JPF: From 2003 to 2023*

Cyrille Artho[1][0000−0002−3656−1614], Pavel Parízek[2][0000−0003−0714−7446], Daohan Qu[3][0009−0004−3811−6591], Varadraj Galgali[4][0009−0007−2086−4542], and Pu (Luke) Yi[5][0000−0001−6669−6520]

[1] KTH Royal Institute of Technology, Stockholm, Sweden artho@kth.se
[2] Charles University, Prague, Czech Republic parizek@d3s.mff.cuni.cz
[3] Nanjing University, Nanjing, China daohanqu@gmail.com
[4] Belgaum, India varad23711@gmail.com
[5] Stanford University, USA lukeyi@stanford.edu

Abstract. We give an account of JPF's current architecture as it has evolved over the last 20 years. Key changes include a modular, extensible design, and Java 11 support.
Java 11 brought with it fundamental changes in the language and its runtime, in particular, a new modular library system, different compilation of string expressions to bootstrap methods, and changes in many internal interfaces that allow access to the loaded code and the virtual machine state. These changes required numerous adaptations in JPF to ensure a successful compilation and correct behavior under Java 11.

Keywords: JPF · Java · Software model checking · Program analysis.

1 Introduction

JPF is a framework for Java bytecode analysis [1,2] that can be used to verify and search for bugs in programs written in Java-like languages. At the core of the system is an explicit-state model checker [3], which can be extended to allow many other analyses, such as symbolic execution [4].

Earlier papers covered the original architecture of JPF as a virtual machine for Java bytecode [1] or gave an abridged account of the current architecture [2]. This paper gives a detailed description of the current architecture, which is much more modular and extensible than 20 years ago and supports native methods through a well-designed interface.

As part of the developments of the last two decades, Java 11 was the first long-term release that brought major changes to Java and gave up on full backward compatibility.[6] Key changes at the bytecode level include a new modular library system, a different compilation of string expressions to bootstrap methods, and the removal of or changes in many internal interfaces [5].

* Supported by Google Summer of Code.

[6] While most of these changes were introduced with Java 9 as a development release, we will group any changes between Java 8 and Java 11 under the latter.

For its program analysis, JPF has to support the full functionality of Java bytecode and integrate closely with the underlying Java Virtual Machine (JVM). Thus, it is impacted by internal changes of the Java platform that do not affect most other applications. This was evidenced by JPF first not even compiling under Java 11. After one year, we had adapted the code base so it compiled, but about 75 % of all regression tests failed initially. Four years of additional work addressed the major changes that were needed to support Java 11. During this time, we added over 140 new regression tests (and removed a few obsolete ones) and implemented new functionality with about 10,000 lines of additional code.

This paper is the first detailed publication presenting JPF's architecture and capabilities as they have evolved over the last 20 years since the early version of JPF, which was designed differently [1]. It also gives an overview of the challenges involved in adapting a bytecode-level program analysis tool to major architectural and implementation-level changes of the underlying platform. The remainder of this paper is organized as follows: Section 2 covers the background and related work, while Section 3 describes JPF's architecture. Section 4 describes the key changes from Java 8 to Java 11 and the adaptations in JPF to support them. Section 5 covers other major enhancements from the last five years, and Section 6 shows the evolution of JPF over that time. Finally, Section 7 summarizes our work and concludes.

JPF is freely available on GitHub, in the repository `https://github.com/javapathfinder/jpf-core/`. Extensive user and developer documentation, including an installation and how-to-run guide, is provided in the form of wiki pages at `https://github.com/javapathfinder/jpf-core/wiki`.

2 Background and Related Work

JPF is an extensible framework for Java bytecode analysis [1,2] and built as an explicit-state model checker [3].

In order to explore all possible and relevant outcomes of a program execution, JPF explores all possible outcomes of non-deterministic choices. Such choices can be induced by a non-deterministic thread schedule or unspecified variables/inputs. JPF has the ability to backtrack an execution to a previous point to analyze alternative outcomes. Therefore, it implements a fully-fledged JVM by itself but uses the underlying JVM (the "host JVM"; also see Section 3) to access the underlying platform's functionality. This access happens by delegating native methods at the JPF level to the host JVM.

By default, JPF reports a failure if an uncaught exception occurs or an assertion is violated. Another type of failure can occur due to a deadlock (defined as no remaining runnable thread being able to make progress, either due to waiting on a lock that is being held by another thread or waiting for a notification that never occurs). The set of built-in properties that JPF is able to check includes also the absence of data races. JPF reports a data race when multiple threads access the same memory location without synchronization and at least

one of the accesses is write. To implement their own properties, users can add listeners to support, e. g., temporal-logic properties [6].

2.1 History of JPF

JPF started in 1999; it and its community evolved significantly in the time since, due to JPF being reimplemented and rewritten and eventually published under the Apache 2 License. We outline the key milestones here:

1999: First version of JPF, developed at the NASA Ames Research Center in the form of a translation from Java to Promela [7]. This first version was limited because regular model checkers like SPIN [3], which analyzes Promela models, cannot handle the dynamic creation of objects and threads (unless an upper bound is known at compile time).
2000: Reimplementation as a concrete virtual machine for Java bytecode that can backtrack execution [1]. JPF has used this approach since then.
2003: Introduction of extension interfaces and the architecture that modern JPF has until today.
2005: JPF was released as open-source software on Sourceforge, being the first NASA software project to be released in this way.
2008: First participation in Google Summer of Code, which supports students working on open-source software with stipends.
2009: JPF moved to its own server, hosting extension projects and the documentation (wiki).
2017: Moved to GitHub. This allowed JPF to implement continuous integration [8] and accept outside contributions more easily.

