Java Pathfinder at SV-COMP 2019 (Competition Contribution)

Cyrille Artho[1(✉)] and Willem Visser[2]

[1] EECS, KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
artho@kth.se
[2] Department of Computer Science, University of Stellenbosch, Stellenbosch, South Africa
visserw@sun.ac.za
https://people.kth.se/~artho/

Abstract. This paper gives a brief overview of Java Pathfinder, or jpf-core. We describe the architecture of JPF, its strengths, and how it was set up for SV-COMP 2019.

Keywords: Java Pathfinder · Software model checking · Java program analysis

1 Verification Approach

Java Pathfinder (JPF) is a framework for Java bytecode analysis [13]. At the core of the system is an explicit-state model checker [4], often just called JPF (but officially called jpf-core). This core can be extended to allow a variety of other analyses, most notably there is an extension for doing symbolic execution, called Symbolic Pathfinder [9]. Here however we focus only on the core system, i.e., on the explicit-state model checker.

JPF is a mature system with its first version released in the late 1990s. It was first open-sourced by NASA in 2004 and since around 2016 it is a community project hosted on GitHub [12]. It is based around the core algorithms for doing on-the-fly explicit state model checking, similar to SPIN. Unlike SPIN however, it does not support temporal logic property checking by itself. Instead, this functionality can be added as an extension; the core system used here only checks for uncaught exceptions (which include assertion violations).

2 Software Architecture

The main architectural component of JPF is a Java virtual machine (JVM), implemented in Java. This component supports functionality for executing bytecode as well as backtracking over already executed code. Additionally a fingerprint of each state of the JVM (using hash-compaction [5]) is stored to allow

C. Artho—Jury member.

D. Beyer et al. (Eds.): TACAS 2019, Part III, LNCS 11429, pp. 224–228, 2019.
https://doi.org/10.1007/978-3-030-17502-3_18

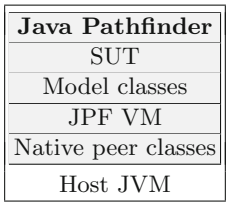

Fig. 1. Architecture of JPF

state-matching and to keep the analysis linear in the size of the state-space of the program being analyzed. Another optimization to allow for the efficient analysis of concurrent programs is a form of partial-order reduction that groups bytecode together in a transition as long as they cannot have any visible effect on other threads. Note that both the hash-compaction and the implementation of partial-order reduction (JPF group instructions that can only have a local effect into the same transition, but this is based on heuristics) used can cause behaviours to be missed during analysis and for this reason JPF is only a bug-finding tool, not a verification tool.

At its core, JPF treats any source of non-determinism as a choice; common choices are scheduling choices and non-deterministic choices over a range of values, e.g., whether a network is available or not [3]. In the context of SV-COMP, symbolic inputs were always treated as entirely non-deterministic choices when using jpf-core.

Java Pathfinder itself is written in Java and therefore runs on the standard JVM, called *host JVM* (see Fig. 1). The system under test (SUT) is run inside the JPF VM, and cannot directly interact with the host VM. This allows JPF to capture the full address space of the program, and revert any changes in memory when backtracking the state of the SUT during the state space search. However, this approach cannot handle native methods, which execute unmanaged code (written in C or C++) that is not supervised by JPF. Changes to memory by native methods, or other side effects thereof, are not visible to JPF. To overcome this limitation, JPF allows *model classes* to be defined, which replace the standard library classes with custom code. With this mechanism, a class with native methods can be replaced with a Java-based model class that does not use native code. Such model classes are fully managed by JPF. Sometimes, though, it is necessary to access native code, for example, to perform input/output. To achieve this, JPF supports *native peer classes*, which are executed directly on the host VM. This means that any built-in library functionality (such as I/O) is available to native peer classes. Furthermore, native peers have access to low-level data structures inside the JPF VM, and thus can read and modify the state of model classes or any other classes that are managed by the JPF VM.

Java Pathfinder is highly extensible and modular. Its VM can handle different platforms and instructions sets (such as Java bytecode and Dalvik code for Android), use different state space exploration strategies and schedulers, and

also allows listeners to receive notifications of program state changes or execution actions, allowing users to build run-time monitoring algorithms on top of JPF. JPF extensions are vital to expand its capabilities, and allow it to handle features like the verification of distributed systems [7,11], generating missing native code on the fly [10], or monitor temporal-logic properties [8].

