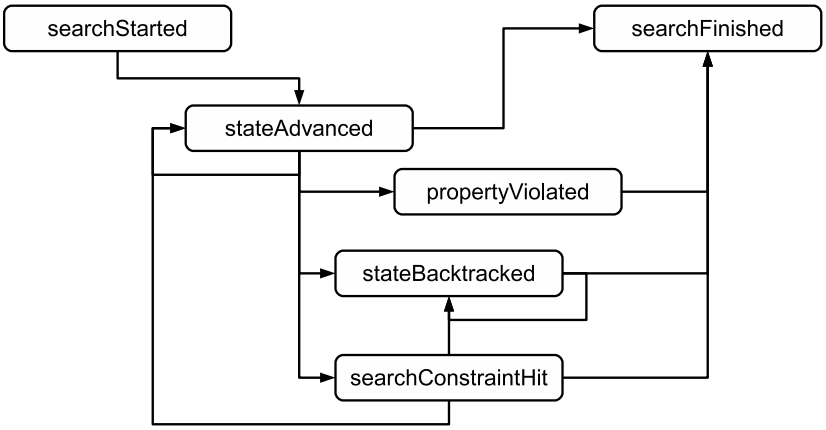


Figure 2.2: JPF Events

Listener instances register themselves with the Search and the VM object (Subject), get notified when their corresponding Subjects perform certain operations, and can then interact with the Subject to query additional information, or even control the successive Subject behavior.

Changed facets of the Subjects are mapped into separate Observer methods, passing in the corresponding Subject instance as a parameter.

2.3.3 Configuration and Inputs

Since JPF is an open system that can be parameterized and extended in a variety of ways, there is a strong need for a general configuration mechanism. The challenge for this mechanism is that many of the parts which are subject to parameterization are configured themselves (i.e. classes instantiated via classname parameters). This effectively prohibits the use of a configuration object that contains concrete fields to hold configuration data, since this class would be a central "design bottleneck" for a potentially open number of concrete JPF components like Search, Heuristic and Scheduler implementations. The goal is to have a configuration object that – (1) is based on symbolic values, (2) can be extended at will, and (3) is passed down in a hierarchical initialization process so that every component extracts only its own parameters.

JPF achieves this by means of a central dictionary object which is initialized through a hierarchical set of Java property files that target three different initialization layers:

1. *Site* – optionally installed JPF components
2. *Project* – settings for each installed JPF component
3. *Application* – the class and program properties JPF should check (this is part of your system under test)

4. *Command line* – command line parameters mostly used for debugging purposes

Initialization happens in a prioritized order, which means the user can override anything from later configuration stages, all the way up to command line parameters. Actually, this can be even overridden by using the explicit Verify API at runtime, which is a developer topic.

Besides properties configurations, the most important and mandatory input is program (bytecodes) whose model should be checked by JPF. All resources (e.g. class directories, jar files, etc.) needed by the program must be available and accessible on the classpath. Moreover, the main class of the program with main method has to be specified as an input parameter. This class is considered to be the root point from which all execution traces begin.

2.3.4 Outputs

There are three different ways a JPF run can produce output, each of them with a different purpose:

Application output – This is the most simple form of output, which usually just consists of `System.out.println(...)` calls embedded in the application code. There is only one caveat - since this is executed by JPF as part of the application, the same print statement might be executed several times. This can occur after the JPF backtracked to some previous state and tries another execution path. The ouput will be visible each time the JPF executes particular state where the log message is written to the application output. Since it can be sometimes confusing to see the same output twice without knowing if there is an iteration in the application, or JPF did backtrack between executions, there are two configuration options to control the output behavior – *vm.tree_output* and *vm.path_output*.

JPF logging – This is a more interesting form of JPF output, primarily intended to show what JPF does internally. For this purpose, it has to support various levels of details, ranging from severe errors to fine grained logging of JPF operations.

JPF's logging mechanism does not reinvent the wheel, it piggybacks on the standard `java.util.logging` infrastructure. While this means it would be possible to use customized `LogHandler` and `Formatter` (e.g. to log in XML format), there are specialized JPF incarnations of these classes, mainly to enable logging configuration via the standard JPF configuration mechanism rather than system properties.

Using the JPF Logging involves two aspects – controlling log output destination, and setting log levels. Both are done with JPF property files.

JPF reporting system – The JPF reporting system is used to show the outcome of a JPF run, to report property violations, print traces, show statistics and much more. This is in a way the most important part of the standard JPF user interface, and might involve various different output formats (text, XML, API calls) and targets (console, IDE). Depending on application and project, users might also need control over what items are displayed in which order. It is also obvious this needs to be an extensible mechanism, to adapt to new tools and properties. The JPF report system provides all this, again controlled by JPF's general configuration mechanism.

JPF Reporting

The basic concept is that reporting depends on a predefined set of output phases, each of them with a configured, ordered list of topics. The output phases supported by the current system are:

- *Start* – processed when JPF starts
- *Transition* – processed after each transition
- *Property violation* – processed when JPF finds a property violation
- *Finished* – processed when JPF terminates

The standard property violation topics include:

- *Error* – shows the type and details of the found property violation
- *Trace* – shows the program trace leading to this property violation
- *Snapshot* – lists each thread's status at the time of the violation
- *Output* – shows the program output for the trace (see above)
- *Statstics* – shows property statistics information

Last not least, the finished list of topics that usually summarizes the JPF run:

- *Result* – reports if property violations were found, and shows a short list of them
- *Statstics* – shows overall statistics information

Following section describes an default output of the JPF which checks the model of one of the sample programs used later in this thesis – TestDeadlock. Particular fragmets of the output are depicted with description and proposal of possible improvements.

```
JavaPathfinder v6.0 (rev 617+) - (C) RIACS/NASA Ames Research Center


========================================== system under test
application: TestDeadlock.java
```

Figure 2.3: The JPF output during start phase of execution

Start
Figure 2.3 shows messages which are written to the log right after a JPF run starts. The JPF version and the main class name of the program that is being checked is printed.

Error
Figure 2.4 depicts the JPF output in case of an error hit. In this particular case it is a deadlock and thus JPF lists all threads and their states. This part of output informs the user that a deadlock occured, the user can see the threads which are blocked, however, these information is not detailed enough to let the user know where and why the deadlock occured. More details can be found in the next output fragment.

```
========================================== error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
  thread id=1,name=Thread-1,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  thread id=2,name=Thread-2,status=BLOCKED,priority=5,
```

Figure 2.4: The JPF output during error phase of execution

Snapshot
Figure 2.5 shows the snapshot of the virtual machine state at the time of error hit. Again, there are threads listed, but this time there are more details added like owned locks, lock on which the thread was blocked and current call stack.

The important thing that would help the user to easily understand what happened and how the program got to the error is missing – variables and their values. The execution path of most programs is highly dependant on the values of variables (e.g. branches which are done based on a variable value, loop iteration index, etc.).

Result
Figure 2.6 depicts the output fragment with results overview. It is a list of errors that have been found. With the default configuration, JPF finishes after

```
========================================== snapshot #1
thread id=1,name=Thread-1,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  owned locks:TestDeadlock$SyncRunnable@141
  blocked on: TestDeadlock$SyncRunnable@142
  call stack:
        at TestDeadlock$SyncRunnable.doSomething(TestDeadlock.java:12)
        at TestDeadlock$SyncRunnable.run(TestDeadlock.java:25)

thread id=2,name=Thread-2,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  owned locks:TestDeadlock$SyncRunnable@142
  blocked on: TestDeadlock$SyncRunnable@141
  call stack:
        at TestDeadlock$SyncRunnable.doSomething(TestDeadlock.java:12)
```

Figure 2.5: The JPF output during snapshot phase of execution

first error is found. If the user changes the configuration to find all errors, JPF will list all errors in error, snapshot and result fragments of the output.

```
========================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:
```

Figure 2.6: The JPF output during result phase of execution

Statistics

In statistics fragment of the output (Figure 2.7) statistical details are listed. The important items are states, search, and max memory. The states and search rows give the user the overall picture how complex the state exploretion was. As mentioned in previous chapter, the memory allocation is important for storing states. The max memory allocation is important from the point how much more complex can the program be, to be able to check the model by JPF on current hardware environment.

```
========================================== statistics
elapsed time:       00:00:00
states:             new=8, visited=2, backtracked=2, end=1
search:             maxDepth=8, constraints hit=0
choice generators:  thread=8 (signal=0, lock=4, shared ref=0), data=0
heap:               new=347, released=11, max live=347, gc-cycles=9
instructions:       13412
max memory:         15MB
```

Figure 2.7: The JPF output during statistics phase of execution

Finished
Finished fragment (Figure 2.8) just informs the user when the actual JPF run finished.

```
========================================== search finished: 3/31/12 3:05 PM
```

Figure 2.8: The JPF output during finish phase of execution

The key points that should be improved by our approach are user-friendliness and availability of the list of variables for each thread and their stack frames.

The user-friendliness means to provide the user with a graphical user-interface (GUI) which will be integrated into the widely spread IDE. It is always beneficial to visualize information of such level of detail, so the user does not need to study the text information and realize all the neccessary relations and dependancies himself/herself.

As mentioned before, variable values are important for proper identification how and why the program executed along the trace which ended up in an error state. The proposed user interface will provide all variables which are available for each thread and its stack frame. This lets the user easily identify the error cause.

Chapter 3

Eclipse Debug Framework

The Eclipse Debug framework provides a powerfull API for launching and debugging program written in any programming language. Developers supporting new languages and environments can extend the framework for their own unique needs. The whole framework is packed into several Eclipse plugins. Next sections describe modules of the framework as were introduced in [21].