2.2 Related Work

JPF is an explicit-state model checker for Java bytecode at its core. It inspired similar works such as JNuke [9] and Moonwalker [10], which implement model checking for Java bytecode or .NET code without native methods, respectively.

Other tools that analyze programs by using their own execution engine include Valgrind [11], which looks for incorrect memory usage (corruption, leaks) in binary programs using dynamic analysis, and KLEE [12], which implements a symbolic execution engine on top of the LLVM [13] infrastructure.

Other software model-checking approaches are closer to the first version of JPF [7] and analyze code after transforming it into a representation that can express that entire state space at compile time. Examples include SLAM [14], which converts C code to a Boolean program for model checking, and CBMC [15], which uses a SAT solver on propositions derived from C code. This approach is more popular for analyzing C code because the inability to handle dynamic memory allocation and thread creation is less relevant for C programs where memory and thread usage are often bounded, especially for safety-critical systems [16].

Also, there exist several dynamic analysis frameworks that can be used to detect runtime errors by monitoring the execution of a program within a particular

virtual machine. Notable examples include RoadRunner [17] and DiSL [18] for Java programs running on JVM, and SharpDetect [19] for C#/.NET programs. All these frameworks work on the same principle, recording specific events and the program runtime state on a dynamic execution trace, and forwarding this information to a custom analysis plugin that detects the actual errors. They are useful especially for multi-threaded programs and discovering possible concurrency errors (e. g., deadlocks and race conditions).

Table 1 gives an overview summary of the aforementioned tools by looking at their overall approach (state space exploration vs. runtime monitoring) and supported platforms. In this table, "state space exploration" can refer to model checking or symbolic execution. The table shows that related tools are too different (in terms of the principal approach or platform) for a straightforward quantitative comparison. In particular, we are not aware of any tool for Java bytecode with similar types of capabilities that JPF now has.

Table 1: Comparison of JPF with related tools by approach and target platform

		Platform			
		Java bytecode	x86 code	CIL bytecode	C source code
Approach	State space exploration	JNuke (Java 5) JPF (Java 11)	KLEE	Moonwalker	SLAM CBMC
	Runtime monitoring	RoadRunner DiSL	Valgrind	SharpDetect	

3 JPF's Architecture

JPF's architecture separates bytecode execution from the functionality of library classes, access to the underlying host JVM, and various ways of adapting and extending the built-in functionality.

3.1 Functionality

At its core, JPF implements a virtual machine for bytecode instructions. The entire state of a program (with the state of all its threads and the shared heap) is managed by that virtual machine. Unlike a regular virtual machine, JPF is capable of keeping copies of past program states and comparing them to other states. Past states can be restored from these copies, which allows JPF to implement a state space search of a program.

The state space search algorithm is configurable (depth-first search being the default setting) and by default explores all outcomes of all non-deterministic choices. This includes thread scheduling in case multiple threads are enabled at a given state and the outcomes of all possible values of a non-deterministic data choice. Such choices are either implemented through the `Verify` application

programming interface (API) or extensions such as SPF [4] and typically model unspecified user inputs.

Sequences of instructions that do not exhibit any choices are executed inside a *transition* in JPF. A transition ends whenever a choice is hit during execution. Therefore, JPF computes the state space of a program *on the fly*, as the extent of a transition depends on the instructions therein and their side effects. Partial-order reductions optimize the state space and avoid unnecessary thread scheduling choices. JPF is close to being a sound verification tool in the sense that there are very few cases of property violations that it misses, but a few implementation choices in the state hashing and partial-order reductions are not sound in all cases, which makes JPF a bug-finding rather than a verification tool in the strict sense [2], unless the default behavior is changed so a fully exhaustive search is used, at the cost of being significantly slower in certain cases. Several extensions aiming to make JPF a sound verification tool have been already proposed, including the support for sound dynamic partial-order reduction [20] and coverage of all behaviors permitted by the Java memory model [21,22].

3.2 Design

JPF itself is written in Java and runs on the so-called *host JVM,* which provides internal functionality such as loading classes or interacting with the system via native methods [1,2]. The system under test is executed by JPF's virtual machine, which keeps track of any effects (such as changes to memory) of an executed instruction (see Figure 1).

The main functionality of JPF is implemented in `jpf-core`, while optional extensions can extend that functionality.

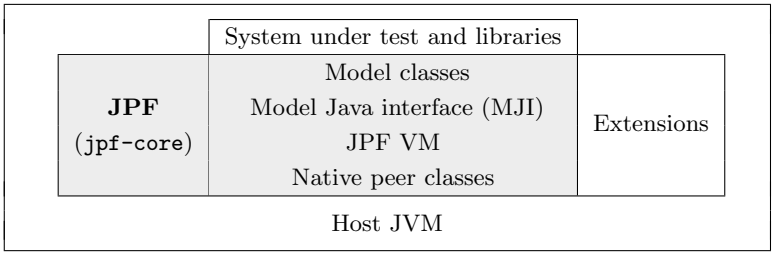

Fig. 1: Architecture of JPF

The functionality of JPF (`jpf-core`) itself is divided into modules (see Table 2). Module `main` implements the core analysis capability, while other modules (explained below) implement models of library classes (`classes`), a bridge to the underlying host JVM (`peers`), or auxiliary functionality.

