

Wikiprint Book

Title: JPF Features and Classification

Subject: Java Path Finder - intro/classification

Version: 3

Date: 02/21/2013 04:31:17 AM

Table of Contents

TracNav	3
Introduction	3
Installing JPF...	3
User Guide...	3
Developer Guide...	3
Projects...	3
About...	3
JPF Features and Classification	3

[TracNav](#)

- [JPFWiki](#) - Welcome Page

[Introduction](#)

- [What is JPF](#)
- [Testing vs model checking](#)
- [Random Example](#)
- [Race Example](#)
- [JPF classification](#)

[Installing JPF...](#)

[User Guide...](#)

[Developer Guide...](#)

[Projects...](#)

- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

[About...](#)

- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

JPF Features and Classification

By now, it should be clear that JPF is not just a model checker: it is a JVM that can be used for model checking. However, if you are familiar with formal methods, you might want to know what kind of model checking methods and features are supported.

Some of the basic model checking traits are:

Explicit State model checking is JPF's standard mode of operation. It means JPF keeps track of concrete values of local variables, stackframes, heap objects and thread states. Other than the intentionally different scheduling behavior, JPF should produce the same results like a normal JVM. Of course it is slower (it is a JVM running on top of a JVM, doing a lot of additional work).

Symbolic Execution means that JPF can perform program execution with symbolic values obtained from how a certain variable was used along the current path of execution (e.g. "x > 0 && x < 43"). Moreover, JPF can even mix concrete and symbolic execution, or switch between them. Please see the Symbolic Pathfinder project documentation for details.

State Matching is a 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. What variables contribute to state matching, and how state matching is performed can be controlled by the user.

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.

Partial Order Reduction is what JPF employs to minimize context switches between threads that do not result in interesting new program states. This is done on-the-fly, i.e. without prior analysis or annotation of the program, by examining which instructions can have inter-thread effects. While this is fast, it cannot handle the "diamond case", since it cannot look ahead of the current execution.

JPF's flexibility is achieved by an architecture that provides a large number of extension points, of which the most important ones are:

- search strategies - to control in which order the state space is searched
- listeners - to monitor and interact with JPF execution (e.g. to check for new properties)
- native peers - to model libraries and execute code at the host VM level
- bytecode factories - to provide different execution models (like symbolic execution)
- publishers - to generate specific reports

In general, flexibility through configuration is JPF's answer to the scalability problem of software model checking.