3.1 The Launch Framework

The launching in Eclipse means running or debugging a program within Eclipse. This action is performed by the launcher. It is a set of classes that live in an Eclipse plugin. The launcher can run Java application, JUnit test suites etc. In context of this thesis, it will be responsible for running a model checker (JPF) to check a Java program.

The launch framework includes facilities for spawning an OS process, persisting information about how something is launched, a framework for editing launch parameters (GUI), and an extension set of launch modes (run, debug, profile etc.).

The launch key items are:

Launch configuration types – Each launch configuration is of a specific type, e.g. Java application, JUnit test suite. All launch types are available from launch manager. The *launchConfigurationTypes* extension point allows new launch types to be contributed to the platform. Each launch type has domain specific attributes associated with it, which are stored in individual launch configurations.

Launch configurations – A launch configuration is a description of how and what to launch, in a form of persisted map of keys and values. Two types of launch configurations exist - *launchConfiguration* and *launchConfigurationWorkingCopy*. A launch configuration is read-only, while a working copy is used to edit a configuration in a transaction-like manor. A working copy is created from a launch configuration. Launch tabs modify a working copy and can commit changes to the original or revert. Launch configurations are designed to be shared across different

launch modes. The Debug platform defines three launch modes - run, debug and profile (this set can be extended via *launchModes* extension point).

Launch manager – Launch manager manages all configurations, configuration types and launches. A launch manager can be queried for all available configurations, registered configuration types or registered launch modes.

Launch delegates – A launch configuration type contributes a launch delegate for specific launch modes. The *launchDelegates* extension point allows a launch delegate to be contributed to an existing launch configuration type for a specific launch mode. The debug platform provides an abstract launch delegate that should be sub-classed. The abstract delegate (*LaunchConfigurationDelegate*) provides infrastructure to perform scoped builds before launching, allowing subclasses to specify the set of projects that should be compiled, scoped search for errors/problems, allowing the launch to be aborted and scoped search for unsaved editors, allowing the editors to be saved.

Launch objects – A launch object is a container of processes and debug targets created by launching. The debug platform provides a standard implementation of *IProcess* based on `java.lang.Process`. The debugger provides a console to display the standard I/O streams of a process. For each *IProcess* added to an *ILaunch*, the debug platform allocates a console attached to its I/O streams.

Tab group – A tab group is a set of tabs used to display and edit a single launch configuration. Tabs are displayed in the launch dialog when a configuration is selected. The *launchConfigurationTabGroups* extension point allows a tab group to contribute to an existing launch configuration type.

Launch shortcuts – A launch shortcut is added to the Run As... context menu. It provides a simple way for users to launch a file/program. When a shortcut is clicked, it creates a configuration (if one does not already exist) and launches it. It is contributed via the *launchShortcuts* extension point.

3.2 Debug Model

The debugger has to have some representation of the system, i.e., to debug with abstractions rather than bites and bytes, those abstractions have to be defined. The Eclipse Debug Model contains the basic abstractions of most imperative execution environments, e.g. process, thread, stack frame, variable, breakpoint, etc. The Eclipse Debug UI interacts with these abstractions and the different plugin debuggers provide the implementations.

The Debug Model key items are:

Debug model elements – The standard debug model contains debug target (`IDebugTarget`), threads (`IThread`), stack frames (`IStackFrame`), variables (`IVariable`), and register groups (`IRegisterGroup`). Variables, in this context, contain values (`IValue`), which can contain other variables to represent complex data structures.

Capabilities – The standard capabilities are Terminate (`ITerminate`), Step (`IStep`), Suspend and Resume (`ISuspendResume`), Drop to Frame (`IDropToFrame`), and Disconnect (`IDisconnect`). The standard debug elements implement standard capabilities -

```
IDebugTarget extends ITerminate, ISuspendResume, IDisconnect
IThread extends ITerminate, ISuspendResume, IStep
IStackFrame extends ITerminate, ISuspendResume, IStep
```

The debug toolbar buttons/actions operate against these interfaces.

Debug events – A debug event describes something that has happened in a program being debugged or in a running process. An event has a type (kind) and detail code. The user interface requires debug model elements and process implementations to generate the debug events. Required events are specified in the `DebugEvent` class. Detail codes describe why an event occurred (e.g. suspend could be caused by step end, breakpoint, client request, evaluation, or evaluation implicitly).

Debug views – The debug views display debug elements, which are updated in response to debug events. The standard views are Debug, Variables, Registers, Expressions, Breakpoints, and Console. When a view refreshes it appempts to maintain selection and expansion state based on element equality. If element remain stable, so will the view.

Debug actions – The actions operate on the debug elements. The actions update in response to selection change and debug events. The standard debug actions are step, suspend, resume, terminate, disconnect, and drop to frame.

Debug model presentation – Debug model elements are displayed with text and images (standard images are provided by the platform). Default elements are just element names. To provide custom labels and images it is needed to contribute a *debugModelPresentation* extension point.

3.3 Breakpoints

Breakpoints are a way to suspend execution at specific location or upon a specified condition. There exist several types of breakpoint, i.e. line breakpoints, watchpoints, run-to-line, and exception traps. The breakpoint framework provides facilities for add, remove and change notification, persistance of breakpoints across workbench invocations and temporarily skipping breakpoints.The

types of breakpoints a debugger provides depends on the capabilities provided by the underlying debug architecture and aggregate functions that can be built with those capabilitites.

The breakpoints key items are:

Breakpoint extension – The platform provides an *breakpoints* extension point for contributing kinds of breakpoints.

Breakpoint – A model object which is representing a breakpoint implements `IBreakpoint`. The platform also defines base interface for `IWatchpoint` and `ILineBreakpoint`. A breakpoint contains the information required to install itself into a debug target. All implementations must have a default constructor such that the platform can instantiate persisted breakpoint on workspace startup. The platform's implementation provides enablement and attribute persistance. Breakpoint behaviour is provided by client implementations, i.e. custom properties, installation, and adhering to enablement.

Marker – Breakpoint attributes are stored in markers. *IMarker*s are provided by the platform as general markers in files (bookmarks, compilation errors, etc.), that can be displayed in an editor ruler. An marker is just a store of key/value pairs of primitive data types. All breakpoints have an associated marker to persist its attributes and display in an editor.

Breakpoint manager – The breakpoint manager (`IBreakpoitnManager`) is a repository of breakpoints in the workspace. The manager is responsible for breakpoint registration/removal, if the breakpoint is created/deleted. It provides change notification as breakpoints are added, removed, and when a breakpoint attribute changes. Clients interested in breakpoints implement `IBreakpointsListener` and register with the manager for change notification.

Debug target – The debug target installs breakpoints. When a debug target is created, it should query the breakpoint manager for all existing relevant breakpoints and install them (deffered breakpoints). It listens for breakpoints being added/removed/changed during its lifecycle, and updates them in the underlying runtime.

Retargettable actions – Most debuggers support a common set of breakpoint types - line breakpoints, method breakpoints, and watchpoints. Global actions are provided for creating these kind of breakpoints. They promotes a common look and feel across debuggers and avoids polluting menus with similarly named actions.

Editor – The editor visualizes the location of breakpoint/watchpoints. It displays markers in vertical ruler and updates as markers are changed. Editors that subclass `AbstractDecoratedTextEditor` have a ruler to display markers associated with the file (resource) they are editing.

Ruler double click – Double click in ruler can have more than one action depending on context. It can set a line breakpoint (line or method entry) or set a watchpoint.

Breakpoint properties dialog – Breakpoints have editable properties, e.g. hit count, suspend policy, enablement, condition.

3.4 Source Lookup Framework

Highlighting the current source code line or statement is standard and expected feature in modern debuggers. The debugger has to find source code for a binary location and display that source in the editor. Typically, finding source code means looking for a particular file along a path of directories, zip archives, Java archives, etc.

The source lookup key items are:

Source locator – An implementation of a standard "search along a path" type of source locator, which consists of director, participant and container. A participant maps stack frames to filenames and a container finds files by filename in directories, zips, jars, etc.

Source director – A director holds the ordered list that is the "path". The default implementation of the source lookup director provides an implementation of a "path" as a consistently ordered sequence of containers. If the user has not specified an explicit source path, the director computes a default source path (set of source containers), based on launch configuration type.

Source participant – A participant maps stack frames to filenames. Source lookup director usually has one participant.

Source container – The platform provides implementations of standard source containers, i.e. workspace folders, projects, archives and local file system directories and archives.

3.5 Variables

The Variables view provides facilities for emphasizing variables that change value, displaying "logical structures" vs. raw implementation structures and displaying "details" for a selected variable. A logical structure is an alternate presentation of a variable's value. Often, it is more natural to navigate a complex data structure via an alternate semantic presentation of the value, rather than its implementation. For example, no matter how a list is implemented (linked, array, etc.), the user wants to see the elements in the list in terms of an ordered collection.

Chapter 4

Application Design

This chapter describes the approach which improves the user interface (UI) of JPF. As described in the previous chapter, UI of JPF is far from user-friendly and easy to use. It requires a deep knowledge of Java threads, Java scheduling and assertions, but it still takes a while to realize where the problem is. This thesis proposes new debugging interface to JPF. By utilizing Eclipse Debug framework in Eclipse Integrated Development Environment (IDE), JPF can be used to identify critical faults of the tested software and it can be easily debugged via standard debugging interface.

The main idea is to detect potential problems with JPF and afterwards stop the application execution at the point which JPF identified as problematic. This gives the user clear view where the problem is and what was the current state of the application when the problem occurred. All this information is clearly displayed in Eclipse via standard Java application debugger interface.