Table 2: Main JPF modules and their purpose

Module	Purpose
`annotations`	Run-time annotations in programs analyzed by JPF
`classes`	Model library classes
`examples`	Small example programs
`main`	Core functionality (VM, state search, etc.)
`peers`	Native peers for accessing JPF from model classes
`tests`	Unit tests

Main. JPF has a very extensible design that allows developers to customize almost any functionality. The base implementation of the VM in `main` and its instruction set are generic, and while Java bytecode is the default concrete implementation, other instruction sets can be supported.

The extensibility of JPF is achieved by all key functionality being customizable through interfaces. We present the key interfaces below:

- A `SearchListener` can track events arising from the state space search (e. g., when program analysis starts or ends, when a new state is created, or when an existing already visited state is backtracked to).
- A `VMListener` gets notifications from program execution (e. g., when a method call begins or a certain type of instruction is executed).
- A `ChoiceGenerator` can override the way how non-deterministic events are explored or implement new types of choices.
- Instruction-related interfaces allow changing how sets of instructions or individual instructions are analyzed.
- A `Scheduler` has the capability of changing how the state space is analyzed, e. g., to analyze the state space of multiple processes [23]
- A `PublisherExtension` creates reports on program analysis results.

Classes. Any Java program has to access functions of the Java library; this starts by using the common super class `Object` as the first application-specific class is loaded. Many Java library classes have functionality that is not suitable for JPF's analysis, as they include functionality that incurs globally visible side effects (such as writing to a file) and use native methods. Native methods are not available as Java bytecode but instead implemented by a system-specific run-time library, usually written in C or C++. As JPF only interprets the application-level bytecode, it is not able to track the effects of native methods.

Therefore, classes using native methods have to be replaced by *model classes* (in `classes`), which represent a Java implementation of code that makes invisible side effects (through native method calls) in the regular library implementation visible to JPF at the model class level. In this way, model classes solve the problem of not being able to track the outcome of native code execution. Other approaches have been attempted, such as using process-level virtualization to track the state of an entire operation-system-level process. This approach is less

efficient because it can only analyze the state of a process at a "black-box" level, preventing state compaction or partial-order reduction [24].

A model classes can completely replace the functionality of a library class, if it can implement this entirely in Java, making all (side-) effects visible to JPF. However, access to native methods requires the peers module (see below).

Annotations. This module implements annotations that are visible to JPF at run time. The most important annotations are `@MJI`, which marks a method as a bridge to a native peer, and attributes that affect the state space exploration by ignoring fields during program analysis (`@FilterField`) or marking them as not shared by multiple threads (`@NonShared`).

Peers. Model classes by themselves are limited to functionality that can be implemented directly through bytecode. Much functionality, such as printing to the console, requires access to the underlying run-time environment. *Native peers* bridge this gap between bytecode and native code (see Figure 2). At the model class level, native methods are declared normally (see Fig. 2b, line 12).

To delegate a native method call, JPF uses a so-called Model Java Interface (MJI) layer to specify the *native peer*, a JPF-level class implementing native methods. MJI methods handle parameters from bytecode at the JPF level and pass them in the appropriate form to an actual native function on the host JVM. A native peer usually accesses the host JVM and maps the state of the host JVM object to the JPF-level object and also manages potential side effects of the host JVM method (see Figure 1).

MJI classes and methods follow a name-mangling scheme to encode the package name as part of the class name, and the return type, method name, and parameter list of the underlying native method as part of the method name (see Fig. 2b, line 15). This way, JPF can identify the right method at run time.

Figure 2 shows how the different layers of JPF interact when executing code that prints "Hello, World!". The bytecode first loads a reference to the PrintStream instance and the string "Hello, World!", in order to call `println`.[7] This method (available entirely in bytecode through the Java library) constructs the correct final string with the newline character at the end and internally calls `OutputStreamWriter.write`. That method uses `OutputStreamWriter.encode` to convert the string to a byte array. Because the encoding of a string uses a native method, `encode` is declared as such in the model class. The JPF native peer counterpart is an MJI method, which internally accesses the native character-to-byte conversion functionality of the host JVM.

Generally, a native peer can delegate its functionality to the underlying native method for calls that have no side effects on the Java-level object [25], or it can implement its own logic and manage side effects in a complex way, such as when

[7] To save space, we elide Java package names (java/lang for System and java/util for PrintStream), instruction and constant pool offsets, and the "L" denoting a fully qualified class name.

```
getstatic      // Field System.out:PrintStream;
ldc            // String Hello, World!
invokevirtual // Method PrintStream.println:(String;)V
```

(a) Bytecode to be executed

```
/** Java library: java.io.PrintStream (provided by the JVM) */
public void PrintStream.println(String s) {
  ...
  OutputStreamWriter.write(...);
}

/** JPF model class: java.io.OutputStreamWriter (in "classes") */
public void write(String s, int off, int len) throws IOException {
  ... = encode(...);
}
// native method declaration in the model class
private native int encode (String s, int off, int len, byte[] buf);

/** JPF native peer: JPF_java_io_OutputStreamWriter (in "peers") */
@MJI public int encode__Ljava_lang_String_2II_3B__I (MJIEnv env,
                int objref, int sref, int off, int len, int bref) {
  ... // access to the host JVM
}
```

(b) Model class interfacing with the native peer via MJI

Fig. 2: Interaction between code of the Java library (line 2), a model class with a native method (line 12), and its native peer (line 15).

synchronizing program states between an application that is analyzed by JPF and external applications that are connected through the network [26].

