Automatic Handling of Native Methods in Java PathFinder

Nastaran Shafiei
NASA Ames Research Center, Moffett Field
nastaran.shafiei@nasa.gov

Franck van Breugel
York University, Toronto
franck@cse.yorku.ca

ABSTRACT

Java PathFinder (JPF) is a model checker for Java applications. Despite its maturity, JPF cannot be used to verify any realistic Java application without a nontrivial amount of work done by its user. One of the main limiting factors towards model checking such applications is handling native calls. JPF provides ways for users to handle such calls. However, those require modeling the behaviour of the native methods in Java which is labour intensive and hinders the uptake of JPF by developers. This paper presents our tool that extends JPF to address this problem. Our work alleviates this burden for users by automatically handling native calls. Our approach is based on delegating the execution of native calls from JPF to their original execution environment. We showcase our extension by applying it to a variety of simple yet realistic Java applications that JPF, without our extension, cannot handle.

1. INTRODUCTION

JPF is an explicit-state model checker for Java applications. The core of JPF is a Java virtual machine (JVM) using a runtime scheduler that executes the system under test (SUT) in all possible ways. As it explores the state space of the SUT, it checks for certain properties, such as deadlocks and unhandled exceptions. The JPF distribution, called jpf-core, comes with several classic concurrency examples, e.g., 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, they cannot be considered realistic. They lack ingredients in today's Java applications, such as a graphical user interface, interaction with a database, and communication over the Internet. JPF crashes on almost any realistic Java application and applying it requires a considerable amount of work.

One of the main challenges of applying JPF on real-world applications is to handle *native methods*, that is, methods invoked from the Java application but written in a different language, such as C and C++. The JVM of JPF can only execute Java bytecode instructions. It crashes as soon as it encounters a call to a native method unless the call is intercepted by JPF. JPF provides two different mechanisms (Section 2) to handle calls to native methods. Both approaches require the user to manually model native methods in Java which is tedious and error prone. Moreover, many such methods are not part of the documented Java APIs, and their sources might not be even available.

In this paper, we present our tool, *jpf-nhandler*[1], which is an extension of JPF that automatically handles native calls. To handle such calls, it provides a way to delegate their execution to the operating system (OS). There are existing tools that take a similar approach. VeriSoft [8], a model checker for C code, intercepts certain system calls such as operations on communication objects. KLEE [4], a symbolic execution tool, also delegates read accesses of file systems to the OS. Another example is the Moles framework [6] which supplements a symbolic execution tool, Pex. It replaces calls with alternative implementations to eliminate environment dependencies such as the file system. Such calls need to be specified by the user.

Our tool has been successfully applied to a variety of simple yet realistic Java applications. In the paper we present a variety of examples which confirm that jpf-nhandler can be used to model check Java applications including unhandled native calls. For all these examples, JPF without our extension crashes. We also discuss the limitations of our approach. Before we outline our approach, in the next section, we explain those JPF features that are key to our work.

2. NATIVE CALLS IN JPF

JPF is a JVM on top of which the SUT runs. JPF itself is written in Java and, hence, runs on top of another JVM, called the *host JVM*. The latter runs on top of the OS. There are two approaches adopted by JPF to handle native calls, both require modeling native methods in Java. In one approach such model is provided at the JPF level within *model classes*. These classes model the behaviour of actual Java classes, and often abstract from particular details of the actual classes. They are part of JPF and have the same names as the classes they model. Whenever JPF encounters a class for which there exists a model class, it model checks the model class instead of the actual class.

Another way is to provide a model of the native method at the host JVM level using JPF's *model Java interface* (MJI).

[1]https://bitbucket.org/nastaran/jpf-nhandler

MJI is implemented in analogy to the *Java native interface* (JNI) [9]. Every standard JVM has a JNI, and whenever it gets to a native call, JNI is used to transfer the execution from the JVM to OS. The so called *native peer* classes play a key role in MJI. JPF uses a specific name pattern to associate native peer methods with the methods of SUT classes. Whenever it gets to a call associated with a native peer method, it delegates the call to the host JVM. Hence, the native call is not model checked, but executed on the host JVM.

Since classes and objects are represented differently in JPF than in an ordinary JVM, to implement a native peer class one often has to translate from the one representation to the other and back. This is one of the challenges of developing native peers. For example, consider the native method `getLength` of the class `java.lang.reflect.Array`. Let `arr` be an array object. The call `Array.getLength(arr)` returns the length of `arr`. In JPF, `arr` is represented by an `ElementInfo` object. To handle such a call, one needs to be familiar with the `ElementInfo` structure to retrieve the length. Moreover, the user needs to use the required naming pattern when implementing the native peer to make JPF associate it to `getLength`.