The original proposal to use JPDA as a debugging framework was changed to Eclipse Debug Framework. The main reason for this decision was the fact that JPDA is fully-featured "heavy" debugging framework for Java environment and the complexity of implementing all of its interfaces would add features that are not goals of this thesis.

Such advantages would be possibility to use this solution with other JPDA compliant debuggers (however the compatibility is not guaranteed as there might be some implementation differences and JPF is not JVM for which JPDA was designed). Another advantage would be remote debugging which is currently not available in this solution.

Disadvantages of using JPDA would be complexity and cumbersomeness of design and implementation. JPDA uses low level byte stream communication. It supports many features that are not applicable and/or not available in JPF environment and the implementation would be inadequately difficult or even not possible (e.g. variable value change in runtime, watches).

On the other hand, Eclipse IDE is widespread across Java developers and the solution is simple, readable and fulfilling all goals of this thesis.

Advantages of using Eclipse Debug Framework are:

- Simplicity and readability – Implementing the debug model and calling well-defined Java API make the solution easy to implement, read and

understand without any workarounds.

- Possibility to propose cummunication protocol that fits best for JPF and Eclipse integration.
- Full compatibility with Eclipse – the most common Java development environment (based on research report [22] - Eclipse has 65% of market share).

Drawbacks of using Eclipse Debug Framework are:

- Usage of ad-hoc communication protocol
- Tight-coupling with Eclipse development environment

4.1 Application Components

The overall solution components and their interactions are depicted in Figure 4.1. It consists of six main components (implementations of three of them, gray ones, are parts of this thesis):

- Eclipse (Eclipse Debug framework)
- JPFDeb.core
- debug4jpf
- Listener
- JPF
- JPF JVM

Eclipse, JPF and its JVM components are described in previous chapters as they are already existing parts of the solution. Following sections explain the role of the remaining three components and their mutual communication with each other as well as communication with the former three.

4.2 The debug4jpf Component

JPF and debug4jpf are in a role of debuggee. The debug4jpf component is the front-man of JPF. It initiates JPF, controls its execution and provides requested details from JPF to the debugger.

The debug4jpf component uses its own JPF listener to control the execution of JPF. Every event generated by JPF is forwarded to the listener. These events let debug4jpf follow the progress of the JPF execution.

VM listener's events always receive the `JPF` object instance as a parameter. Therefore, debug4jpf has complete information about the actual execution progress, memory content, etc. At the same time, the search listener's

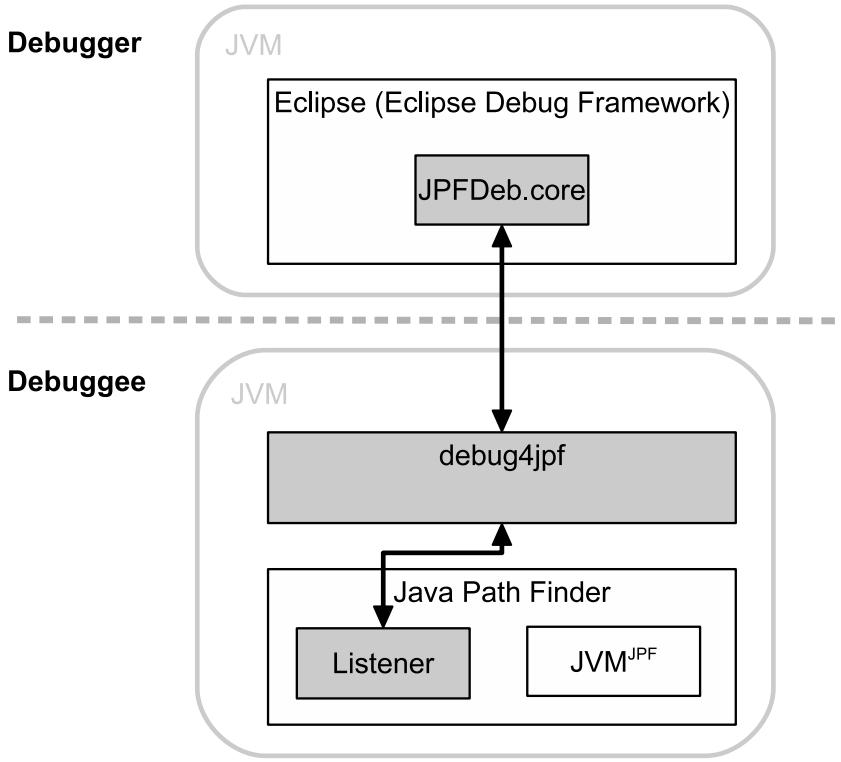


Figure 4.1: Solution architecture

events receive `Search` object as a parameter. This object provides complete information about the actual step of the state space exploration.

The debug4jpf component runs JPF twice. The reason for this are breakpoints. As described in the previous chapters, JPF traces "all" possible traces of execution. In case there is a breakpoint at a particular point in the tested program, JPF would stop execution each time it would get to the breakpoint. This would not be acceptable for users as it can occur several hundreds or even thousands times per one test run. The proposed and implemented solution is running JPF twice. First run (further referenced as exploration run) finds a trace to the error and stores it into a temporary file. The second one (further refenrenced as debug run) uses the stored trace to stear the exploration directly to the error state. First run runs transparently to Eclipse and the user. The second one is controled by the user-defined breakpoints and if error is found, the user is notified.

First run uses its own listener, further referenced as error path listener. This listener handles only one event - propertyViolated. This event handler generates temporary file which contains program trace in JPF format (basically it contains instructions for the choice generator). This file is generated only if an error is found.

Second run uses different listener, further referenced as debugging listener, and the file with the error path generated by the first run. To be more specific, in case the first run does not find any error, the file does not exist. This has

the advantage, that the second run is not executed and the user is notified that there was no error found.

The debug4jpf component is started by Eclipse debgging plugin. As soon as the component is loaded, `started` event is fired (sent to Eclipse) and the processing is started. There are three ways how to finish debug4jpf - send exit command from Eclipse, process through both JPF runs to the end or in case there is no error found, debug4jpf finishes after the first run of JPF. In each case, the component fires `exited` event.

The exploration run of JPF is running at full JPF speed to get to the point of error to be able to store the error path as quickly as possible. Full speed means that there is no additional logic in error path listener that would slow down the proccessing. The debug run of JPF is impacted by the performance of debugging listener. Its logic is responsible for breakpoints handling. This means that after each executed instruction it has to be checked if there is a breakpoint or not.

Both JPF runs are started an handled transparently to the user. It is presented to the user as standard Java program debugger interface.

The main difference between debugging standard Java program and debugging Java program through debug4jpf is stepping. In case of error. Eclipse (or any other IDE) users are used to step through Java programs line by line to check the program execution before the error occurs. In order to understand reasons why this is not supported by debug4jpf, it is neccessary to realize that at the time of debug run of JPF, thread scheduler has been already preconfigured by exploration run of JPF. This means that stepping one line forward (or stepping into method invocation) in one thread could not be done without impacting other threads. It would cause confusion to the user as this is not standard way how stepping should behave. The user always controls only currently selected thread during standard Java program debugging. In JPF environment, this is not possible. An example is described in Figure 4.2. Executing one step in Thread 1 includes also three steps in Thread 2, terminating Thread 3 and starting new Thread 4.

Breakpoints are introduced to minimize impact on user comfort. They allow user to point to a line in source code and whenever the debug run of JPF reaches this line, it suspends its execution (all threads are suspended). At this time all details of all live threads are available - threads, stack frames and variables. JPF processing continues after receiving `resume` command from the debuger.

Overal view on debug4jpf execution is depicted in Figure 4.3. To summarize, the listeners in debug4jpf handle the following events:

Property violated (JPF encountered a property violation) - it is handled by both listeners. The debugging listener handles it as execution suspended and Eclipse is notified. The error path listener handles it as an error path is created and stored in a temporary file.

Execute instruction (JVM is about to execute the next instruction) - it is handled by the debugging listener. Before each instruction is executed, it

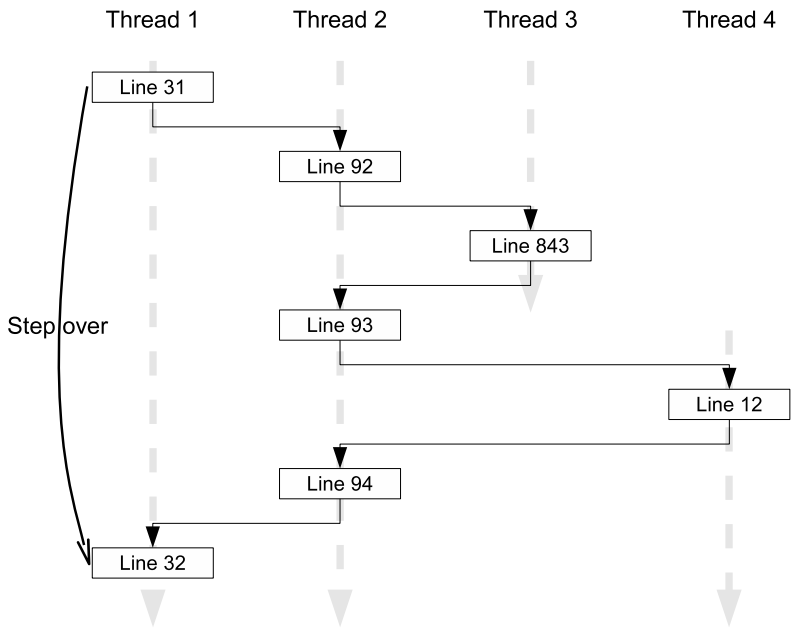


Figure 4.2: Thread scheduling example

is checked whether there is a breakpoint enabled for this source file and line number. If there is a breakpoint, execution is suspended and Eclipse is notified.