Tests. Tests contain over 1000 unit tests that check the internal functionality of JPF, ensuring that key Java language or library features work correctly. Some tests verify the verdict of a full program analysis on a small example.

Examples. A couple of small examples are also provided, along with their configuration files, to exemplify the usage of JPF.

3.3 JPF extensions

JPF *extensions* are modular implementations of enhancements to JPF. They are separate projects that are independent of the main part of JPF and implement additional functionality, e. g., by overriding how unspecified values of variables are interpreted or how an application interacts with its environment.

Notable extensions include symbolic execution [4], automated support of stateless native methods [25], and automated support of certain types of networked applications using either a centralization or a caching approach [23,26].

3.4 Program execution using JPF

When JPF is used, it typically is run with a configuration file that specifies the application under test. JPF then proceeds as follows:

1. The configuration file is parsed and extensions are loaded as specified.
2. The application main class and the necessary library classes are loaded. Program execution begins at `main`.[8] Execution covers the bytecode of the program under test, Java libraries (without native methods), and model classes.
3. Any instruction that creates a new thread, affects the state of another thread, or produces a non-deterministic choice for other reasons is handled by a `ChoiceGenerator`, which causes the current transition to end. New transitions are scheduled and added to the state space search.
4. Any time a model class declares a Model Java Interface method, execution is handled by the corresponding native peer method inside JPF. These methods are able to access (possibly native) methods of the underlying host JVM.
5. The analysis stops when JPF has explored the entire state space of a program, runs out of memory, or finds a property violation to report.[9]

3.5 Challenges in modeling Java library classes

The main challenges in writing a model class (and if needed, its native peer) are the following:

- A model class has to reflect the functionality of the original Java class faithfully; differences may result in an overapproximation of the behavior under JPF, or an unsound underapproximation.
- While a model class can hide side effects of native methods, these side effects are often an essential part of the program behavior, such as for networked applications [23,26]. When native methods interact with the environment, major changes in JPF are necessary to handle the side-effects of both the host JVM and its environment (the underlying operating system) [26,24].
- Because a native peer interacts with the host JVM, it often has additional state information compared to the model class. Care has to be taken that this additional state (which is used by native methods) remains consistent with the state of the model class (which is visible and used by non-native methods).
- It is not possible to create a model for only selected methods, so a model class has to support the entire public API that the program under test requires. Furthermore, it is often not possible to replace a single class in isolation, as multiple classes in the same package or even related packages (such as `java.io` and `java.net`) often interact and have to be replaced as an ensemble.
- The Java base classes (in module `java.base`) in Java 11 alone contain 190 native methods, so the effort of supporting all of them is prohibitive. JPF therefore focuses on the most commonly used native methods.

[8] More classes are loaded at run time as needed [27].
[9] One can specify that JPF continue the state space search after a property violation.

4 Adaptations in JPF for Java 11

Several changes in compiled Java code (bytecode) from Java 8 to Java 11 heavily affect runtime environments, including JPF, and even the build system. Here, we describe these challenges and our solutions to them.

4.1 Module system

Java 11 introduces a module system that provides an additional unit of encapsulation on top of Java packages [28]. Modules have to declare their dependencies explicitly and can also declare the services they provide. In Java 11, built-in modules are bundled in a new archive format (JMOD). Thus, JPF no longer reads the classes directly but delegates reading class files from these archives to the host JVM. To support the module layer, we extended JPF's class loader with new functionality to support the module API. In particular, the Proxy API (which is used for reflection) has to support module information in Java 11.

4.2 Bootstrap methods

Java 8 introduces support for dynamic languages and lambda expressions, which are functions that are not bound to an identifier. These functions cannot be fully resolved at compile time. Internally, they are compiled into so-called *bootstrap methods* that instantiate a valid anonymous function at class load time in order to accommodate concrete uses.

This change allows for optimized string handling: In Java 8 and prior, string output is handled by creating a *StringBuilder* instance and appending strings to its buffer. This requires expensive creations of intermediate objects and subsequent conversions of these objects to *String* instances for tasks as simple as adding a number to an output string. In Java 11, specialized bootstrap methods handle string output with non-string parameters much more efficiently.

Figure 3 illustrates this with a simple example. As can be seen, a simple string expression (Fig. 3a) compiles into complex bytecode (Fig. 3b) under Java 8. The resulting code produced by the Java 11 compiler is much more compact (Fig. 3c); however, most of the functionality is delegated to the bootstrap method *makeConcatWithConstants*, which takes an integer parameter and returns a string. While we cannot show the details here, one can see that parameter `i` is part of the bootstrap method, but the string constant "Number" is not a runtime parameter. The bootstrap method is expanded into a callable anonymous function (a *call site* [29]) at runtime by the class loader, which adds the string constant, before it can be called by the bytecode instruction `invokedynamic`.

