# Automatic Handling of Native Methods in Java PathFinder

Nastaran Shafiei*† and Franck van Breugel*‡
*DisCoVeri Group, Department of Computer Science and Engineering
York University, 4700 Keele Street, Toronto, M3J 1P3, Canada
†NASA Ames Research Center
Moffett Field, CA 94035, USA
‡Department of Computer Science
University of Oxford, Parks Road, Oxford, OX1 3QD, UK

***Abstract*—Java is currently one of the most popular programming languages and Java PathFinder (JPF) is the most popular model checker for Java. Despite its popularity, JPF cannot check a simple property such as the absence of uncaught exceptions for any realistic Java application without a nontrivial amount of work done by its user. This seriously hinders the uptake of JPF by developers. In this paper, we focus on the factor that causes the most trouble: native methods. Since JPF can only handle Java bytecode instructions, calls to native methods need to be intercepted and handled differently. Although JPF provides support for some native methods, the vast majority of native methods is not handled and, hence, JPF crashes on almost any realistic Java application unless its user handles the native methods that cause it to crash. This amounts to the user modelling the behaviour of the native methods in Java. This is generally a time consuming, tedious and error prone task. In this paper we present an extension of JPF that alleviates the user from this burden by automating this task. We showcase our tool by applying it to a variety of simple yet realistic Java applications that JPF, without our extension, cannot handle. Our extension of JPF clears a major hurdle for JPF towards becoming part of every Java developer's toolkit.**

## I. Introduction

According to the TIOBE programming community index[1], C and *Java* are currently the most popular programming languages. Although there are several *model checkers* for Java, including Bandera [6], Bogor [14] and an extension of SAL [13] to name a few, *Java PathFinder* (JPF) is the most popular one. Its popularity is reflected by several statistics. For example, the conference paper [18] and its extended journal version [17] have been cited more than 1200 times according to Google scholar, making it the most cited work on a Java model checker.

Most of the Java applications that come with the JPF distribution[2] are classic concurrency examples such as a solution to the dining philosophers problem and a concurrent implementation of a bounded buffer. Although these examples demonstrate that JPF is a powerful tool to find intricate bugs such as deadlocks, these examples cannot be considered realistic. They lack ingredients today's Java applications contain such as a graphical user interface, interaction with a database, and communication over the Internet. If we apply JPF to such applications, it crashes. In most cases, the culprit is a call to a *native method*, that is, a method written in a language different from Java, such as C and C++. Since JPF can only handle Java bytecode instructions, it crashes as soon as it encounters a call to a native method unless the call is intercepted by JPF and handled differently.

JPF provides two different mechanisms, which can be combined, to handle calls to native methods. We shall discuss these in more detail below. However, both approaches are labour intensive, tedious, and error prone. Nevertheless, both approaches have been successfully applied to handle a large variety of native methods. We shall discuss some of them in Section V on related work. Unfortunately, the vast majority of native methods is not yet handled by JPF. Moreover, Java applications may contain user defined native methods. As a consequence, JPF crashes when checking a Java application that contains a call to such a native method.

Currently, the only option to the Java developer is to exploit the mechanisms provided by JPF. However, as we already mentioned, this will take a lot of time and, hence, will make JPF of less interest to most developers. In this paper, we shall present an extension of JPF, which we call *jpf-nhandler*[3]. It *automates* the use of both mechanisms to handle calls to native methods. The only thing left to the developer is a single simple configuration file.

As is well known, there is no such thing as a free lunch. It will come as no surprise that replacing a time consuming, tedious and error prone task with the task of creating a single simple configuration file comes at a price. In Section IV we shall discuss the limitations of our approach. To mitigate the

[1] www.tiobe.com
[2] babelfish.arc.nasa.gov/trac/jpf
[3] bitbucket.org/nastaran/jpf-nhandler

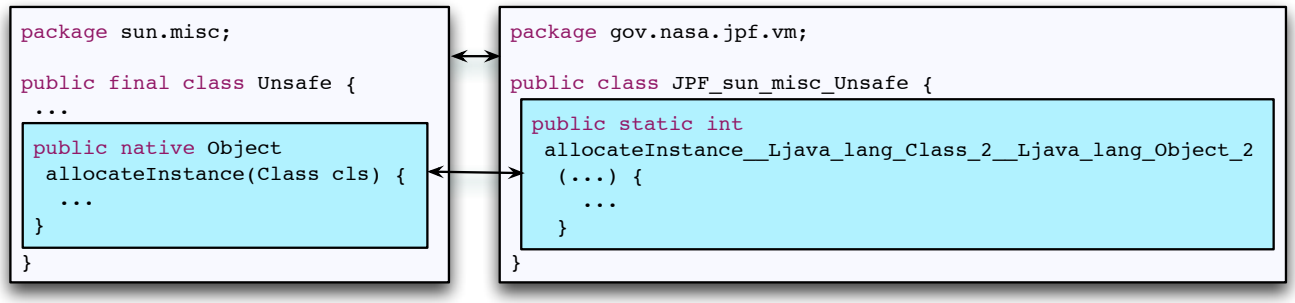


Fig. 1. Naming pattern used by JPF's model Java interface.

impact of these limitations, our extension can generate Java classes that can be modified by the user. In most cases the user only has to create a configuration file, as we shall demonstrate with a wide range of examples in Section II. In those cases where the user needs to modify the generated Java classes, these classes already contain a lot of useful code that the user can exploit, rather than having to start from scratch. Therefore, even in these cases, our extension is of help to the developer.