4.2.1 Runtime Status Holder

JPF uses `Runtime` object to inform debug4jpf about its state and debug4jpf uses this object to request JPF state change. Possible states are:

Not started - debug4jpf has already started and is somewhere in the middle of initialization or execution of the first run of JPF searching for an error trace.

Running - debug4jpf has started, first run of JPF is finished and the trace was written to a file. Second run of JPF has already started and it is running.

Set suspended - Eclipse requested debug4jpf to suspend and debug4jpf is waiting for JPF to finish instruction execution to be suspended.

Suspended - there are three possible reasons for this state - Eclipse requested debug4jpf to suspend, or JPF encountered a breakpoint, or JPF encountered an error in the program under analysis.

Set resumed - Eclipse requested debug4jpf to resume. After JPF is resumed the state will be changed to running.

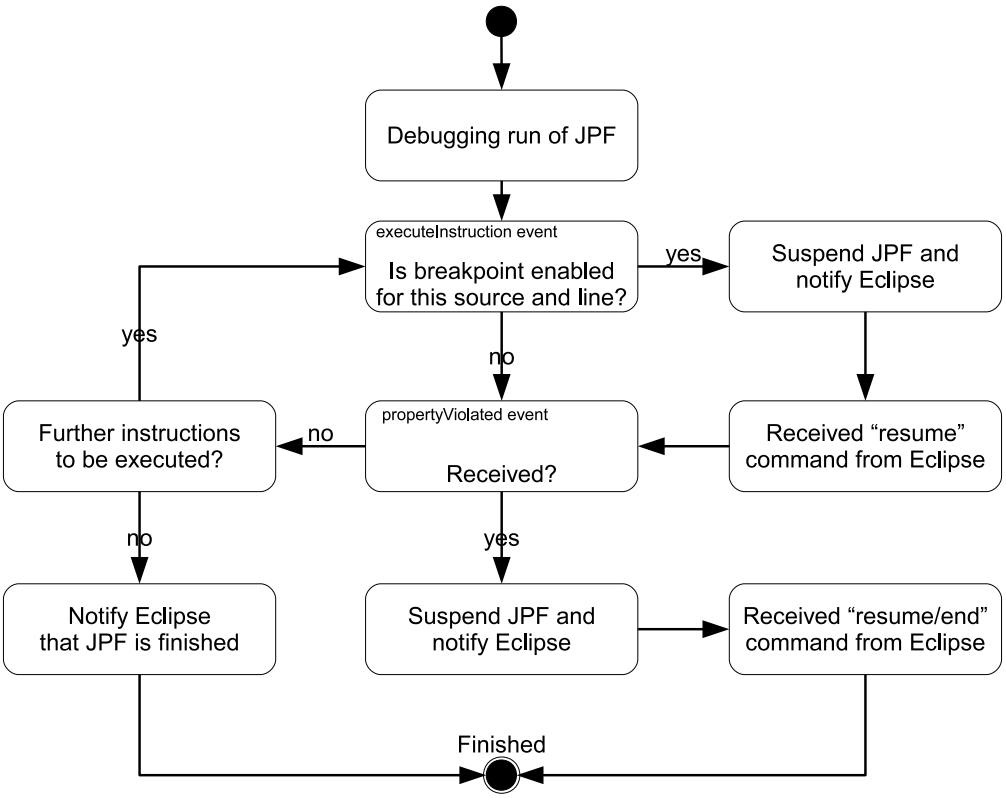


Figure 4.3: Debug run state diagram

Set finished - Eclipse requested debug4jpf to finish and debug4jpf is waiting for JPF to finish instruction execution to be finished.

Finished Both runs of JPF have already finished, debug4jpf is waiting to be exited.

State change diagram is depicted in Figure 4.4. Please note, that not all state change transitions are shown. The first reason is that the picture would be hard to read and the second is that these missed transitions should never be used while using this tool in the standard way. As an example, there should be a transition between states `Set suspended` and `Set finished`. However in the actual implementation, the state `Set suspended` is changed to `Suspended` right after current instruction is executed by JPF.

In order to satisfy Eclipse debugging interface and provide all details that user is used to while debugging Java applications, debug4jpf has to provide runtime details about threads, stack frames, variable names and their values. All these details are parts of JPF data model. Main root object is `JPF`. This object holds `Config`, `Search` and `VM` objects. The `Config` object contains configuration details. Initialization happens in a prioritized order, which means anything can be overriden from later configuration stages, all the way up to command line parameters.

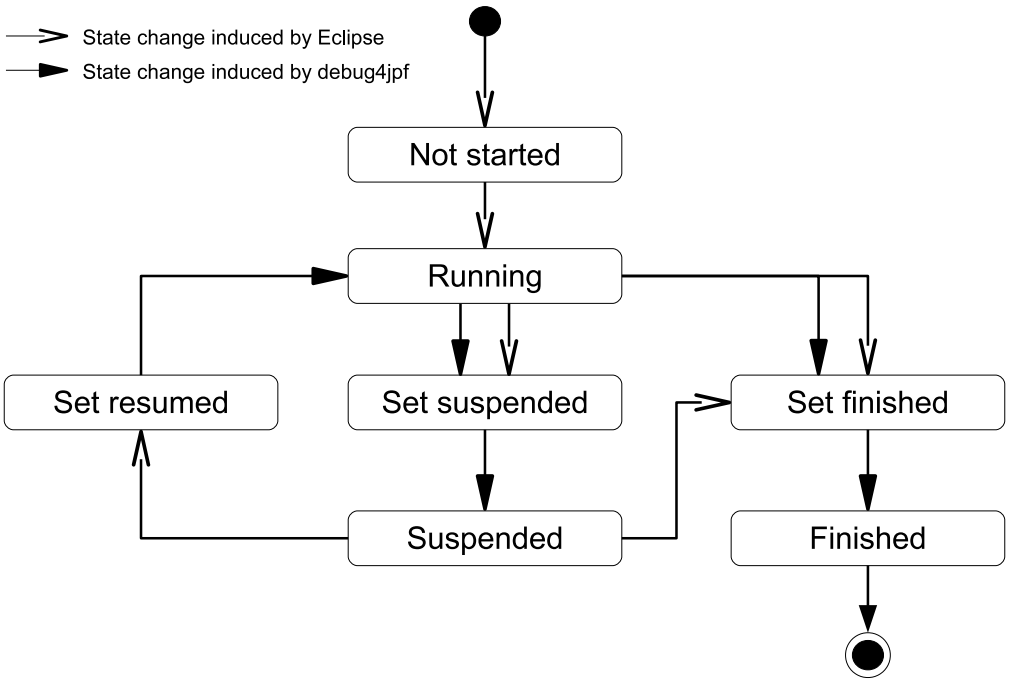


Figure 4.4: debug4jpf runtime state diagram

4.2.2 Data Model

The `Search` object provides details of an algoritm which JPF uses for tracing through program that is being tested. An example of such detail is an execution path from the very begining to the current state. This detail is used in exploration run of JPF to get the exact path to the error state.

The `VM` object holds all runtime information that is related to virtual machine execution. Virtual machine in this context refers to virtual machine within JPF which executes instructions of a program that is being tested (not the virtual machine that executes JPF itself). The virtual machine object provides internal rather technical details of the execution. Objects which are referenced from `VM` can be divided into three categories - bytecode execution, object model and type + code management.

The bytecode execution group contains `SystemState` and `ThreadInfo` objects. The latter one is used in debug4jpf to get details of threads such as count of threads, thread state (e.g. running, blocked, terminated) and stack frames of the current thread. The `StackFrame` object is crutial for retrieving information about stack traces and variables names.

The stack frame and the variable name is used for getting a variable value. Objects from the object model group keep all object instances - fields, local variables and operands. These values are stored for each stack frame, therefore it is possible to reconstruct system state in each stack frame.

If there is a variable of non-primitive type, Eclipse have to know also the structure of this type - names and types of class fields. The type + code

management group of objects provides such details. `ClassInfo`, `MethodInfo` and `FieldInfo` hold class, method and field structures. For classes it is name, fields names and types, class methods etc.; for methods it is name, parameters, return value etc.; for fields it is name and type.

Figure 4.5 shows overal picture of JPF data model and relationships of particular entities. Gray entities are used in our approach.

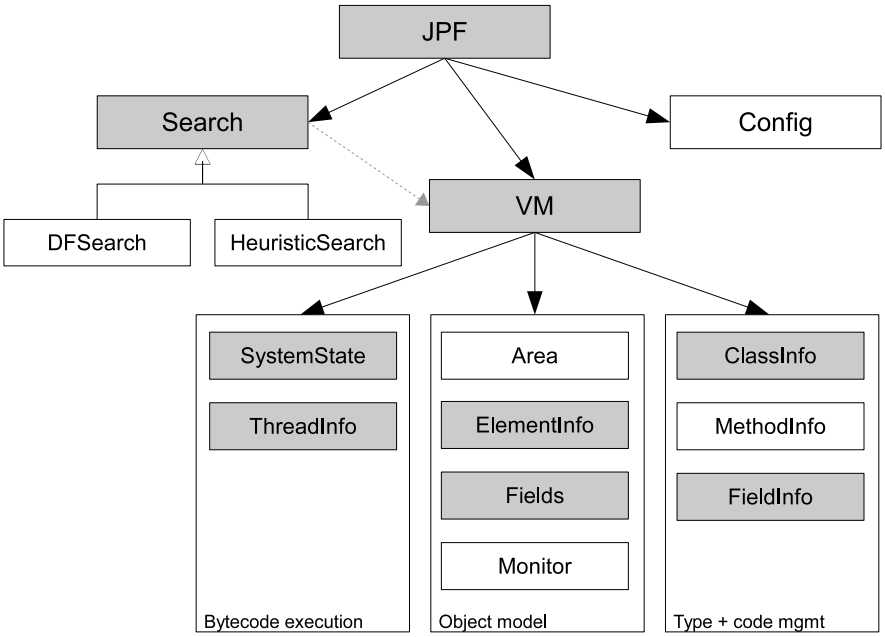


Figure 4.5: JPF data model

4.2.3 Commands and Events Handled by debug4jpf

The debug4jpf component is able to handle several commands. Generally, a source of these commands can be any external component which implements communication protocol used by debug4jpf. In our approach, the external component is Eclipse with JPFDeb.core plugin installed. Commands are divided into three categories - runtime commands, variable commands and breakpoint commands.