When a class is loaded, the target of an *invokevirtual* instruction (a call to a dynamically generated method) has to resolve to a valid call site [29]. The call site is generated from the bootstrap method. The bootstrap method, e. g., `makeConcatWithConstants` in Fig. 3c, instantiates a complete call site by creating an anonymous function using a concrete value ("Number" in Fig. 3d) for the string constant.

```
public static void print(int i) {
  System.out.println("Number " + i);
}
```

(a) Source code

```
getstatic      // Field System.out:PrintStream;
new            // class StringBuilder
dup
invokespecial  // Method StringBuilder."<init>":()V
ldc            // String Number
invokevirtual  // Method StringBuilder.append:(String;)StringBuilder;
iload_0
invokevirtual  // Method StringBuilder.append:(I)StringBuilder;
invokevirtual  // Method StringBuilder.toString:()String;
invokevirtual  // Method PrintStream.println:(String;)V
return
```

(b) Compilation with Java 8

```
getstatic      // Field System.out:PrintStream;
iload_0
invokedynamic  // makeConcatWithConstants:(I)String;
invokevirtual  // Method PrintStream.println:(String;)V
return
```

(c) Compilation with Java 11

```
BootstrapMethods:
  0: #19 REF_invokeStatic StringConcatFactory.makeConcatWithConstants:
  (MethodHandles$Lookup;String;MethodType;String;[Object;)CallSite;
    Method arguments:
      #20 Number \u0001
```

(d) Bootstrap method structure

Fig. 3: String handling and output under Java 8 and Java 11

Lambda expressions are also handled internally via bootstrap methods. Some lambda expressions are serializable, in particular in `java.util.Comparator`, which is often used for sorting.

4.3 Bootstrap methods in JPF

JPF for Java 8 had limited support for bootstrap methods, handling a few common cases. Due to the more widespread use of lambda expressions in Java 11 (especially for string concatenation and output), shortcomings in the earlier implementations had to be addressed. The internal implementation of OpenJDK for bootstrap method resolution generates the bytecode of the call site at load time. This is complex and involves internal APIs and native calls that JPF does not support. JPF instead models the behavior of the bootstrap method and implements its own support of `invokedynamic` for string concatenation.

When used to concatenate strings, instruction `invokedynamic` calls a function that takes arguments to be concatenated and returns the resulting `String` instance. JPF can easily implement this at VM level if all the arguments are `String` instances or primitive types, since their string representation could be

easily constructed from their meta-data stored at VM runtime. However, for other reference type arguments, their `toString()` methods have to be called. This method call no longer happens in the form of a method call in the bytecode but is done automatically by the JVM. JPF mimics this behavior to support Java 11 string expressions by analyzing all arguments on the operand stack to convert them to a string if needed. Our approach therefore avoids the complex dynamic bytecode generation of OpenJDK 11.

To support serializable lambda expressions, OpenJDK uses a special bootstrap method called `altMetafactory`. As JPF cannot use that mechanism, we need another approach to handle serializable lambda expressions: JPF creates an object that implements the target interface of a lambda expression, which happens to make serialization easier — we only need to add `Serializable` to this object's implemented interface list, and the object serialization mechanism can then serialize the lambda expression automatically.

4.4 Reflection

One of the primary goals of adding modules in Java 11 was strong encapsulation, which attempts to limit reflection and promote the modular approach to improve security. As a consequence of this, the reflection API no longer permits access to private fields without issuing a warning (as of Java 11); in later releases, such access is denied by default [30] or removed entirely [31]. Under Java 11, various tests raised warnings due to illegal reflective access taking place. The problem was caused by the executing code using reflection, trying to access non-public fields/methods residing in different modules.

To fix the problem, it needs to be ensured that the accessing code present in the module has access rights to the module it is trying to access the field/method from. Specifying the `--add-opens` option to `jvmArgs` within `build.gradle` for the necessary packages fixes the problem, ensuring that JPF will not break due to illegal-reflective access errors when using newer versions of Java.

4.5 Internal APIs

Many implementations of the Java API (`java.*` packages) use internal APIs, such as `com.sun.*` and `jdk.internal.*`, in their implementation. Such internal APIs are often highly dependent on the native methods and the underlying JVM.

After Java 8, many such internal packages have been replaced with new packages under `jdk.internal`. Replacement for existing functionality is usually available under a redesigned API, such as `java.lang.StackWalker`, which provides more flexible and efficient stack traversal functionalities, like lazy traversal, frame filtering, and criteria for stopping.

The most straightforward strategy to minimize dependency on internal APIs is to use a model class, which replaces the problematic class entirely. However, providing an accurate model class is a challenge of its own. For example, JPF for Java 8 uses a model class for `DateFormat`, providing a simplified implementation of key date methods. The changes in Java 11 also affected date handling and

caused four regression tests to fail. The increased complexity of the Java 11 implementation made it difficult to maintain a model class for such functionality.