Before we outline our approach to automatically handle calls to native methods in JPF, we first discuss how a Java virtual machine (JVM) handles calls to native methods. Most JVMs include a *native method interface* (JNI) [11]. This interface takes care of calls to native methods by transferring the execution from the JVM to the operating system (OS). For example, the method `allocateInstance` of the class `Unsafe`, which is part of the package `sun.misc`, is native. Let `unsafe` be an `Unsafe` object and let `clazz` be a `Class` object. Then the call `unsafe.allocateInstance(clazz)` allocates an instance of the class `clazz` but does not run any constructor and returns the created uninitialized object. Although most developers never use this native method, we use it as our running example in this introduction because it is non-static, it takes an object as an argument and it returns an object. These three characteristics allow us to illustrate all aspects of our approach. Once the JVM reaches the call `unsafe.allocateInstance(clazz)`, it transfers the execution to the OS. JNI makes the objects `unsafe` and `clazz` accessible to the native code. Once the native code has been executed by the OS, JNI makes the result accessible to the JVM and the execution is transferred back to the JVM. As we shall see, our approach to handle native methods shows several similarities to the way JNI handles them.

Next, we introduce those readers unfamiliar with JPF to those features that are key to our work. JPF is itself a JVM. In contrast to an ordinary JVM, which simply executes a Java application, JPF systematically checks all potential executions of a Java application. Since JPF is implemented in Java, it runs on top of an ordinary JVM, which we will call the *host JVM*. The latter runs on top of the OS.

Let us now dive into more details by discussing the two mechanisms that JPF provides to handle calls to native methods and how we exploit them in jpf-nhandler. Whenever JPF encounters a call to a native method, it intercepts that call and tries to handle it differently. If JPF supports the native method, then it will exploit either of the mechanisms to deal with the native method. Otherwise, it will crash.

JPF's *model classes* provide a way to handle calls to native methods. These classes model the behaviour of actual classes. Often the model classes abstract from particular details of the actual classes. These model classes are part of JPF. A model class has the same name as the actual class it models. JPF contains a model class named `sun.misc.Unsafe`. Whenever JPF encounters a class for which it has a model class, it model checks the model class instead of the actual class. Hence, when JPF encounters the call `unsafe.allocateInstance(clazz)` it model checks the `allocateInstance` method of the model class `Unsafe`. Modelling a class can be a daunting task, especially if only the bytecode of the class is available and that code has been obfuscated. In that case, the decompiled Java code contains no comments and lacks descriptive names. Also, neither the source code nor a specification of the native code may be available, making it very difficult to model its behaviour in Java.

JPF's *model Java interface* (MJI) can be used to transfer the execution from JPF to the host JVM. The so called *native peer* classes play a key role in MJI. JPF uses a specific name pattern to associate the native peer classes and their methods with the corresponding classes and methods. For example, the native peer class associated with `sun.misc.Unsafe` is named `JPF_sun_misc_Unsafe` (see Figure 1). Whenever JPF gets to a call associated with a native peer method, it delegates the call to the host JVM. Hence, the native call is

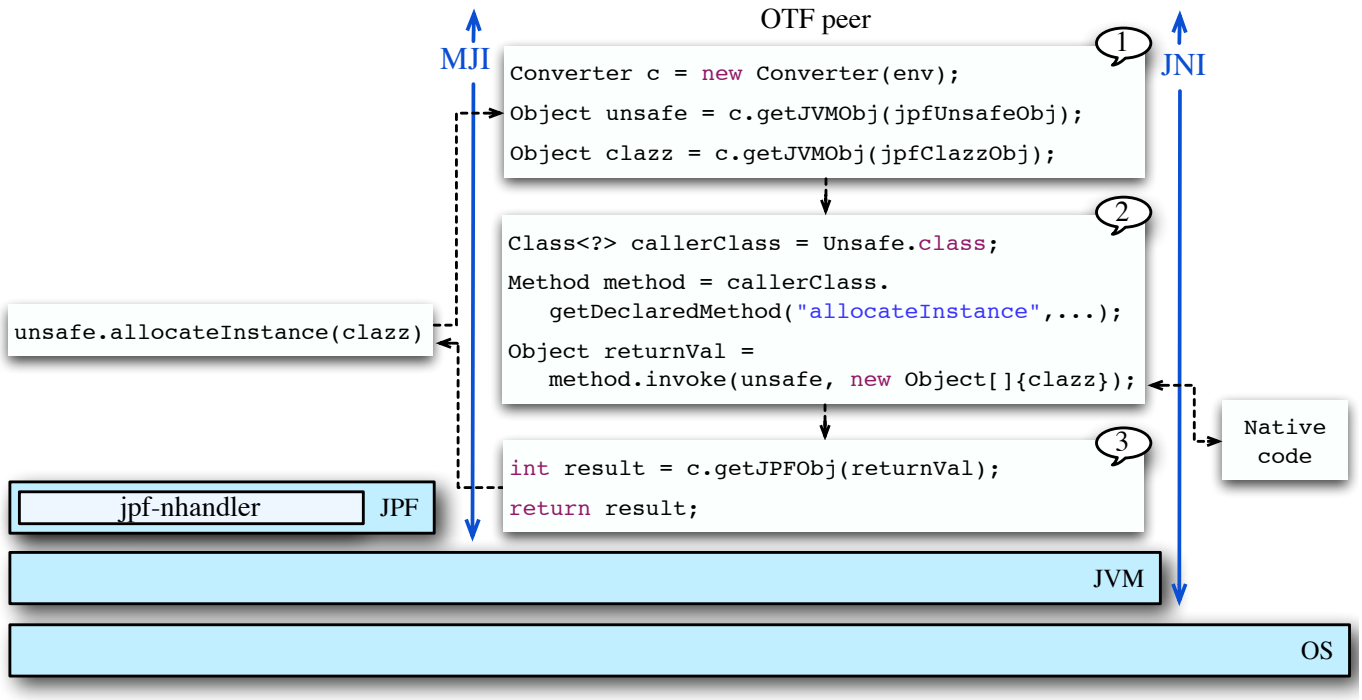


Fig. 2. Handling of `unsafe.allocateInstance(clazz)` by jpf-nhandler.

not model checked, which is impossible since JPF can only handle Java bytecode, but executed on the host JVM. Great care has to be taken when developing a native peer class. For example, since classes and objects are represented differently in JPF than in an ordinary JVM, in a native peer class one often has to translate from the one representation to the other and back.