Runtime commands are responsible for controling debug4jpf processing. They start, suspend, resume and finish the processing of JPF and debug4jpf itself. Table 4.1 lists the supported runtime commands. The response to the runtime command is just notification that the command was accepted and it was queued to be processed.

Variable commands are responsible for threads, stack frames and variables retrieving. The sequence of commands starts with getting the list of all threads (running, blocked, terminated). Next step is getting stack frames (variable names) for all threads. This enables Eclipse to show thread list with links to the source code, i.e. their current positions defined by file name and line

Command	Runtime status change	Result
start	`Running`	Status message
suspend	`Set suspended`	Status message
resume	`Set resumed`	Status message
exit	`Set finished`	Status message

Table 4.1: List of the runtime commands supported by debug4jpf

number. The final step is to get values for all variables of the selected thread and stack frame. Table 4.2 lists all the supported variable commands.

Command	Result
thread	List of the threads and their ids, names and states.
stack	List of the stack frames for the thread and their source file names, line numbers, method names and variable names.
var	Value for the variable in the stack frame of the thread. In case of object it returns also structure of the object.

Table 4.2: List of the variable commands supported by debug4jpf

Breakpoint commands responsibility is to add and remove breakpoints. The debug4jpf component holds the list of all enabled breakpoints at runtime. Two commands in this category modify this list. Table 4.3 lists all the supported breakpoint commands.

Command	Description	Result
set	Adds the breakpoint to the list of enabled breakpoints.	Status message
clear	Removes the breakpoint from the list of enabled breakpoints.	Status message

Table 4.3: List of the breakpoint commands supported by debug4jpf

In addition to all commands, debug4jpf sends events to the source of commands - the debuger. This events notifies the debuger that the requested change was successfully processed. It happens that the event is sent to the debuger even without previous command. This can occur in situations when processing hits a breakpoint (`suspended` event is sent), JPF gets to an error (`suspended` event is sent) or JPF gets to the end (`terminated` event is sent). Table 4.4 lists all supported events.

4.3 The JPFDeb.core Component

The user interface is implemented as a plugin to Eclipse. This enables easy integration and fully featured Java development and debugging environment. Plugins in Eclipse are used to extend the standard skeleton of Eclipse. Eclipse itself is the user interface application with the built-in plugin engine. All Eclipse

Event	Description	Reason
started	The exploration run of JPF started	Response to the `start` command
suspended	The debug run of JPF suspended	Response to the `suspend` command or when breakpoint or error occurs
resumed	The debug run of JPF resumed	Response to the `resume` command
terminated	JPF run terminated	Response to the `exit` command or JPF gets to the end

Table 4.4: List of the events supported by debug4jpf

features, e.g. Java editor, Java compiler, perspectives, views etc., are parts of plugins which come in the standard Eclipse distribution for Java developers.

The plugin, called JPFDeb.core, is used as the interface between user and debug4jpf. This plugin is started during Eclipse inicialization. There are several items added to the Eclipse menu and Run/Debug dialog window (Figure 4.6). Each time the user starts the program analysis by JPF, new instance of Eclipse process and debug target are loaded. The Eclipse process is a container for running the debug4jpf component. It starts new JVM instance and executes debug4jpf in it. The debug target is the main execution class of the plugin and the root element of Eclipse Debug framework data model (more details in Section 4.3.1). After JPF completes state exploration and debug4jpf is finished, the process and debug target objects are destructed.

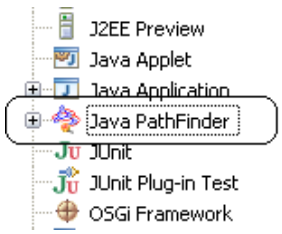


Figure 4.6: JPFDeb.core menu items and Run/Debug dialog

The JPFDeb.core plugin contains several extensions, which are the basic building blocks for Eclipse plugins. The `LaunchConfigurationTypes` extension configures available modes - run and debug. Run mode enables the user to analyze program with JPF and print out the JPF output. Debug mode enables user to fully use all features of JPFDeb.core. `LaunchConfigurationDelegate` which is also configured in this extension is intantiated by executing run/debug launch configuration. The delagate parses the launch configuration, starts the process and the debug target.

The `SourceLocator` and the `SourcePathComputer` extensions are responsible for locating a source code file name and line number. If a breakpoint is hit, these extensions find the file, open a text editor (the editor type is de-

pendent on the file type) and set the cursor to the line where the processing stopped.

The `Breakpoints` and the `Markers` extensions handle breakpoints. The `LineBreakpoints` are implemented. Breakpoints, which are inserted into/removed from source code, are communicated to the debug4jpf component.

4.3.1 Data Model

The JPFDeb.core component implements Eclipse Debug framework data model (overal picture in Figure 4.7). These classes are interacting with Eclipse via the debug target class. The debug target class instance is registered in Eclipse by `LaunchConfigurationDelagate`.

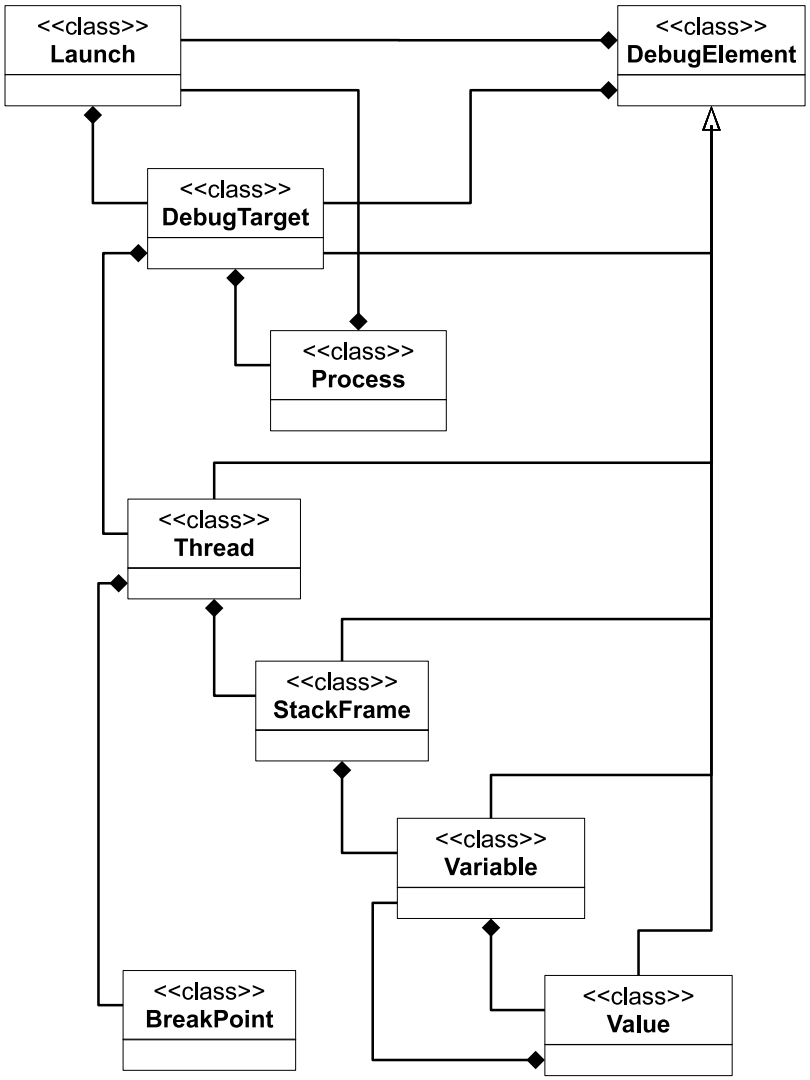


Figure 4.7: Eclipse Debug framework data model implementation

All other classes either directly or indirectly reference the Debug target

UI element (marked as 1 in Figure 4.8). The Thread UI element (marked as 2 in Figure 4.8) instances are representing all threads currently existing in program exploration in JPF. Each thread has a list of Stack frames (marked as 3 in Figure 4.8). If a stack frame is selected, all variables are listed in Variable view. Every variable is represented by Variable UI element (marked as 4 in Figure 4.8) and its value is represented by the Value UI element.

The Breakpoint UI element (marked as 5 in Figure 4.8) instances represent the user defined breakpoints. Breakpoints are marked with a standard blue sphere and in case of breakpoint hit, the blue arrow is shown next to the line.

