

Wikiprint Book

Title: Customizable Trace Server

Subject: Java Path Finder - summer-projects/2010-trace-server

Version: 13

Date: 02/23/2013 12:30:55 PM

Table of Contents

TracNav	3
Introduction	3
Installing JPF	3
User Guide	3
Developer Guide	3
MJI	3
Projects	3
About	4
Customizable Trace Server	4
Abstract	4
Architecture	5
Contact	5
Program & Timeline	5
Repository	6
User Guide	6
Description	6
Project Blog	6

TracNav

- [JPFWiki - Welcome Page](#)

Introduction

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

Installing JPF

- [System requirements](#)
- [Download snapshots](#)
- [Download repositories](#)
- [Create site.properties](#)
- [Install NetBeans IDE plugin](#)
- [Install Eclipse IDE plugin](#)
- [Building and testing](#)

User Guide

- [Application Types](#)
- [JPF Components](#)
- [Configuring JPF](#)
- [Running JPF](#)
- [JPF Output](#)
- [The JPF API](#)

Developer Guide

- [Design](#)
- [Choice Generator](#)
- [Partial Order Reduction](#)
- [Attributes](#)
- [Listener](#)

MJI

- [Mangling for MJI](#)
- [Bytecode Factory](#)
- [Logging](#)
- [Report](#)
- [Embedded](#)
- [JPF tests](#)
- [JPF project layout](#)
- [Create a JPF project](#)
- [Coding Conventions](#)
- [Hosting update site](#)

Projects

- [jpf-core](#)
- [jpf-actor](#)
- [jpf-awt](#)
- [jpf-awt-shell](#)

- [jpf-concurrent](#)
- [jpf-cv](#)
- [jpf-delayed](#)
- [jpf-guided-test](#)
- [jpf-mango](#)
- [jpf-racefinder](#)
- [jpf-rtembed](#)
- [jpf-statechart](#)
- [net-iocache](#)
- [jpf-aprop](#)
- [jpf-numeric](#)
- [jpf-symbc](#)
- [jpf-concolic](#)
- [jpf-symbc-load?](#)
- [jpf-extended-test-gen](#)
- [jpf-parallel-spf?](#)
- [eclipse-jpf](#)
- [netbeans-jpf](#)
- [jpf-inspector](#)
- [jpf-shell](#)
- [jpf-template](#)
- [jpf-trace-server](#)
- [standard NB example](#)
- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

[About](#)

- [About this Wiki](#)
- [About the Mailing Lists](#)
- [About the Development Process?](#)
- [About the Repository?](#)
- [How to Contribute](#)
- [JPF contributor account](#)
- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

Customizable Trace Server

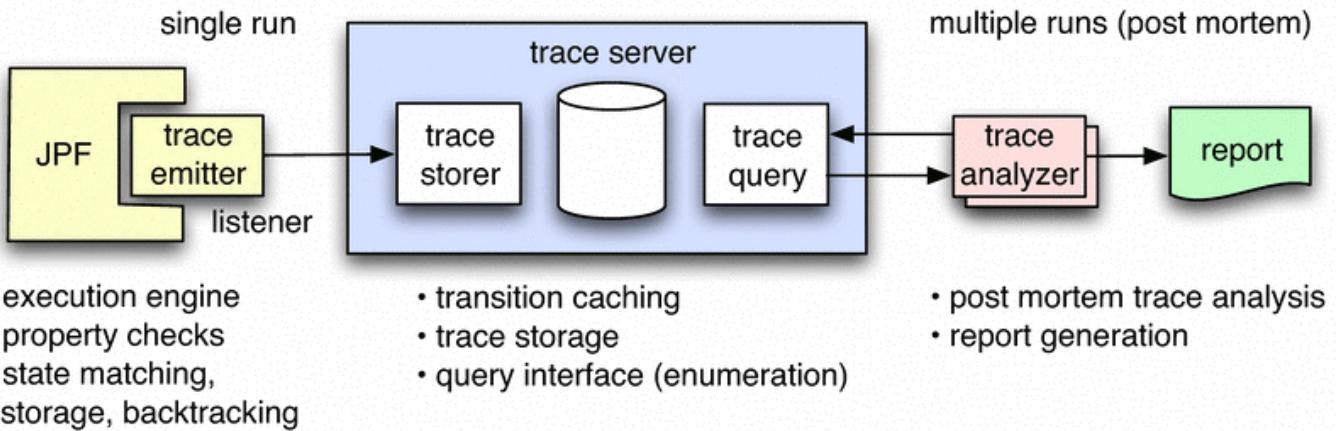
Abstract

Traces are memory hogs. If you have some production code SUT, it is quite normal that you end up with traces that contain millions of steps (Instruction objects). Not good to store millions of objects while you explore the state space and you don't even know yet if you are ever going to need the trace. The normal mitigation is to first run JPF without the "trace" topic in the reports, store the ChoiceGenerator path if you hit a defect or otherwise need a trace, and then replay this path with traces turned on if you need more information. This is a bit complicated. It would be better to configure a "trace server" interface that is just a normal [listener](#) on instructionExecuted, stateAdvanced and stateBacktracked events and