JPF and its extensions currently include a few hundred model classes and native peer classes. These classes are mainly developed to handle native methods. However, there are many more classes with native methods that are not handled. Moreover, the implementation of existing model classes and native peer classes are only compatible with certain Java versions. Furthermore, any Java application can include user-defined native methods. To avoid JPF from crashing on an application with an unhandled native call, one needs to apply one of the two approaches which require knowledge of JPF's internal structure and the method specification. That can considerably limit the use of JPF. Thus, a tool that can assist users to handle native calls automatically can be very useful.

3. THE JPF-NHANDLER TOOL

As mentioned earlier, to handle native calls in a standard JVM, the execution is automatically transferred to the OS level. In a way, jpf-nhandler mimics the same functionality, but at a different level, i.e., it automatically transfers execution between JPF and the host JVM, whereas, the JNI feature transfers the execution between the host JVM and the OS.

Our tool mainly relies on MJI and native peer classes. It consists of three main components, *forwarder*, *code generator* and *converter*. When classes are being loaded into the JPF runtime environment, the forwarder identifies and flags the calls to be delegated by jpf-nhandler. The code generator creates bytecode for native peers on-the-fly (referred to as OTF peers) using the library BCEL[2]. The converter is used by OTF peers to translate objects and classes from JPF to the host JVM and back. Whenever JPF encounters a call to an unhandled native call, jpf-nhandler automatically intercepts the call and delegates its execution from JPF to the host JVM. To delegate the execution, the code generator creates an OTF peer (if it does not exist yet) and adds a native peer method to it (if it does not exist yet).

To see how the body of OTF peer methods are implemented, consider the native method `allocateInstance` of the class `sun.misc.Unsafe`. Let `unsafe` be an `Unsafe` object and let `clazz` be a `Class` object. The call `unsafe.allocateInstance(clazz)` returns an uninitialized instance of `clazz`. The following figure shows how this call is handled by jpf-nhandler. The middle column represents the code generated on-the-fly by jpf-nhandler as the body of the `allocateInstance` native peer method which includes three steps. (1) First, the JPF representation of `unsafe` and `clazz` are transformed to corresponding JVM objects using the converter. (2) Then, using the Java reflection API, the original native method `allocateInstance` is called on the JVM representation of `unsafe` with the JVM representation of `clazz` as its argument. Since this method is native, its execution is delegated from the host JVM to the OS, using JNI. (3) Finally, using the converter, the result of the method call is transformed from its JVM representation to its JPF representation. 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).

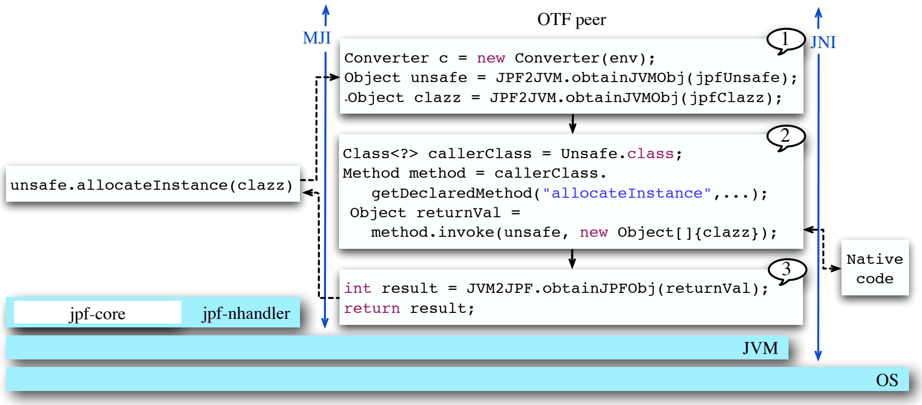
JPF uses instances of the class `ClassInfo` and `ElementInfo` to represent classes and objects, respectively. Since jpf-nhandler interacts with the host JVM (steps 1 and 3 in the following figure), it needs to convert objects and classes from JPF to the host JVM and back. Consider converting a JVM object to a JPF object. Conversion is performed recursively. Using the Java reflection API, the converter goes through the fields of the JVM object and for each non-primitive field it performs a conversion from JVM to JPF.

However, this *generic* converter does not work if there is an inconsistency between a model class and the actual class it models. Since model classes abstract away details from the original ones, they usually do not declare the same fields as declared in the original classes. However, the converter, explained above, relies on an one-to-one correspondence between the fields of the model class and the actual class. To address this issue, the converter uses the *abstract factory* design to instantiate objects of type `Converter`, the subclasses of which implement type-specific conversions. The factory returns the generic converter, encapsulated by `GenericConverter`, if there is no inconsistency, and a hand crafted converter otherwise. For example, since the `java.util.Random` model class and the actual class do not declare the same fields, the factory returns a converter specific to `Random` objects.