JPF and its extensions currently include a few hundred model classes and native peer classes. However, there are many more classes with native methods that are not handled. Rather than putting the heavy burden of implementing model classes or native peer classes on the developer, we provide the developer with our extension jpf-nhandler that deals with calls to native methods automatically. Next, we will briefly discuss our extension.

The approach taken by jpf-nhandler mainly relies on MJI and native peer classes. Whenever JPF encounters a call to a native method, jpf-nhandler automatically intercepts the call and delegates its execution from JPF to the host JVM. It creates bytecode for native peer classes on-the-fly (referred to as OTF peer classes from now on) using the bytecode engineering library (BCEL)[4]. To delegate the execution of a native method to the host JVM, jpf-nhandler creates an OTF peer class (if it does not exist yet) and adds a native peer method to the OTF peer class (if it does not exist yet). This native peer method implements the following three main steps. Let us consider again the call `unsafe.allocateInstance(clazz)`.

1) First, the JPF representation of the objects `unsafe` and `clazz` are transformed to corresponding JVM objects.
2) Then the execution is delegated to the host JVM by calling the original native method `allocateInstance` on the JVM representation of `unsafe` with the JVM representation of `clazz` as its argument.
3) Finally, the result of the method call, which is an object in our example, is transformed from its JVM representation to its JPF representation.

Note that jpf-nhandler has to transform classes and objects from JPF to the host JVM and back. Our class `Converter` accomplishes that. Its method `getJVMObj` transforms JPF objects to JVM objects and `getJPFObj` transforms objects in the other direction. It also contains similar methods for converting classes and arrays.

Figure 2 shows how the native method call `unsafe.allocateInstance(clazz)` is handled by jpf-nhandler. The left column is executed by JPF. The middle column is executed by the host JVM. It includes the code generated on-the-fly by jpf-nhandler as the body of the `allocateInstance` native peer method. In part (1) of the code, the JVM representations of the objects `unsafe` and `clazz` are generated from their JPF representations using the `Converter` class. In part (2), using Java reflection, the method `allocateInstance` is invoked on the `Unsafe` object with the `Class` object as its argument. Since

[4]commons.apache.org/bcel

`allocateInstance` is a native method, its execution is delegated from the host JVM to the OS, using JNI. The right column is executed by the OS. Finally, in part (3), the result from `allocateInstance` is transformed to its JPF representation, which is returned to JPF. This part also includes updating the objects `unsafe` and `clazz` in JPF from their JVM representation, since these objects may have changed due to side effects of the native method executed in part (2).

Our extension jpf-nhandler has been successfully applied to a variety of simple yet realistic Java applications. In Section II we present several examples which confirm that jpf-nhandler can be used to model check Java applications that contain calls to native methods. For all these examples, JPF without our extension crashes. The examples are all small[5], but do contain ingredients found in today's applications such as a graphical user interface, interaction with a database, and communication over the Internet. All but one example have less than 100 lines of code. The one example with a few hundred lines of code is a Java implementation of the game *Hamurabi*. This is one of the very first computer games. It was implemented in BASIC by Ahl in the seventies [1, page 128] and we ported the BASIC code to Java. Since the handling of native methods is independent of the type of property being checked, in our examples we used the standard configuration of JPF which checks for uncaught exceptions. As an aside, our initial implementation of Hamurabi contained a bug which caused an exception. This bug was found by JPF in combination with our extension jpf-nhandler within seconds.

## II. Application of jpf-nhandler

Below, we discuss a variety of simple Java applications. The majority of these applications are part of the jpf-nhandler distribution. None of these applications can be model checked by JPF without our extension jpf-nhandler. We describe how to configure jpf-nhandler so that they can be model checked successfully.

### A. Allocating an Object

We start with the example that we presented in the introduction. In the main method of our application, we use

```
object = unsafe.allocateInstance(clazz);
```

which is a call to a native method. Since JPF does not handle this native method, it crashes when it verifies our application. To configure jpf-nhandler, we modify the application properties file by adding the following to it.

```
@using=jpf-nhandler
nhandler.delegateUnhandledNative=true
```

The first line specifies that we are using the extension jpf-nhandler. By setting the property `nhandler.delegateUnhandledNative` to true, jpf-nhandler deals with any call to a native method that is not yet handled by JPF. As a consequence, it deals with the `allocateInstance` method and verifies the application successfully.

### B. Manipulating Java Archive Files

As is well known, jar files play a central role in the development of Java applications. The package `java.util.jar` contains classes to manipulate jar files. Our application uses the classes `JarFile` and `JarEntry` to print the names of the files in a given jar file. The application gives rise to several calls to native methods which cause JPF to crash. To enable and configure jpf-nhandler, we modify the application properties file by adding the following to it.

```
@using=jpf-nhandler
nhandler.resetVMState=false
nhandler.spec.delegate=\
  java.util.zip.ZipFile.*,\
  java.util.jar.JarFile.*
```

Our class `Converter`, which we already mentioned in the introduction and will discuss in more detail in Section III-C, contains two maps: one from JPF objects to JVM objects and the other from JVM objects to JPF objects. By default, both maps are cleared after a call to a native method has been handled by jpf-nhandler. By setting the property `nhandler.resetVMState` to false, those maps are not cleared. As a consequence, jpf-nhandler does not recreate JVM objects for JPF objects it has already transformed earlier. We will come back to this later in the paper. The last three lines specify which methods, constructors, and static initializers should be delegated from JPF to the host JVM. In this particular case, all methods, constructors, and static initializers of the classes `ZipFile` and `JarFile` should be delegated. In this case, we also delegate methods that are not native. Delegating all methods, constructors, and static initializers of a class such as `JarFile` simplifies matters since all operations on a `JarFile` object are executed on the host JVM. We will come back to this in Section IV.