To enforce strong encapsulation of JDK internals, any JPF constructs using internal data have been rewritten by leveraging model classes and Model Java Interface (MJI) components to intercept method invocations and delegate them to dedicated classes. Notable updates include the `StackFrameInfo` model class that provides support for the *StackWalker* API and the `ServiceLoader` model class that ensures support for the *DateFormat* API. Additionally, the `MethodHandles` mechanism is used to support *CountDownLatch*, *ExecutorService*, and *Semaphore*, which are utility classes to support concurrent programming. We also added support for the `PlatformClassLoader` class and improved `SecureClassLoader` to help with service provider loading.

Many of these changes (such as the `ServiceLoader` model class and the `MethodHandles` mechanism) lift the layer at which a model class is used to a higher level by supporting general delegation mechanisms in the Java library better. This reduces the number of model classes needed and makes it easier to fully support internal functionality, especially when it has few dependencies on native methods.

4.6 Build system

Since 2018, JPF has been using *Gradle* as the build automation tool of choice. For a user, Gradle has the advantage of automatically downloading any dependencies, as long as the available JVM used to run Gradle supports them.

This results in multiple dependencies: The version of Java used has to be able to both run Gradle and compile and execute the software being built and tested. Therefore, targeting a different Java version often requires updating Gradle as well. This ensures the ability to support newer versions of Java and benefit from other improvements in Gradle.

However, these Gradle updates often include major changes that deprecate an old build mechanism or even change Gradle's domain-specific language that is used to declare build tasks. Therefore, major Gradle updates may require a redesign of some build tasks. Deprecations of certain features also usually have an effect with the next major version.

The JPF project has made two major updates in using Gradle: from version 4.7 to version 5.4.1 in 2019, and again to version 8.2.1 in 2023. We eliminated any use of deprecated features to allow at least one more major version upgrade without changes in the future. By redefining tasks using the task configuration avoidance API, which deters creation of unnecessary tasks, the build process became more efficient. We also now use plugins to support publishing to a local Maven repository and measure statement coverage (using JaCoCo).

4.7 Other adaptations

Various other internal changes or defects that were discovered during development for Java 11 support required corresponding adaptations in JPF:

- Support for new internal string representation (JEP 254 [32]), which allows strings in memory to be encoded as either LATIN1 or UTF16. The content of a string is now stored as an array of bytes rather than an array of characters.
- Removal of unnecessary explicit manual boxing of primitive values in objects; adaptations to changes in the unboxing API of primitive classes (`valueOf`).
- Removal of string buffer-related model classes (StringBuilder, StringBuffer) that are no longer needed, because their functionality can be safely delegated to the library of the host JVM.
- Support for the stream API.
- Better support for loading classes from JAR files and URLs.
- Better support for correct type handling.
- Various other fixes (file I/O, internals of threads and concurrency packages).
- Updates in the documentation and renaming of the git branches to make Java 11 the default Java version for JPF.

5 Other Enhancements

Other enhancements that have been included with the Java 11 support include test pollution detection and bit-flip simulation, which are both compact enough to be included in `jpf-core` rather than as an extension of their own.

5.1 Test pollution detection

Flaky tests are software tests that non-deterministically pass or fail. They undermine developers' trust in the test infrastructure, and waste many man-hours to investigate non-existing bugs. Order-dependent flaky tests are a prominent type of flaky tests [33], where polluter tests, the kind of software tests that modify the program state shared among other tests, cause other tests to fail. It is therefore worthwhile to proactively detect test pollution and prevent order-dependent flakiness. A technique dubbed PolDet was developed to detect polluter tests [34].

Because JPF provides infrastructure support for detecting other potential issues in software tests, such as race conditions and deadlock, PolDet is re-implemented in JPF (PolDet@JPF) [35] to combine the capability of PolDet and JPF. PolDet@JPF captures and compares the states before and after the test execution with JPF's state serialization mechanism. Its implementation is only about 200 lines of code on top of JPF but can detect 26 polluter tests existing in 13 Java projects. This demonstrates JPF's versatility for rapid prototyping of various software tools in research.

5.2 Systematic Bit-Flip Fault Injection and Exploration

Computer hardware is susceptible to errors. Hardware defects or radiation can induce errors to the hardware, which can result in a memory bit being flipped. With the increasing complexity of computer hardware according to Moore's law [36], bit flips become more likely [37], making it important to improve the resiliency

```
public static void foo(@BitFlip int n) {
  System.out.println(n);
}
```

Fig. 4: An annotation triggering a bit-flip analysis for the method parameter

of software against hardware errors. However, these hardware errors are non-deterministic and hard to reproduce. To evaluate software's resiliency against bit flips, fault injection [38] can simulate the outcome of such events at the software level.

To support such fault injection, we use JPF to systematically inject and explore bit-flip faults in the user specified variables in Java programs. Specifically, the users can specify a list of variables and for each variable v_i specify k_i, the number of bits to flip in it. Considering that any k_i bits of v_i are possible to be flipped in real hardware faults, we let JPF execute the program in all possible ways to systematically evaluate programs' resiliency to bit-flip faults. For a simple example, consider the code in Figure 4. We want to know what happens if some error causes a bit to be flipped in the argument to method `foo`. Since `n` is of type `int`, our implementation explores all the 32 cases in which bit is flipped, so the expected output of calling `foo(0)` is `1 2 4...-2147483648`.