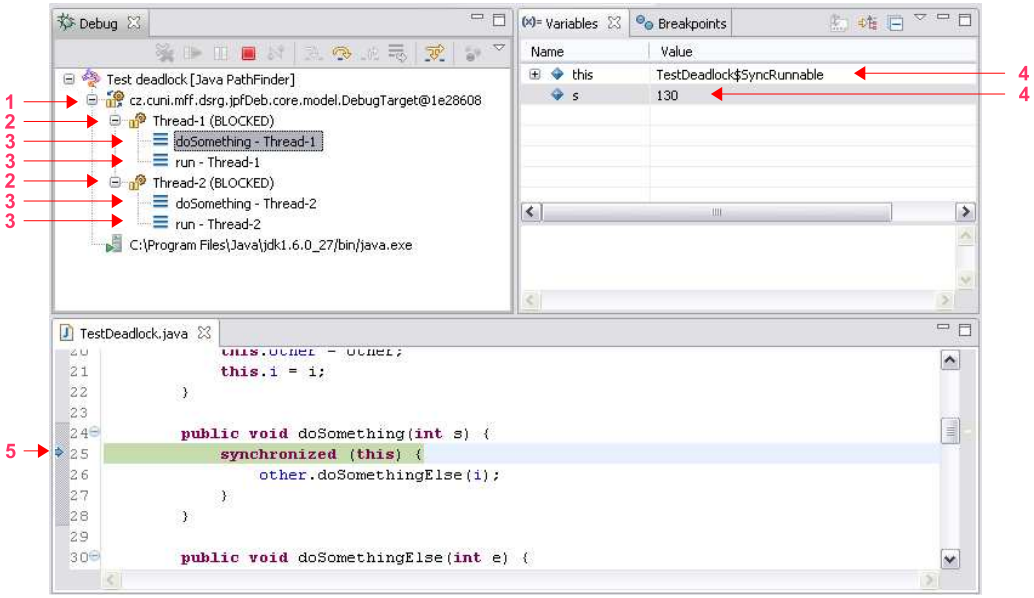


Figure 4.8: JPFDeb.core user interface components

4.4 Communication Protocol

Communication between debug4jpf and JPFDeb.core components is using an ad-hoc socket-based transfer protocol. There are two data flows defined - one for commands and another for the events flow. Each of these flows has one separate socket. The socket for commands is transfering data both ways - from the debugger to the debuggee (commands) and from the debugee to the debugger (responses). The socket for events is one-way only - from the debuggee to the debugger (events). Both sockets work with character strings as commands, responses and events.

Different situations require different communication types. All types of communication are described in Figure 4.9. Breakpoint handling commands are implemented in the request-response style (1). A command from the debugger is sent to the debugge and a confirmation response is returned back. Variables handling commands use also the request-response style but, unlike in the previous case, the response contains a payload (2), e.g. list of threads or stack

frames. The third and most complicated communication style, which is used by the runtime handling commands, is using the request-response plus the notification pattern (3). As soon as a command is parsed, a response is sent synchroniously back. After the command is handled properly, a notification (event) is sent from the debuggee to the debugger. The last communication style sends a simple event acknowledgement (4). This occurs in case debug4jpf changes its state based on an internal condition like a breakpoint hit or an error hit (not based on an external command). Regarding race conditions, the synchronized access to the sockets and the `Runtime` object makes sure that all the commands and events are processed in an expected order.

Error handling is based on various status codes in responses. If a command is processed successfully, the "OK" status code is returned. Otherwise, a particular error code describing what went wrong is returned.

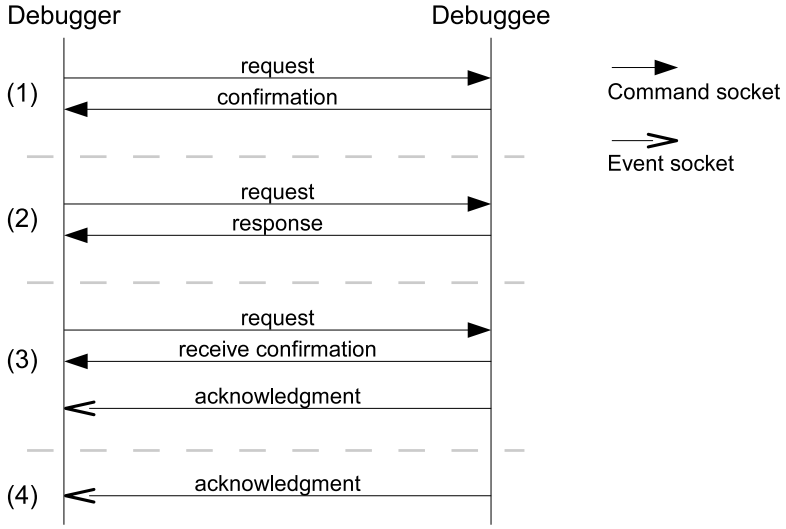


Figure 4.9: Communication styles

List of all supported commands and events:

`start`

- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `started` event

`suspend`

- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `suspended` event

`resume`

- this command does not have any parameter

[1]Status message format is `status <status>`, where <status> can be either "OK" or any string describing the status.

- the receive confirmation is status message[1]
- the acknowledgement is `resumed` event

`exit`
- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `terminated` event

`thread`
- this command does not have any parameter
- the response format is
  `<thread1Id>|<thread1Name>|<thread1Status>#<thread2Id>...`

`stack <threadId>`
- this command has one parameter - thread id
- the response format is
  `<sourceFile1>|<lineNumber1>|<methodName1>|<varName1_1>|`
  `<varName1_2>|...#<sourceFile2>...`

`var <threadId> <stackFrameId> <varName>`
- this command has three parameters - thread id, stack frame id, variable name
- the response format is
  `<varName>=<value>` for primitive types
  `<varName>=<varType>(<varName>.<varChildName1>,...)` for non-primitive types

`set <sourceFileName> <lineNumber>`
- this command has two parameters - source file name and line number where the breakpoint is located
- the confirmation is status message[1]

`clear <sourceFileName> <lineNumber>`
- this command has two parameters - source file name and line number where the breakpoint is located
- the confirmation is status message[1]

Chapter 5

Implementation

The previous chapter elaborates on the design of the approach. This chapter lists technical details. The first implementation decision was programming language and development environment. The JPF API as well as Eclipse plugins are implemented in Java, so Java was chosen as the programming language for debug4jpf and JPFDeb.core plugin. Programs that are tested by JPF are Java programs too. In order to simplify the frontend part implementation in form of Eclipse plugin, Eclipse was chosen as the development environment for JPFDeb.core and debug4jpf. It supports many usefull utilities and plugins that help with Eclipse plugins implementation.

Besides standard Java and Eclipse libraries, JPF is used as an external library. There were two options to link JPF to debug4jpf component

(i) include JPF in debug4jpf library

(ii) let user to link local JPF installation to the debug4jpf.

The first option was dismissed because it would not be able to reuse the JPF installation for any other purposes. Another reason is that debug4jpf is build as superior/control component on top of JPF and not as an JPF extension. The second option to let user link any local JPF instance turned out to be problematic. The development and issue fixing activities on JPF are not backward compatible. JPF data model API changed rapidly during debug4jpf implementation phase and the whole component stopped working after JPF had been updated. The final version of debug4jpf component was adapted to be fully compatible with the latest version of JPF[1]. This limits the versions of JPF, which are guaranteed to be fully supported by debug4jpf, to one. Following versions of JPF are not guaranteed to be fully compatible with debug4jpf.

5.1 Implementation Structure

The two components listed in Chapter 4 – debug4jpf and JPFDeb.core – are implemented as two separate Java projects. The JPFDeb.core project is

[1] At the time of working out the master thesis - revision 617+.

dependant on the debug4jpf one, as it instantiates debug4jpf. The dependencies are shown in Figure 5.1.

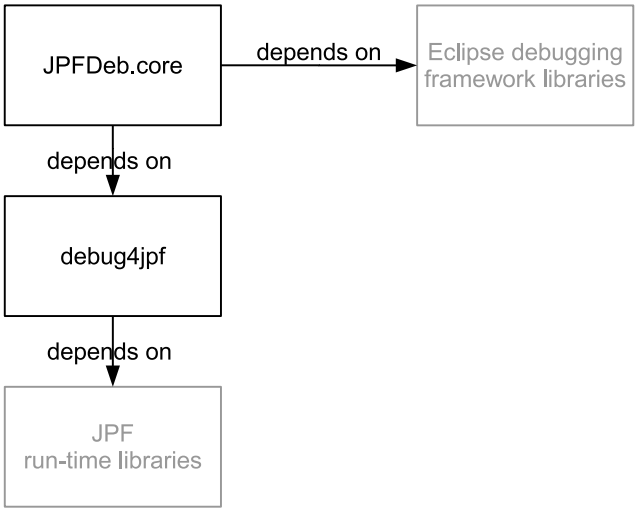


Figure 5.1: Projects and libraries dependencies

5.1.1 The debug4jpf Project Structure

The debug4jpf project is a standard Java project which is linked with JPF runtime libraries. This project is stuctured into the following packages:

cz.cuni.mff.dsrg.debug4jpf.jpf – This package contains classes responsible for instantiating JPF (exploration run and debug run).

cz.cuni.mff.dsrg.debug4jpf.jpf.command – This package contains classes that handle commands received from JPFDeb.core plugin.

cz.cuni.mff.dsrg.debug4jpf.jpf.connector – This package contains classes that handle network transfers.

cz.cuni.mff.dsrg.debug4jpf.jpf.listener – This package contains listeners for both runs of JPF.

cz.cuni.mff.dsrg.debug4jpf.jpf.launcher – This package contains the main class of the project which is used to start the debug4jpf component.

cz.cuni.mff.dsrg.debug4jpf.jpf.util – This package contains several utility classes which are used by classes from the previous packages.

After the Java source files are built into classes, they are packed to a single Java archive called `debug4jpf.jar`.