### C. Communicating over a Network

Many applications communicate over a network. For example, sockets provide network communication. In Java, the class `Socket` of the package `java.net` implements sockets. We implemented a client application and a server application. Both contain calls to native methods that are not handled by JPF. We run the server application on one machine and model check

[5]The difficulty of model checking a Java application is not proportional to the number of lines of code of the application. There are Java applications consisting of less than a dozen lines of code for which JPF either crashes, runs out of memory, or does not complete its verification effort within a day.

```
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuInitNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuDeviceGetNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuCtxCreateNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuModuleLoadNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuModuleGetFunctionNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuMemAllocNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuMemcpyHtoDNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuLaunchKernelNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuCtxSynchronizeNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuMemcpyDtoHNative
* DELEGATING Unhandled Native -> jcuda.driver.JCudaDriver.cuMemFreeNative
At index 1 found 1.07374182E9 but expected 2.0
Test FAILED
```

Fig. 3. Part of JPF's output when model checking `JCudaVectorAdd`.

the client application on another machine. To the application properties file, we add the following.

```
@using=jpf-nhandler
nhandler.delegateUnhandledNative=true
nhandler.spec.filter=\
  java.io.FileDescriptor.close0
```

The method `close0` is a native method in JPF's model class `java.io.FileDescriptor`. Since this method is not part of the original `FileDescriptor` class, we have to make sure that jpf-nhandler does not delegate a call to this native method to the host JVM. This is done by means of the property `nhandler.spec.filter`.

Red Hat's JGroups[6] provides a framework for reliable multicast communication. JGroups is used in hundreds of Java projects. In our simple example, two applications communicate by using a `org.jgroups.JChannel`. One application sends a message which is received by the other one. We execute the receiver application on one machine while we model check the sender application on another machine. We refrain from providing the configuration details since they are similar to the previous example.

### *D. Exploiting Graphics Processing Units*

Graphics processing units (GPUs) provide generally a much simpler instruction set than ordinary CPUs, but they are generally much faster due to the much larger number of cores. Therefore, it sometimes makes sense to run parts of an application on the GPU and the remainder on the CPU. NVIDIA's compute unified device architecture (CUDA) contains an environment to program their GPUs. The package *jcuda*[7] enables Java applications to run CUDA code on the GPU. We consider the application `JCudaVectorAdd` which is part of the jcuda distribution. The application creates two 100,000 element arrays, runs some CUDA code on the GPU to add the arrays, and finally checks the result. It will come as no surprise that several native methods are needed to bridge the gap between the Java code and the CUDA code. To enable jpf-nhandler to model check the application, we add the following to the application properties file.

```
@using=jpf-nhandler
classpath=.../jcuda-0.5.0a.jar
native_classpath=.../jcuda-0.5.0a.jar
nhandler.delegateUnhandledNative=true
nhandler.resetVMState=false
nhandler.spec.filter=\
  java.lang.Class.getByteArray...Stream
```

The `classpath` includes paths to classes and libraries that are model checked by JPF and the `native_classpath` includes paths to classes and libraries that are executed by the host JVM. Since jpf-nhandler switches between JPF the host JVM, we need to make the jcuda library accessible to both.

When we model check the application `JCudaVectorAdd` with JPF and our extension jpf-nhandler, we obtain some output, part of which is given in Figure 3. Our extension informs the developer which native methods are handled by jpf-nhandler. Note that the test failed. This failure is not caused by jpf-nhandler but by a bug in JPF. It turns out that arrays of floats are not converted correctly within JPF itself. After fixing the bug, the test passes.

### *E. Querying a Database*

Many applications interact with a database. Apache Derby[8] is a relational database implemented entirely in Java. It is one of the popular choices for database applications written in Java. In our application, we connect to a database, create a new

[6] www.jgroups.org
[7] jcuda.org
[8] db.apache.org/derby

table, insert two records into the table, and finally close the connection to the database. Despite the fact that the database is implemented in Java, our application gives rise to several calls to native methods that are not handled by JPF. However, jpf-nhandler is able to model check the database application successfully.

### F. Scraping the Web

We have developed a web scraper which simply reads the HTML of the conference web page. Of course, one could subsequently try to extract interesting information from the read data such as, for example, the paper submission deadline. However, since such string manipulation can be easily handled by JPF, we refrain from doing that. Our application contains several calls to native methods that are not handled by JPF. For example, we create a `URL` object which gives rise to a call to the native method `doPrivileged` of the class `java.security.AccessController`. By configuring jpf-nhandler appropriately, we can successfully model check this application as well.

### G. Invoking Web Services

Google's translate web service[9] translates phrases between natural languages. In our application, we use this web service to translate a phrase from English to French. JPF is not able to model check this application. When we configure jpf-nhandler to delegate the method `retrieveJSON` of the class `com.google.api.GoogleAPI` (this method forms an HTTP request, sends it, and returns the request result as an instance of `org.json.JSONObject`) to the host JVM, the application can be model checked successfully.