Our implementation provides a `BitFlipListener`, a JPF listener that monitors the list of user-specified variables and performs bit-wise fault injection before the relevant instructions execute. Specifically, we support injecting bit-flip faults to three kinds of variables, (1) static and instance fields, (2) method arguments, and (3) local variables, whose type can be any primitive data types. For the fields and local variables, the bit flips are injected when they are written by the programs. For the method arguments, the bit flips are injected when the method is invoked and the arguments are assigned. `BitFlipListener` registers a Choice Generator to inject all possible bit flips to the corresponding operand in the operand stack before the store/write instruction or the invoke instruction, depending on the variable type.

A user can specify the variables to flip in three ways: (1) calling `getBitFlip` API in the application code, (2) adding `@BitFlip` annotation to the variables, and (3) specifying in the command line arguments without changing the application code. For example, in Figure 4, we can (1) add `n=getBitFlip(n,1)` at the beginning of the method `foo`, (2) annotate `n` with `@BitFlip(1)` (where `(1)` can be omitted because $k = 1$ by default), or (3) specify bit-flip fault injection in method `foo`, parameter `n`, and $k = 1$ in the command line arguments.

Our implementation is based on JPF's `Verify.getInt`, which generates all possible integer values in a given range. The `BitFlipListener` parses the annotations and the command line arguments and adds the specified variables to a watch list.

A key challenge in the implementation is that JPF cannot register several choice generators at the same point of the application code. We resolve this issue by registering only one choice generator even when the number of bits to flip, k,

in a variable is $k > 1$. Specifically, we register only one `IntIntervalGenerator` that produces an integer m in range $[0, \binom{n}{k})$ where n is the number of bits of the variable, and then decode the integer using binomial coefficients to get the set of k in n bits to flip. The decoding process is as follows: Because $\binom{n-1}{k}$ out of $\binom{n}{k}$ combinations do not select the n^{th} bit, if $m > \binom{n-1}{k}$, we select the n^{th} bit, let $m' = m - \binom{n-1}{k}$ and $k' = k - 1$; otherwise, we do not select the n^{th} bit and let $m' = m, k' = k$. If we then let $n' = n - 1$, with the same process on n', m', k', we can decode out the set of k' in the remaining n' bits, and then recursively get all the k bits to flip.

We implemented a JPF regression test class that checks our injection engine in various scenarios and documents the basic usage. It verifies all bits are flipped exactly once in a variable using a global counter at JPF level. Besides, we used our tool to check the resiliency of Cyclic Redundancy Check (CRC) and International Standard Book Number (ISBN) algorithms against bit-flip faults. We confirmed that both algorithms can detect all one-bit flips, but cannot detect all two-bit flips, as expected. Our implementation has been included in JPF.[10]

6 Project Evolution and Evaluation

The evaluation of the validity of JPF is based on automated unit tests, which execute key features of JPF. JPF-level unit tests internally run small applications using JPF's analysis engine, thus effectively implementing system tests [39]. Our experimental evaluation is therefore based on these unit tests, which grew from 864 tests, for the code base supporting Java 8, to 1002 tests at the time of writing.

Figure 5 shows how the code base grew in the last five years, to accommodate for Java 11 functionality.[11] We sometimes observe sudden code size increases and drops in failing tests, which is usually because larger amounts of work had been merged into the Java 11 development branch at that time.

The first large decrease in failing tests was thanks to preliminary support for Java 11 string handling; the final large code growth was from incorporating patches and tests from the mainline development (for Java 8) into Java 11.

The graph shows that while we also had contributions at other times of the year, virtual summer internships supported by Google Summer of Code were a major factor in the contributions, as it was possible to carry out development that was not directly tied to a short-term research goal.

7 Conclusions

This paper has provided a detailed account of JPF's current architecture. JPF's modular design separates bytecode execution from code that models key library

[10] `https://github.com/javapathfinder/jpf-core/pull/295`

[11] JPF used to be handled as a Mercurial repository, and older versions of Mercurial were too slow to handle a large project history, so the project history was purged in 2018 by importing the entire code base as an initial commit in a new git repository.

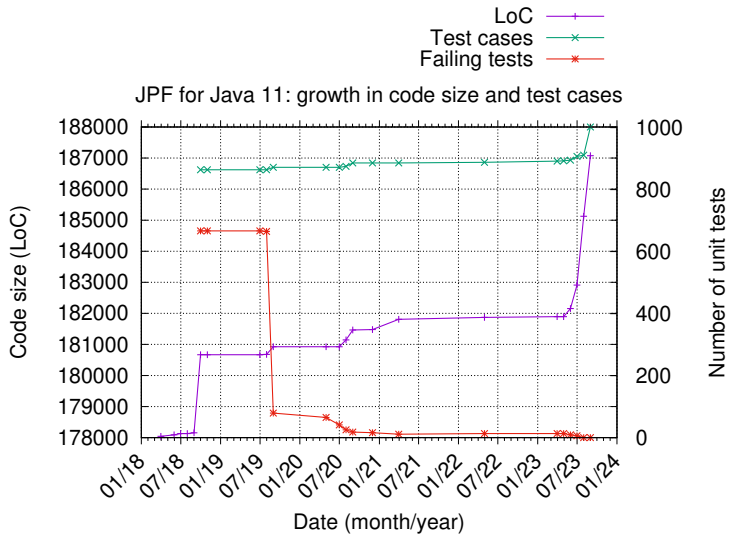

Fig. 5: Evolution of Java 11 development from 2018–2023.

functions and code that interfaces with the underlying run-time environment. Thanks to a very extensible design, JPF's functionality can be modified from concrete to symbolic execution or from a single application to multiple processes, which can even be networked.