5.1.2 The JPFDeb.core Project Structure

The JPFDeb.core project is an Eclipse plugin project. Besides the standard Java builders there are additional builders for plugins - Plugin manifest builder, Extension point schema builder, and API analysis builder. All these builders prepare contents of the final plugin archive. This project is stuctured into the following packages:

cz.cuni.mff.dsrg.jpfDeb.core – This package contains classes for the plugin and the launch configuration delegate initialization.

cz.cuni.mff.dsrg.jpfDeb.core.launching – This package contains classes that support launch in run-time.

cz.cuni.mff.dsrg.jpfDeb.core.model – This package contains classes that implement the Eclipse Debug framework data model.

cz.cuni.mff.dsrg.jpfDeb.ui.launching – This package contains class that implements a dialog window with launch configuration from the user interface point of view.

cz.cuni.mff.dsrg.jpfDeb.ui.model – This package contains classes that support the data model on the user interface level.

The JPFDeb.core plugin is built into zip archive which contains two Java archives - one archive with the sources and one archive with the built classes.

5.2 Installation and User Guide

This section describes installation and usage instructions. Folders and files paths used in instructions are relative paths on the CD attached to this master thesis.

The only installation prerequisite is the proper Java 1.6 installation available on the target computer. Installation and configuration steps for users with the Windows operating system (for UNIX/Linux users, notes with differences are added to steps, if applicable):

1. Download and extract Eclipse v3.7.2 archive `eclipse_XXbit.zip`, which is located on the CD in the directory `/resources/Eclipse/` or the latest Java developer version of Eclipse can be downloaded from the internet[2], to a local directory (in further text referenced as `<ECLIPSE_DIR>`).
   Note: UNIX/Linux users should use `eclipse_linux_XXbit.tar.gz`.

2. Download and extract JPF r617+ release archive `jpf-core.zip` which is located on the CD in the directory `/resources/JPF/`, to a local directory (in further text referenced as `<JPF_DIR>`).
   Note: It is not recommended to download JPF from the internet as JPF Java API versions are highly non-backward compatible.

[2] www.eclipse.org

3. Copy plugin Java archive `JPFDeb.core_1.0.0.jar` which is located on the CD in the directory `/resources/plugin/`, into the local directory `<ECLIPSE_DIR>/eclipse/plugins/`.

4. Run Eclipse by executing the `eclipse.exe` binary file located in the local directory `<ECLIPSE_DIR>/eclipse/`.
   Note: UNIX/Linux users should execute the `eclipse` binary file.

5. In the top menu, open menu item Window → Preferences → Run/Debug → String Substitution and modify "jpfHome" variable value to `<JPF_DIR>/jpf-core/`.

At this point, the JPFDeb.core plugin, the debug4jpf component and Java PathFinder are installed and configured for instant use.

5.2.1 Launch Configuration Setup

Eclipse supports two standard launch configuration types - run configuration and debug configuration. If the launch configuration delegate is able to handle both types, configuration of one type is automatically cloned to the other type. JPFDeb.core supports both types, therefore it is neccessary to configure only one configuration. This configuration defines which launch configuration delegate should be used, which class in which project should be run and the input arguments (if applicable).

The basic JPFDeb.core configuration can be created by following these steps:

1. Create new configuration in the Java PathFinder launch group.

2. Write configuration name.

3. Select Java project in which the main class of the tested program is located.

4. Select the main class name of the tested program.

The basic configuration example is depicted in Figure 5.2. The main class `TestDeadlock` (marked as 1) from project `TestJava` (marked as 2) can be checked by JPF by running the launch configuration called "Test deadlock" (marked as 3).

5.2.2 Model Checking in Run Mode

In case the user just want to see the standard output of JPF after the program is checked, then run mode should be used. The JPFDeb.core plugin in run mode starts JPF and prints out the standard JPF output. No enhanced features are active.

Custom JPF properties, e.g. listeners, checked constraints, etc., are supported. The project specific `jpf.properties` file has to be available on the

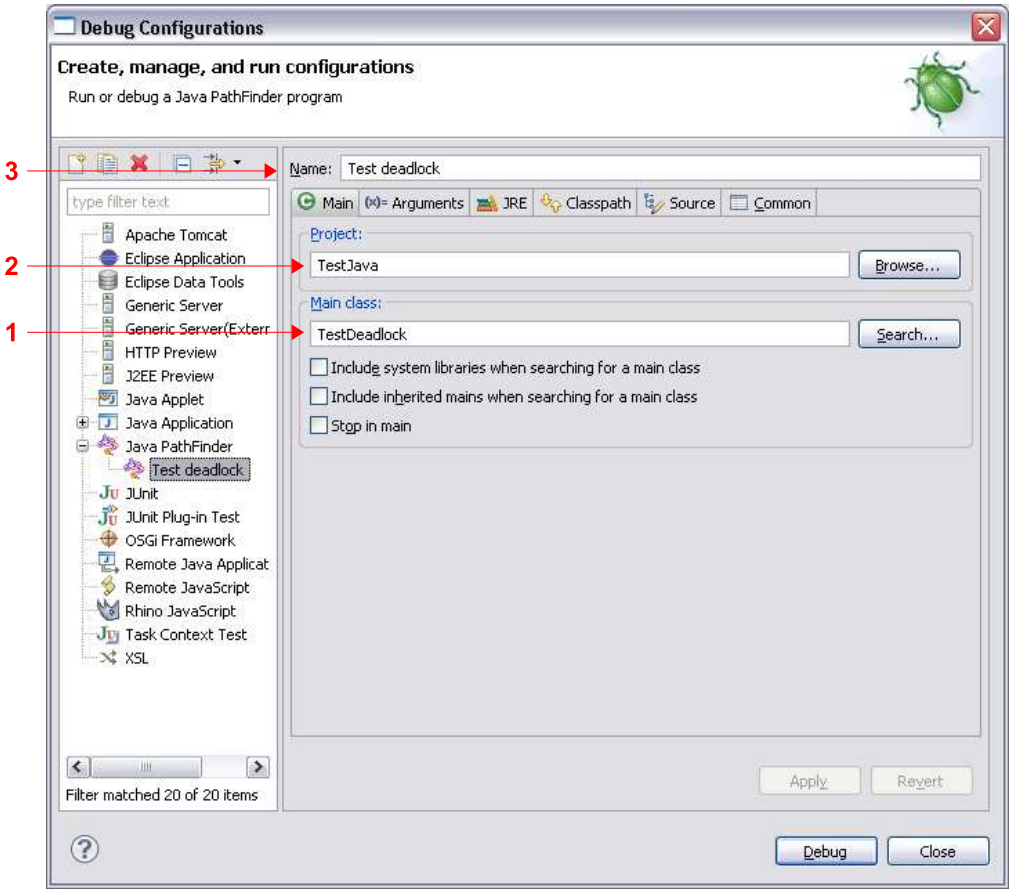


Figure 5.2: JPFDeb.core launch configuration

project classpath. To avoid overwritting changes in JPF configuration properties done by the debug4jpf, it is recommended to put only those JPF properties into property file, that the user want to change.

5.2.3 Model Checking in Debug Mode

On the other hand, if JPFDeb.core is started in debug mode, all enhancements are active and the user can exploit all advantages of this plugin.

Figure 5.3 shows Eclipse workbench right after the program checker is started. There are two important views - Debug (marked as 1) and Console (marked as 2). The Debug view is listing all currently running threads. In the picture, there is one thread called Main and its state is Running. The Console view shows the standard JPF output which is generated by the current JPF run.

As soon as JPF finds an error (if there is any), the processing stops. Figure 5.4 shows the Eclipse workbench right after an error is detected. The Debug view lists all threads (marker as 1) with their states and after extending the tree node representing thread, stack frames (marked as 2) are listed. The user can list all variables linked with the stack frame by selecting one in the tree.

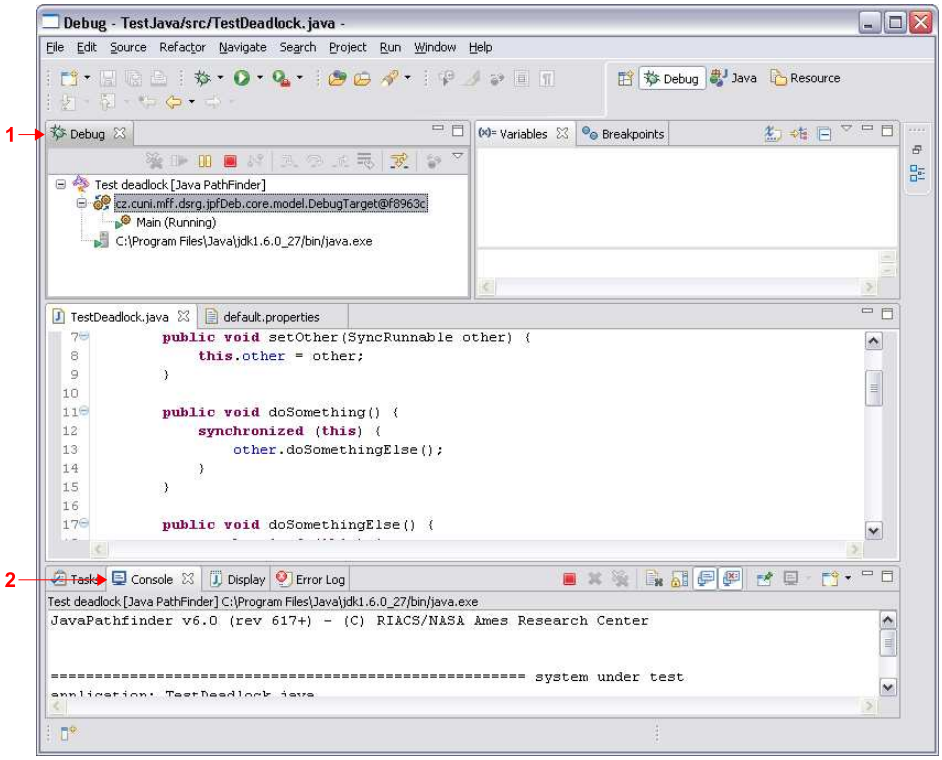


Figure 5.3: Starting the model checker

The variables (marked as 3) are listed in the Variable view. Each stack frame links also the actual source file name and line number. This file is opened in Java editor (marked as 4) and cursor is placed on the actual line (marked as 5).