### H. Playing Games

The Java code of computer games is full of calls to native methods since games extensively use graphics and sound, often communicate over the Internet, etcetera. In [15] the authors describe how they have successfully used jpf-nhandler to model check two simple computer games. The one game is Hamurabi which we already discussed in the introduction. The other game is a graphics based version of rock-paper-scissors.

### I. Solving Ordinary Differential Equations

The Apache Commons mathematics library[10] provides a large variety of packages related to mathematics and statistics. In our application, we use the library to numerically solve ordinary differential equations. Without jpf-nhandler, JPF crashes due to unsupported native methods in the class `java.lang.StrictMath`, such as `log`. Using jpf-nhandler, JPF model checks our application successfully where jpf-nhandler is configured to delegate all unhandled calls to native methods.

[9]code.google.com/p/google-api-translate-java
[10]commons.apache.org/math

## III. Implementation of JPF-nhandler

Our extension jpf-nhandler has been implemented in such a way that no changes to the core of JPF are required. It consists of three main components. Before discussing those components in some detail, let us first paint the overall picture of jpf-nhandler. The first component is a listener which is notified by JPF whenever a class is loaded. The second component generates the bytecode and Java code of the OTF peer classes. The third and final component translates objects and classes from JPF to the host JVM and back.

### A. Listener

The listener is implemented in our class `ExecutionForwarder`, which is part of the package `gov.nasa.jpf.vm`. Before going through the details of this class, let us first discuss how JPF captures native methods and how it executes their corresponding native peer methods on the host JVM.

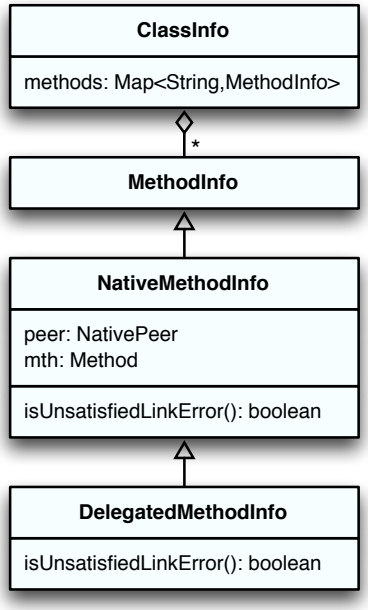


Fig. 4. A UML diagram of the classes representing classes and methods in JPF and jpf-nhandler.

Classes are represented in JPF by instances of `gov.nasa.jpf.vm.ClassInfo`. The `ClassInfo` class includes a map, called `methods`, which keeps track of all the methods declared in the class. The keys of this map are string representations of the method signatures. The values of this map are `gov.nasa.jpf.vm.MethodInfo` objects. The class `gov.nasa.jpf.vm.NativeMethodInfo` is a subclass of `MethodInfo`. It represents native methods. The `NativeMethodInfo` class declares a field of type of `gov.nasa.jpf.vm.NativePeer`, called `peer`, which

contains the corresponding native peer class. It also declares the field `mth` of type of `java.lang.reflect.Method`, which contains the corresponding native peer method. Whenever JPF encounters a call to a method represented by a `NativeMethodInfo` object, instead of model checking it, JPF delegates its execution to the host JVM. JPF uses Java reflection to invoke the method `mth` of the class `peer` on the host JVM. After the method `mth` has been executed by the host JVM, JPF resumes its model checking effort.

As we already mentioned earlier, the class `ExecutionForwarder` is a listener. It receives notifications from JPF whenever a class is loaded. By the time `ExecutionForwarder` receives the notification, JPF has already created a `ClassInfo` object that represents the loaded class and initialized its `methods` field with a `MethodInfo` object for each method of the loaded class. Once the `ExecutionForwarder` receives a notification, it goes through the collection of `MethodInfo` objects of the loaded class. For each native method, it checks if the method is already handled by JPF. That is, it checks if JPF has associated a native peer method with it. If the native method is not handled by JPF, the `ExecutionForwarder` replaces its corresponding `MethodInfo` object with a `gov.nasa.jpf.vm.DelegatedMethodInfo` object. The class `DelegatedMethodInfo` is a subclass of `NativeMethodInfo` (see Figure 4 for the relationships between these classes). This class is part of jpf-nhandler.

The class `NativeMethodInfo` contains the method `isUnsatisfiedLinkError`. This method checks whether a native peer method is associated with a `NativeMethodInfo` object, that is, it checks whether its `mth` field is null. JPF executes this method before it attempts to invoke the native peer method represented by `mth`. In our `DelegatedMethodInfo` we override the `isUnsatisfiedLinkError` method. Consider, for example, the unhandled native method `allocateInstance` from the introduction. The first time the method `isUnsatisfiedLinkError` is invoked on `allocateInstance`'s `DelegatedMethodInfo` object, jpf-nhandler creates the corresponding OTF peer class and method (if they do not already exist) and initializes the fields `peer` and `mth` of the `DelegatedMethodInfo` object to the OTF peer class and method, respectively. As a consequence, whenever JPF encounters a call to `allocateInstance`, it delegates the execution of its OTF peer method `mth` of its OTF peer class `peer` to the host JVM.

### *B. On-the-fly Native Peer Classes Generator*

The generation of the OTF peer classes is implemented in the package `nhandler.peerGen`. By default, jpf-nhandler generates both bytecode and Java code for the OTF peer classes. To only generate bytecode, the user can set the property `nhandler.genSource` to false in the configuration file.

The OTF peer classes and methods follow the same naming pattern as the JPF native peer classes and methods, with the exception that the name of the OTF peer classes is prefixed by `OTF_`. The files containing the bytecode and Java code generated by jpf-nhandler can be found in the `onthefly` directory of jpf-nhandler. The user can configure jpf-nhandler to reuse the existing OTF peer classes for future runs of JPF by setting the property `nhandler.clean` to false in the configuration file. Since generating the bytecode and Java code is expensive, using this feature can speed up jpf-nhandler considerably.