Our tool can be configured in a variety of ways. It can be used to skip native calls instead of delegating them. It also provides a way to specify which methods (native and non-native) are delegated or skipped. Moreover, for methods with primitive return types, it can make their OTF peers return a nondeterministically chosen value. It can also be configured to generate source code for OTF peers on-the-fly.

There are some limitations to our tool which are outlined below. Providing the source code of OTF peers allows users to refine the implementation when a limitation is encountered. In general, 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 the call `unsafe.allocateInstance(clazz)`. Only changes made to the objects `unsafe` and `clazz` and their classes are reflected

[2]http://commons.apache.org/bcel/



```
unsafe.allocateInstance(clazz)
```

```
Converter c = new Converter(env);
Object unsafe = JPF2JVM.obtainJVMObj(jpfUnsafe);
Object clazz = JPF2JVM.obtainJVMObj(jpfClazz);
```

```
Class<?> callerClass = Unsafe.class;
Method method = callerClass.
   getDeclaredMethod("allocateInstance",...);
 Object returnVal =
   method.invoke(unsafe, new Object[]{clazz});
```

```
int result = JVM2JPF.obtainJPFObj(returnVal);
return result;
```

in JPF. However, if the method were to change any other objects or classes, those changes would not be reflected in their corresponding JPF representations. Moreover, delegation of a method to the host JVM amounts to the assumption that its execution is atomic. Furthermore, as JPF explores the state space of the SUT it may backtrack to already visited states. Therefore, a method may be delegated multiple times which may lead to undesirable consequences, e.g., a method that adds an entry to a database.

Due to these limitations, our tool is not *sound*, i.e., it may explore executions which are inconsistent with the SUT behaviors, leading to false positives. It is also not *complete*, i.e., it does not capture the state of the delegated calls, and therefore the model checker may miss executions. However, in practice, jpf-nhandler has proven to be useful. As is shown in Section 4.1, it is applicable to a large variety of applications.

4. RESULTS AND TOOL USAGE

To compare our approach with the hand crafted approach adopted by JPF, we selected four Java types that have been already modeled by JPF. For each type we use a class with a test suite that checks the correctness of the class methods. We apply jpf-core and jpf-nhandler on our test classes. In both cases, the classes are model checked successfully, and all the tests pass. For each type, jpf-nhandler is configured to delegate all the methods modeled by jpf-core. We apply jpf-nhandler with two different settings. In one setting, it generates OTF peers, and in the other one, it reuses the OTF peers created in previous runs.

The following table presents our results which are the average of ten runs. The average standard deviation is 14 ms. The last column of the table presents the total size (in terms of lines of code) of the native peer and the model class used to model the type in jpf-core. For classes `Math` and `Array`, which are mostly implemented natively, jpf-nhandler would avoid developing 443 lines of code requiring knowledge of JPF's internal structure and the method specifications.

4.1 Application of jpf-nhandler

Recently, our tool has been used in verifying a prototypical next-generation air traffic controller system, Autoresolver, at NASA. Below, we discuss simple Java applications which contain ingredients found in today's applications. None of them can be verified by JPF without jpf-nhandler.

Communicating over a Network. A common way for Java applications to communicate is using `java.net.Socket` objects. We implemented a client application and a server application. Both applications contain native calls that are not handled by JPF. Using jpf-nhandler, we model check the client on one machine when the server is running on another machine. Red Hat's JGroups provides a framework for reliable multicast communication. In our example, two applications communicate using a `org.jgroups.JChannel`. One application sends a message which is received by the other one. We execute the receiver on one machine while we model check the sender on another machine.

Exploiting Graphics Processing Units (GPUs). NVIDIA's compute unified device architecture (CUDA) contains an environment to program their GPUs. The package *jcuda* enables Java applications to run CUDA code on the GPU. We consider the application `JCudaVectorAdd` which is part of the jcuda distribution. It creates two 100,000 element arrays, runs some CUDA code on the GPU to add the arrays, and finally checks the result. Several native methods are needed to bridge the gap between the Java code and the CUDA code. By delegating those calls using jpf-nhandler, we were able to model check this application which led to detection of a bug in JPF, i.e., arrays of floats are not converted correctly.

Querying a Database. Apache Derby is a relational database written in Java. In our application, we connect to a database, create a new table, insert records into the table, and finally close the connection. Our application includes several unhandled native calls. Using jpf-nhandler, we model check this application successfully.