Most of the development effort of the last five years was focused on supporting Java 11. Its major changes required corresponding adaptations in JPF: The new modular library system required extensions in the class loader; a different compilation of string expressions to bootstrap methods required support for them; and internal API changes brought both simplifications in the code base (because some model classes could be dropped) as well as complications, because new types of interfaces to internal VM data structures had to be supported.

Future work includes ensuring seamless Java 11 support for JPF's extensions and support for the next stable Java release (Java 17).

Acknowledgments

We would like to thank Alexander Kohan, Dan Smith, Gayan Weerakutti, Jeanderson Barros Candido, John Toman, Malte Mues, Nastaran Shafiei, Quoc-Sang Phan, Vaibhav Sharma, Wen Zhang, Willem Visser, Amgad Rady, and Yuvaraj Anbarasan for their contributions to supporting Java 11.

We would also like to thank the Google Summer of Code program for their support and Soha Hussein and other JPF organization administrators.

This work was partially supported also by the Czech Science Foundation project 23-06506S.

References

1. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
2. Cyrille Artho and Willem Visser. Java Pathfinder at SV-COMP 2019 (competition contribution). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 224–228, Cham, 2019. Springer International Publishing.
3. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
4. Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180, 2010.
5. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification, Java SE 11 Edition*. Oracle, 2018.
6. Matt Walker, Parssa Khazra, Anto Nanah Ji, Hongru Wang, and Franck van Breugel. jpf-logic: a framework for checking temporal logic properties of Java code. *ACM SIGSOFT Software Engineering Notes*, 48(1):32–36, 2023.
7. Klaus Havelund. Java PathFinder, a translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops Trento, Italy, July 5, 1999 Toulouse, France, September 21 and 24, 1999 Proceedings 5*, pages 152–152. Springer, 1999.
8. Martin Fowler and Matthew Foemmel. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`, 2006.
9. Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. JNuke: Efficient dynamic analysis for Java. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 462–465, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
10. Niels HM Aan de Brugh, Viet Yen Nguyen, and Theo C Ruys. Moonwalker: Verification of .NET programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pages 170–173. Springer, 2009.
11. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
12. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
13. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
14. Thomas Ball and Sriram K Rajamani. The SLAM toolkit. In *Proceedings of CAV 2001 (13th Conference on Computer Aided Verification)*, volume 2102, pages 260–264, 2000.
15. Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pages 389–391. Springer, 2014.

16. Les Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.
17. Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In Sorin Lerner and Atanas Rountev, editors, *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, pages 1–8. ACM, 2010.
18. Lukas Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tuma, and Zhengwei Qi. Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2012.
19. Andrej Čizmárik and Pavel Parízek. SharpDetect: Dynamic analysis framework for C#/.NET programs. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2020.
20. Kyle Storey, Eric Mercer, and Pavel Parizek. A sound dynamic partial order reduction engine for Java Pathfinder. *ACM SIGSOFT Software Engineering Notes*, 44(4):15–15, 2021.
21. Jeremy Manson, William Pugh, and Sarita V Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.
22. Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A Sanders. Java memory model-aware model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 18*, pages 220–236. Springer, 2012.
23. Nastaran Shafiei and Peter Mehlitz. Extending JPF to verify distributed systems. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.
24. Cyrille Artho, Kuniyasu Suzaki, Masami Hagiya, Watcharin Leungwattanakit, Richard Potter, Eric Platon, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. Using checkpointing and virtualization for fault injection. *IJNC*, 5(2):347–372, 2015.
25. Nastaran Shafiei and Franck van Breugel. Automatic handling of native methods in java pathfinder. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 97–100, 2014.
26. Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering*, 40(5):483–501, 2013.
27. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.
28. Alan Bateman, Alex Buckley, Jonathon Gibbons, and Mark Reinhold. JEP 261: The module system. `https://openjdk.org/jeps/261`, 2014.
29. Oracle. Class CallSite. `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/invoke/CallSite.html`, 2018.
30. Alex Buckley and Mark Reinhold. JEP 396: Strongly encapsulate JDK internals by default. `https://openjdk.org/jeps/396`, 2020.
31. Alex Buckley and Mark Reinhold. JEP 403: Strongly encapsulate JDK internals. `https://openjdk.org/jeps/403`, 2021.

32. Brent Christian and Xueming Shen. JEP 254: Compact strings. `https://openjdk.org/jeps/254`, 2014.
33. Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, 2019.
34. Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 223–233, New York, NY, USA, 2015. Association for Computing Machinery.
35. Pu Yi, Anjiang Wei, Wing Lam, Tao Xie, and Darko Marinov. Finding polluter tests using Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 46(3):37–41, 2021.
36. Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.
37. Laszlo B Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.
38. M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
39. The JPF Team. Writing JPF tests. `https://github.com/javapathfinder/jpf-core/wiki/Writing-JPF-tests`, 2023.