A key class in our package `nhandler.peerGen` is `PeerClassGen`. As we already mentioned in Section III-A, the method `isUnsatisfiedLinkError` of the class `DelegatedMethodInfo` creates OTF peer classes. In particular, it creates an instance of the class `PeerClassGen`. For every OTF peer class, there exists one instance of `PeerClassGen`. This object is created the first time that jpf-nhandler attempts to handle a method of the class. Before generating the OTF peer class, the `PeerClassGen` object checks whether the `onthefly` directory of jpf-nhandler already contains the OTF peer class. If so, it loads the class. Otherwise, it generates the OTF peer class.

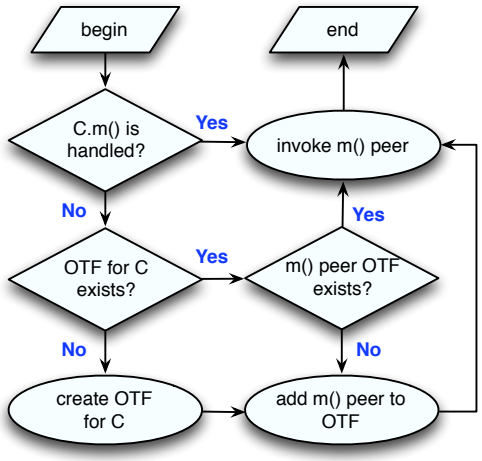


Fig. 5. A UML diagram depicting how `PeerClassGen` decides to handle a call to the native method `m` of the class `C`.

To extend an OTF peer class with a native peer method, jpf-nhandler invokes the method `createMethod(NativeMethodInfo)` on the appropriate `PeerClassGen`. This method first checks if the OTF peer class already contains the native peer method. If not, it adds it to the OTF peer class. The diagram in Figure 5 shows how `PeerClassGen` decides to handle a

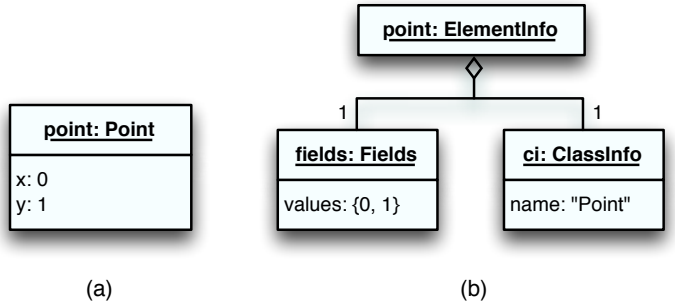


Fig. 6. UML diagrams of a Point object represented in a JVM (a) and JPF (b).

native method.

As we have already seen in the introduction, to handle the call `unsafe.allocateInstance(clazz)`, the body of the OTF native peer method includes the following three main steps. The first step includes representing all the relevant JPF objects and classes in the host JVM. In our example, these are the JPF objects `unsafe` and `clazz` and their JPF classes `Unsafe` and `Class`. In the second step, using Java reflection, the method `allocateInstance` is invoked on the JVM representation of `unsafe` with as argument the JVM representation of `clazz`. Finally, in the third step, the return value of the invocation is translated to a JPF object. This step also includes updating the JPF objects `unsafe` and `clazz` and the JPF classes `Unsafe` and `Class`.

One may wonder why we did not generate the OTF peer classes before running JPF. In that case, one would have to statically analyze the Java application to determine which native methods it may call. This may include native methods that will not be encountered during the model checking of the application and, hence, we may generate many more OTF peer classes and methods than are actually needed. Note also that we can configure jpf-nhandler to reuse existing OTF peer classes.

It turns out that we do not need to generate the OTF peer classes at all. We can implement the invocation of the native method using Java reflection. However, the major advantage of generating the OTF peer classes only reveals itself when jpf-nhandler fails to automatically handle all calls to native methods. In that case, the developer can modify the generated Java code of the OTF peer classes, rather than having to start from scratch.

### C. Converter

As we already mentioned in the introduction, the way that objects and classes are represented in JPF is different from the way they are represented by the host JVM. As discussed in Section III-A, JPF uses instances of the class `ClassInfo` to represent classes. To represent objects it uses instances of the class `gov.nasa.jpf.vm.ElementInfo`. Figure 6 contains an example, contrasting the different representations for a simple object.

Since jpf-nhandler interacts with the host JVM, as shown in Figure 2, we need to convert objects and classes from JPF to the host JVM and back. Such a conversion is implemented in the package `nhandler.conversion`. The earlier mentioned class `Converter` is part of this package.

The class `Converter` contains methods that convert a JPF object or a JPF class to a corresponding JVM object or JVM class. The class is used in the OTF peer methods. Such a method starts by creating a `Converter` object. This object is subsequently used to convert JPF objects and classes to their JVM counterparts. Furthermore, just before the OTF native peer method returns, the `Converter` object is used to convert the result of the call back to JPF, and also to update some of the objects and classes that may have changed as a result of the call to the native method.

## IV. Limitations of jpf-nhandler

As we have shown in Section II, jpf-nhandler is applicable to a large variety of applications. However, there are some limitations to our extension. We will discuss them below. How we plan to address some of these limitations is discussed in the concluding section of this paper.

Native code can modify arbitrary objects and classes through JNI. Currently, we only reflect in JPF the changes made by the native code to some objects and classes. For example, consider again the call `unsafe.allocateInstance(clazz)`. Only changes made by the method `allocateInstance` to the objects `unsafe` and `clazz` and the classes `Unsafe` and `Class` are reflected in JPF. However, if the method `allocateInstance` were to change any other objects or classes, then their JPF representations and JVM representations would be out of sync. As a consequence, in such a case JPF could for example incorrectly report an uncaught exception. In our case studies, we have not

encountered such a scenario. If such a case were to arise, the developer could modify the Java code of the generated OTF peer class to reflect the changes.