Scraping the Web We developed a web scraper which reads the HTML of a web page. Our application contains several calls to native methods that are not handled by JPF. Using jpf-nhandler, this application is successfully model checked.

Invoking Web Services Google's translate web service-translates phrases between natural languages. In our application, we use this web service to translate phrases from English to French. By delegating a call that sends an HTTP request and returns the result as an object, the application is verified successfully.

type	jpf-core (ms)	jpf-nhandler (ms)	reusing peers(ms)	overhead	reusing peers overhead	modeling effort(loc)
String	3767	4582	4038	21%	7%	771
Math	6169	6974	6574	13%	6%	171
Array	44406	5007	4802	13%	8%	272
AtomicLong	4250	4719	4485	11%	5%	42

Playing Games. The Java code of computer games is usually full of calls to native methods. In [10] we describe how jpf-nhandler is used to model check two computer games, Hamurabi and a graphics based version of rock-paper-scissors.

Solving Ordinary Differential Equations. The Apache Commons mathematics library includes packages related to mathematics and statistics. In our application, we use the library to numerically solve ordinary differential equations. Successfully verifying the application requires using jpf-nhandler.

5. RELATED WORK

The work presented in [7] also extends JPF to model check some parts of the code and executes the rest on the host JVM, but their objective is reducing the execution time and not handling native calls. Their approach also translates JPF objects to JVM objects and back, but has several limitations from which ours does not suffer, e.g., their translation from JPF to JVM handles neither arrays nor instances of classes without a default constructor. They also use reflection to invoke methods on the host JVM, but they do not handle constructors and static initializers which are handled by us.

There are several other extensions of JPF that also deal with native methods. The work presented in [2] introduces a framework to model check distributed Java applications. It consists of several model classes that model network communication in Java and contain native methods. The JPF extension *jpf-net-iocache* [1] also model checks distributed Java applications. It allows for model checking one component of the distributed application, while the others run in their normal environment. This is similar in flavour to our tool. Since JPF can backtrack to previously visited states, one has to prevent communications between that component and the others from being repeated. To address this, they introduce a cache that keeps track of communications. The work presented in [11] discusses the JPF extension *jpf-concurrent* which models the package `java.util.concurrent`. Although their main aim is to improve the performance of JPF, they also handle several native methods. Without using jpf-concurrent, our tool can model check Java applications using `java.util.concurrent`, e.g., we have successfully model checked the implementation [3] of a concurrent binary search tree.

6. CONCLUSION AND FUTURE WORK

This paper presents our extension jpf-nhandler of JPF that automates the handling of native methods. It automatically delegates the execution of the native method to the host JVM. It automates the intertwining of the model checking of Java code and the execution of native code. In a way, it is similar to concolic execution.

Next we outline our plans to address some of the tool's limitations. To avoid a call from being delegated more than once, we intend to use a cache (as it is used in [1]) to record the effects of a delegated method call. If JPF encounters the same call later, we simply reflect the cached effects in JPF. The jpf-nhandler converter (Section 3) has a map which associates JPF objects to their JVM representations created by the converter. By default, this maps is cleared after a native call is handled. jpf-nhandler can be configured to not clear the map to avoid recreating JVM objects. However, in this setting, if JPF modifies an object that is stored in the map, the JPF object and its JVM counterpart may get out of sync. This discrepancy can be addressed using a lazy update strategy explained in [7].

Objects are represented in JPF by `ElementInfo` objects which contain more information than their corresponding JVM counterparts. Hence, in the vocabulary of abstract interpretation [5], the JPF representations form the concrete domain and the JVM representations form the abstract domain. The conversion from JPF to JVM 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.

7. REFERENCES

[1] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *TOOLS*, 2008.

[2] E. D. Barlas and T. Bultan. NetStub: a framework for verification of distributed Java applications. In *ASE*, 2007.

[3] T. Brown and J. Helga. Non-blocking k-ary search trees. In *OPODIS*, 2011.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[5] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2), 1996.

[6] J. d. Halleux and N. Tillmann. Moles: Tool-Assisted Environment Isolation with Closures. In *TOOLS*, 2010.

[7] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, 2006.

[8] P. Godefroid, J. D. Herbsleb, L. J. Jagadeesan, and D. Li. Ensuring Privacy in Presence Awareness Systems: An Automated Verification Approach. In *CSCW*, 2000.

[9] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. 1999.

[10] N. Shafiei and F. v. Breugel. Towards model checking of computer games with Java PathFinder. In *GAS*, 2013.

[11] M. Ujma and N. Shafiei. jpf-concurrent: an extension of Java PathFinder for java.util.concurrent. In *JPF Workshop*, 2011.