When the user finishes the current state exploration, plugin processing can be resumed with the Resume button. Finally, the plugin lets the JPF, the debug4jpf and the Eclipse launch finish. All JPF output is written to the Console view, so the user is not deprived of this standard JPF feature that he/she is used to.

There is one difference in controlling the debug launch execution. While the user is allowed to control each thread separatelly, it is not possible in the JPF environment. The reasons for this are listed in Section 4.2. This means that resume operation can be executed only on the level of debug target and not on the thread level.

Breakpoints

The breakpoints are the only control mechanism, that can be used to suspend th JPF states exploration. The step over and step in debugging feature is not available for the reasons listed in Section 4.2.

As soon as the debug run of JPF hits the source file and line number on which a breakpoint is enabled, the state exploration is suspended and the control is passed to Eclipse. At this point, the workbench is behaving the same

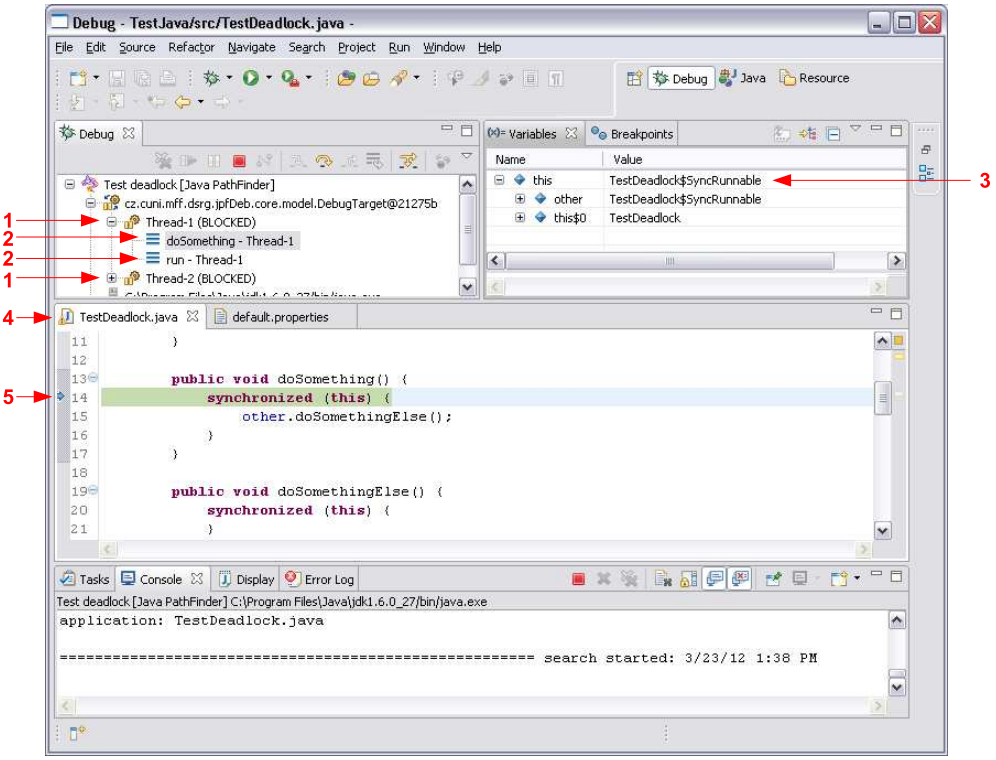


Figure 5.4: The model checker found an error

way as it behaves in case of an error hit.

5.2.4 Sample Projects

Several sample projects are located on the attached CD. They can be used for quick tests of how our approach behaves in various situations.

They can be found in the directory `/samples/` on the attached CD. The list of projects:

- TestDeadlock
- DiningPhilosophers
- NumericValueCheck
- Racer
- StopWatch

These sample projects are originally part of the JPF project and should present the capabilities of JPF. They are reused also for the presentation purposes of our approach capabilities.

In order to run these projects, the user has to import the Eclipse project into the workspace and then continue with the standard procedure described in the prior parts of this section.

Conclusion

The main goals of this thesis were to propose and implement new debugging interface to JPF, which would enable the visualization of all necessary details to easily interpret the JPF results.

The proposed approach was to use Eclipse, the widely spread development environment, as a container for the debugger part of the implementation. There were two subprojects designed and implemented – JPFDeb.core and debug4jpf.

The former one is an Eclipse plugin which display all the details to the user. It uses Eclipse Debug framework for integration into the Eclipse development environment.

The latter one is responsible for controlling and communicating with the JPF instance. The listener is used to control JPF state exploration. This listener is notified each time an event occurs. This lets the debug4jf component inform the Eclipse instance about the state exploration status.

By utilizing this new debugging interface to JPF, the user gets the possibility to receive the JPF results in a visual form. In case of hitting an error, there is a tree view with threads and stack frames as nodes. The user can select any tree item to see the details. If the stack frame node is selected, the list of variables and their values is shown.

All these new features provides the user with the understandable, clear and well-known (same as Java debugging) interface to JPF.

Bibliography

[1] D. Choundhary and V. Kumar. Software Testing. *Journal of Computational Simulation and Modeling*, 1(1):1–9, 2011.

[2] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, 1998.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[4] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[5] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[6] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 6th SPIN Workshop*, volume 1680 of *LNCS*, 1999.

[7] W. Visser, K. Havelund, and J. Penix. Adding Active Objects to SPIN. In *Proceedings of the 5th Workshop on the SPIN Verification System*, Trento, Italy, 1999.

[8] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[9] G. Holzmann. State Compression in Spin. In *Proceedings of the Third Spin Workshop*. Twente University, 1997.

[10] E. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proceeding of the 5th Internation Conference for Computer-Aided Verification*, volume 697 of *LNCS*, 1993.

[11] E. Emerson and A. Sisla. Symmetry and Model Checking. In *Proceeding of the 5th International Conference for Computer Aided Verification*, volume 697 of *LNCS*, 1993.

[12] C. Ip and D. Dill. Better Verification Trough Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Application*, North Holland, 1993.

[13] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Summetry Reductions in Model Checking. In *Proceedings of the 10th International Conference for Computer-Aided Verification*, volume 1427 of *LNCS*, 1998.

[14] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Preceedings of 6th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, 1997.

[15] H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 92–101, 1999.

[16] S. Das, D. Dill, and S. Park. Experience with Predicate Abstraction. In *Preceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, 1999.

[17] H. Saidi and N. Shankar. Abstract and Model Check while you Prove. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 443–454, 1999.

[18] M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 10th Internation Conference on Computer Aided Verification*, volume 1427 of *LNCS*, 1998.

[19] S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs.* ACM Transactions on Computer Systems, 1997.

[20] P. C. Mehlitz, W. Visser, and J. Penix. *The JPF Runtime Verification System*, 2005.

[21] D. Wright and M. Rennie. The Eclipse Debug Framework. In *Proceedings of the EclipseCon 2006 conference.* IBM, 2006.

[22] O. White. JAVA EE Productivity Report 2011. Technical report, ZEROTURNAROUND, January 2011.

List of Figures

List of Tables

Appendix A

CD Contents

The contents of attached CD have the following structure:

```
/
├── javadoc/
│   ├── debug4jpf/
│   └── JPFDeb.core/
├── resources/
│   ├── Eclipse/
│   ├── JPF/
│   └── plugin/
├── samples/
│   ├── DiningPhilosophers/
│   ├── NumericValueCheck/
│   ├── Racer/
│   ├── StopWatch/
│   └── TestDeadlock/
├── sources/
│   ├── debug4jpf/
│   └── JPFDeb.core/
├── thesis/
└── README.txt
```

The directory descriptions are listed in the following table.

Path	Content description
/javadoc/debug4jpf/	Contains generated JavaDoc documentation for the debug4jpf project.
/javadoc/JPFDeb.core/	Contains generated JavaDoc documentation for the JPFDeb.core project.
/resources/Eclipse/	Contains Eclipse v3.7.2 release packages for 32bit/64bit Windows/Linux OS.
/resources/JPF/	Contains JPF r617+ release archive.
/resources/plugin/	Contains JPFDeb.core plugin which includes also debug4jpf archive.
/samples/DiningPhilosophers/	Contains the sample Eclipse project which demonstrates deadlock error.
/samples/NumericValueCheck/	Contains the sample Eclipse project which demonstrates range check error.
/samples/Racer/	Contains the sample Eclipse project which demonstrates race condition error.
/samples/StopWatch/	Contains the sample Eclipse project which demonstrates exploration of off-nominal paths.
/samples/TestDeadlock/	Contains the sample Eclipse project which demonstrates deadlock error.
/sources/debug4jpf/	Contains the source code of the debug4jpf project.
/sources/JPFDeb.core/	Contains the source code of the JPFDeb.core project.
/thesis/	Contains the PDF version of the master thesis.
README.txt	Contains the directories description similar to this one.