Delegation of a method to the host JVM amounts to the assumption that its execution is atomic. However, this need not be the case. For example, consider an application consisting of two threads that share an integer variable `x` which is initialized to zero. The one thread simply consists of the statement `assert x % 2 == 0`. The other thread calls a method whose body consists of `x++; x++;`. JPF detects an uncaught exception, since there exists an interleaving of the two threads in which the assertion is not true. However, if we delegate the method to the host JVM, the method is assumed to be atomic and JPF does not consider the troublesome interleaving. Hence, one has to be careful if one uses jpf-nhandler to delegate non-native methods.

Since JPF considers all potential executions of an application, it may delegate a method multiple times. This may have undesirable consequences. Consider, for example, an application that prompts the user for the amount to donate to a charity. After the user has entered the amount (A), the application starts two threads. The one thread calls a method that debits the user's credit card (B), whereas the other thread calls a method that interacts with a database to extract information about the user (C). Now assume that both methods are delegated to the host JVM. In that case, JPF checks two interleavings: ABC and ACB. Hence, B is executed twice and, therefore, the user's credit card is debited twice.[11] We will come back to this limitation in the concluding section.

As we have seen in Section II-C and II-D, some of JPF's model classes are incompatible with the classes they model. In these two cases, we could configure jpf-nhandler appropriately so that we could still model check the application. However, we have also encountered situations where we had to make changes to JPF's model classes. Also this limitation will be discussed in the conclusion.

Recall that by setting the property `resetVMState` to false, the maps storing JPF and JVM objects are not cleared after handling a native method. However, if JPF changes an object that is stored in the maps, then the corresponding JVM object is not updated accordingly. In this way, the JPF representation of an object and its JVM representation may get out of sync. We will also come back to this limitation in the conclusion.

We cannot apply jpf-nhandler to certain classes such as `java.lang.Thread` as it would compromise the consistency of JPF. Luckily, we never need to apply jpf-nhandler to any of these classes since JPF already provides model classes and native peer classes for them.

[11] Obviously, this example is a bit contrived. When model checking such an application, a credit card should never be debited.

## V. Related Work

The work most closely related to ours is that of d'Amorim et al. [8]. Although their extension of JPF also model checks some parts of the code and executes the other parts of the code, their objective is not handling native methods but reducing the execution time of JPF. Despite that they have a different objective, their approach shares several ingredients with ours. First of all, they also translate JPF objects to JVM objects and back. However, their translations have several limitations from which ours do not suffer. For example, their translation from JPF objects to JVM objects handles neither arrays nor instances of classes without a default constructor. In our case studies, we have to handle both. Secondly, they also use reflection to invoke methods on the host JVM. Whereas they only handle methods, we also deal with several other elements of Java such as constructors and static initializers. For our examples presented in Section II it is essential to handle those other Java elements as well.

In the remainder of this section, we shall discuss some of the other extensions of JPF that also deal with native methods. Gligoric and Majumdar [10] have developed DPF, an extension of JPF to model check database applications. They consider both in-memory databases and on-disk databases. For the former, they reimplemented the native methods used by the database H2[12]. For the latter, they intercept all native method calls and those method calls that access the database. All these methods were reimplemented as well. In Section II, we already mentioned that jpf-nhandler can deal with a simple database application without having to reimplement any Java class.

In [3], Barlas and Bultan introduce a framework to model check distributed Java applications called *NetStub*[13]. The framework consists of several model classes that model those parts of the Java standard library related to network communication and containing native methods. Each component of the distributed application is represented by a thread so that the whole distributed application runs in a single JVM and, hence, can be verified by JPF.

Artho et al. [2] developed the JPF extension *jpf-net-iocache*[14] It provides a different approach to model check distributed Java applications. Only one component of the distributed application is model checked, the other ones are simply executed. Clearly, this approach is similar in flavour to jpf-nhandler. Since JPF attempts to systematically check all potential executions of that one component, one has to be careful to prevent communications between that component and the other components from being repeated. To address that problem, Artho et al. introduce a cache that keeps track of those communications. As in the work of Barlas and Bultan,

[12] h2database.com

[13] www.cs.ucsb.edu/~bultan/netstub

[14] babelfish.arc.nasa.gov/trac/jpf/wiki/projects/net-iocache

those classes that contain native calls are reimplemented as model classes. As we have seen in Section II, jpf-nhandler can also deal with simple distributed applications fully automatically without the need to reimplement any class.

JPF's extension *jpf-awt*[15] [12] provides a framework to model check graphical user interfaces. In op. cit., Mehlitz et al. replace classes of the packages `java.awt` and `javax.swing`, which contain native methods, with model classes. In [15] the authors show that jpf-nhandler can model check simple computer games, including games with graphical user interfaces, without having to develop any model classes.

In [16], Umja and the first author of this paper discuss the extension *jpf-concurrent*[16]. In this extension, numerous classes of the package `java.util.concurrent` are modelled. Although their main aim is to improve the performance of JPF when model checking applications that use this package, they also handle several native methods. JPF itself also models some other classes of `java.util.concurrent`. Even if we disable all these model classes, our extension jpf-nhandler can model check Java applications that use classes of `java.util.concurrent`. For example, we have successfully model checked the Java implementation[17] [4] of the concurrent binary search tree of Ellen et al. [9].