stores/updates transitions/executed instructions (potentially with additional data like operand values) in a database, preferably outside the JPF process so that you don't eat up memory JPF needs for program execution and state storage. Once you have such a database, you can use post mortem analyzers to find out about defects etc. Listeners like the `DeadlockAnalyzer` really should be implemented that way, since there is no need to run them while JPF is still searching for a deadlock. Post mortem analyzers would not only speed up JPF in the first place, but also avoid having to re-run JPF on a large system under test if you need to try several trace analyzers. This project could be split into designing/implementing the VM-trace server interface, and in developing an instance of such a server that provides a post-mortem analyzer interface (e.g. using the `DeadlockAnalyzer` reimplementation as an example).

JPF extensions would also benefit from the ability to augment traces with their custom information. The Trace Server framework should therefore be extensible enough to allow several listeners to augment a trace with extra data.

Architecture



The trace server contains a database and two front ends, one for storing and one for querying data. During the execution of the SUT by JPF, a *trace emitter* stores data using the *trace storer* interface. This decoupling allows multiple emitters to listen for events, allowing JPF extensions to easily augment the trace information database with custom data. To simplify the creation of trace emitters, a default trace emitter should take care of tying events to the right JPF states. Custom trace emitter then augments that empty state information with data.

The trace server may reside on a different machine (host) than JPF, allowing both entities to use the full amount of memory on their machines. Report generation may be done during execution (on the fly) or post-mortem, after the program has terminated.

On the analysis side, a *trace analyzer* queries the database, generating a number of reports as a result. Report generation in the analyzer should be handled in a way such that events generated by extensions can be included in the trace. This also applies to cases where the analyzer has not been written with these additional events in mind. For example, a deadlock analyzer may by default generate a report consisting of a sequence of method calls. Additional information may be provided by JPF extensions, for example, lock identities. To achieve this, extensions that add custom events therefore also have to provide an implementation that includes that data in human-readable form in a report. Event items should therefore implement interface functionality for storage and retrieval (such as serialization), and reporting (e.g., conversion to a string).

We plan to use the non-relational database Neo4J for storing the trace information. Neo4J does not need a predefined schema, and it natively supports graph structures, which makes it easy to extend node/edge properties. The database can be queried either through the Neo4J API or through its query language. The goal is to make trace analysis simpler and more concise than in the current version, so a syntactically terse way of extracting trace information is desirable.

Contact

student: Igor Andjelkovic <igor.andjelkovic "at" gmail.com> (IA)
mentor: Cyrille Artho <cyrille.ortho "at" gmail.com> (CA)
co-mentor: Peter Mehltz <pcmehltz "at" gmail.com> (PM)
co-mentor: Darko Marinov <marinov "at" illinois.edu> (DM)

Program & Timeline

This project is funded by the [Google Summer of Code \(GSoC\)](#) program. It follows the [GSoC timeline](#) (start 05/24, finish 08/16)

Milestones:

1. Support for generic state space change events to manage incoming data in the trace storer (associate data to the right state/transition).
2. Deadlock analyzer (analyzer for cyclic deadlocks could be done using lock acquire/release events, analyzer for other deadlocks with thread status information, i.e., thread activation/suspension).
3. OverlappingMethodAnalyzer (a kind of data race analyzer).
4. Generic error trace pretty-printer (output similar to current error trace).
5. Extension of the pretty printer with custom information (e.g. parameter data), to show extensibility of the framework for JPF extensions. In other words, the code for adding/showing that custom information should be in a class separate from the default error trace code.

Repository

The sources for this project are available from a Mercurial repository on <http://bitbucket.org/igor.andjelkovic/jpf-trace-server>

User Guide

[User guide](#) describes how to use the current version of the `trace server`.

Description

Project Blog

(most recent on top)

2010-06-02 (CA) - project has started, repository on bitbucket has moved to jpf-trace-server

2010-05-18 (CA) - added info about Neo4J, milestones

2010-05-10 (CA) - overview of architecture

2010-05-07 (CA) - added abstract/stub for architecture

2010-05-06 (PM) - project page created