3 Discussion of Strengths and Weaknesses of the Approach

As expected, JPF performed very well on examples with simple non-deterministic inputs such as Boolean parameters. In this case, the state space is small enough that an exhaustive search is easy, and there is no need to track path conditions (which are implemented by the Symbolic Pathfinder extension [9]). It is therefore also unsurprising that JPF did poorly in some cases where a constraint solver is required to analyze the full state space effectively. An example would be `assert3` in the jbmc-regression suite where an error occurs when the input satisfies the constraint $i \geq 1000 \wedge i \leq 1000$. JPF will only enumerate the inputs for small ranges of values. Because the range of i is not directly specified, but indirectly derived through constraints, JPF does not analyze this variable and therefore misses this error. Finally, there were a few cases where JPF did not conclude its analysis due to missing model classes or native peers.

Java Pathfinder would really excel when analyzing simple to moderately complex concurrent applications, and applications using advanced functionality like input/output and network communication. In the 2019 benchmark set, no concurrent applications were present, so JPF is not fully utilized in this preliminary evaluation. The addition of networked applications would require additional configuration information, so it is therefore not clear how soon the benchmark suite can be extended with such additional, realistic applications. Examples that have been successfully verified by JPF in the past include a WebDAV client [2], and scp client [6], and HTTP servers [1,7].

4 Tool Setup and Configuration

Java Pathfinder is available on GitHub [12]; the submitted compiled version is archived under https://gitlab.com/sosy-lab/SV-COMP/archives-2019/blob/master/2019/jpf.zip.

Java Pathfinder (jpf-core) has no external dependencies; JUnit is necessary to run the unit tests, but not to build and use jpf-core.

JPF is compiled with `./gradlew build`.

We used the default values for all options, except that `cg.enumerate_random` was changed to `true` from the default configuration, because this option forces JPF to explore all possible values for random choices.[1] This setting was necessary

[1] JPF will explore all outcomes for Boolean choices, and a set of predefined corner cases for choices on integers.

to enable JPF to explore non-deterministic inputs. It is not enabled by default because JPF is normally used to analyze concurrency.

JPF participated in all Java benchmarks.

5 Software Project and Contributors

The project is managed by the Java Pathfinder group. Contact person is Cyrille Artho (`artho@kth.se`). JPF is available under the Apache License, version 2.0 and hosted on GitHub [12].

Acknowledgements. We thank all contributors who have participated in the development of JPF over the last 20 years.

We would also like to thank Peter Schrammel and Dirk Beyer for their support, and for providing the scripts and configuration files for the competition.

References

1. Artho, C., Hagiya, M., Potter, R., Tanabe, Y., Weitl, F., Yamamoto, M.: Software model checking for distributed systems with selector-based, non-blocking communication. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013), Palo Alto, USA, pp. 169–179. IEEE Computer Society (2013)
2. Artho, C., Leungwattanakit, W., Hagiya, M., Tanabe, Y., Yamamoto, M.: Cache-based model checking of networked applications: from linear to branching time. In: Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 447–458. IEEE Computer Society (2009)
3. Artho, C., et al.: Using checkpointing and virtualization for fault injection. IJNC **5**(2), 347–372 (2015)
4. Holzmann, G.: The SPIN Model Checker. Addison-Wesley, Boston (2004)
5. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods Syst. Des. **13**(3), 289–307 (1998). https://doi.org/10.1023/A:1008696026254
6. Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model checking distributed systems by combining caching and process checkpointing. In: Proceedings of the 26th International Conference on Automated Software Engineering (ASE 2011), Lawrence, USA, pp. 103–112. IEEE Computer Society (2011)
7. Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M., Takahashi, K.: Modular software model checking for distributed systems. IEEE Trans. Softw. Eng. **40**(5), 483–501 (2014)
8. Lombardi, M.: jpf-ltl (2013). https://bitbucket.org/michelelombardi/jpf-ltl. Accessed 11 Jan 2019
9. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013), pp. 391–425 (2013). https://doi.org/10.1007/s10515-013-0122-2

10. Shafiei, N., Breugel, F.V.: Automatic handling of native methods in Java PathFinder. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014, pp. 97–100. ACM, New York (2014). https://doi.org/10.1145/2632362.2632363
11. Shafiei, N., Mehlitz, P.: Extending JPF to verify distributed systems. ACM SIGSOFT Softw. Eng. Notes **39**(1), 1–5 (2014)
12. The Java Pathfinder Group: jpf-core (2019). https://github.com/javapathfinder/jpf-core. Accessed 11 Jan 2019
13. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. J. **10**(2), 203–232 (2003)