## VI. Conclusion

Our extension jpf-nhandler of JPF automates the handling of native methods. It accomplishes this by automatically delegating the execution of the native method to the host JVM. Therefore, it automates the intertwining the model checking of Java code and the execution of native code. In a way, jpf-nhandler is similar to concolic execution.

As we have seen in Section II, jpf-nhandler can deal with a large variety of native methods. However, as we pointed out in Section IV, our extension has some limitations. In the remainder of this section we shall outline how we plan to address some of those limitations.

Assume that `list` is a `List` object consisting of 100,000 integers. When we use jpf-nhandler to delegate the execution of the call `list.get(0)` to the host JVM, we convert the `List` object `list` and its 100,000 elements from their JPF representation to the corresponding JVM representation. Obviously, this is very expensive. It is also clear that one does not need to convert all 100,000 elements of `list`. In [8], d'Amorim et al. propose a lazy translation that only converts those parts of the objects involved that are needed in the delegated method. This is achieved by code instrumentation of the Java code of the delegated method. Hence, it is only applicable to the delegation of non-native methods. We plan to incorporate lazy translation into jpf-nhandler. Once we consider code instrumentation of the Java code of the delegated method, we may also ensure that changes to objects and classes made by the delegated method are reflected in JPF.

To avoid that a call to a method is delegated more than once, we plan to use the technique developed by Artho et al. in [2]. We intend to cache the effects of a delegated method call. If we encounter the same call later in JPF's verification effort, then we simply reflect the cached effects in JPF, rather than delegating the method call again.

As we have seen in Section II, incompatibilities between some of JPF's model classes and the corresponding JVM classes limit the applicability of jpf-nhandler. The JPF extension *jpf-conformance-checker*[18] [5], developed by Ceccarello and the first author of this paper, checks whether a model class of JPF is compatible with its JVM counterpart. We plan to incorporate such a check in our `Converter` class to detect incompatibilities. In case such incompatibilities are detected, we would like to take care of them automatically, rather than leaving it to the developer to handle them in the configuration file.

When we set `resetVMState` to false and JPF modifies an object that is stored in the maps used for conversion, the JPF object and its JVM counterpart may get out of sync. This discrepancy can be addressed in different ways. For example, although expensive, one could update the JVM object whenever it is used. To offset the cost, one could use a lazy update strategy, similar to the above mentioned lazy translation scheme.

As we mentioned earlier, objects are represented in JPF by `ElementInfo` objects. These objects contain more information than their corresponding JVM counterparts. Hence, in the vocabulary of abstract interpretation [7], the JPF representations form the concrete domain and the JVM representations form the abstract domain. The conversion from JPF representations to JVM representations is the abstraction function, and the conversion in the opposite direction is the concretization function. We are interested to see whether we can transfer results from abstract interpretation to our setting.

*Acknowledgements*

We would like to thank Marcelo d'Amorim for providing us with the source code of his tool described in [8] and Vladimir Blagojevic for his help with JGroups. We are thankful to Jason Keltz for answering our numerous questions about system related matters. A special thanks to Peter Mehlitz for all his help and feedback. Finally, we thank NVIDIA for providing a GPU that we used in one of our case studies.

[15] babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-awt

[16] babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-concurrent

[17] bitbucket.org/discoveri/concurrent

[18] bitbucket.org/nastaran/jpf-conformance-checker

## REFERENCES


[1] David H. Ahl, editor. *101 BASIC Computer Games*. DEC, 1973.

[2] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, and Yoshinori Tanabe. Efficient model checking of networked applications. In Richard F. Paige and Bertrand Meyer, editors, *Proceedings of the 46th International Conference on Objects, Models, Components, Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 22–40, Zurich, Switzerland, June/July 2008. Springer-Verlag.

[3] Elliot D. Barlas and Tevfik Bultan. NetStub: a framework for verification of distributed Java applications. In R.E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 24–33, Atlanta, GA, USA, November 2007. ACM.

[4] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In Antonio F. Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Proceedings of the 15th International Conference on Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 207–221, Toulouse, France, December 2011. Springer-Verlag.

[5] Matteo Ceccarello and Nastaran Shafiei. Tools to generate and check consistency of model classes for Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

[6] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM.

[8] Marcelo d'Amorim, Ahmed Sobeih, and Darko Marinov. Optimized execution of deterministic blocks in Java PathFinder. In Zhiming Liu and Jifeng He, editors, *Proceedings of the 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 549–567, Macao, China, November 2006. Springer-Verlag.

[9] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing*, pages 131–140, Zurich, Switzerland, July 2010. ACM.

[10] Milos Gligoric and Rupak Majumdar. Model checking database applications. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 549–564, Rome, Italy, March 2013. Springer-Verlag.

[11] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[12] Peter Mehlitz, Oksana Tkachuk, and Mateusz Ujma. JPF-AWT: Model checking GUI applications. In Perry Alexander, Corina S. Păsăreanu, and John G. Hosking, editors, *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 584–587, Lawrence, KS, USA, November 2011. IEEE.

[13] David Y.W. Park, Ulrich Stern, Jens U. Skakkebæk, and David L. Dill. Java model checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 253–256, Grenoble, France, September 2000. IEEE.

[14] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276, Helsinki, Finland, September 2003. ACM.

[15] Nastaran Shafiei and Franck van Breugel. Towards model checking of computer games with Java PathFinder. In *Proceedings of the 3rd International Workshop on Games and Software Engineering*, pages 15–21, San Francisco, CA, USA, May 2013. IEEE.

[16] Mateusz Ujma and Nastaran Shafiei. jpf-concurrent: an extension of Java PathFinder for java.util.concurrent. In *Proceedings of the Java Pathfinder Workshop*, Lawrence, KS, USA, November 2011.

[17] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[18] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, Grenoble, France, September 2000. IEEE.