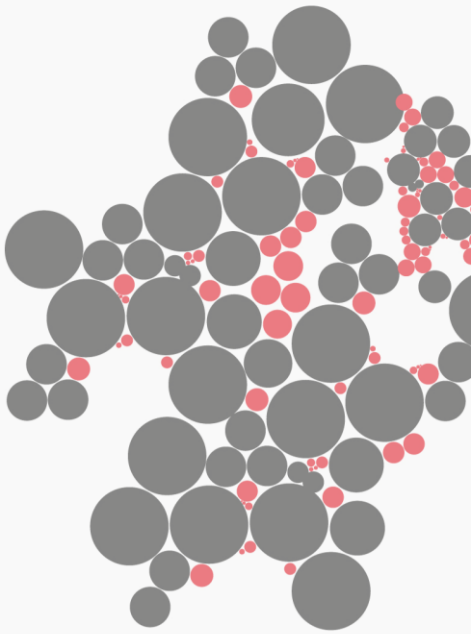
MASTER THESIS



Maintaining the Causal Thread of Execution in Distributed and Multilanguage Software

Rik Schutte

Twente Research and Education on Software Engineering

Faculty of Electrical Engineering, Mathematics and Computer Science

University of Twente

Netherlands

EXAMINATION COMMITTEE
dr. S. Malakuti
dr. ing. C.M. Bockisch
prof. dr. ir. M. Aksit

18-10-2011

Abstract

Runtime verification is a technique used to check the execution trace of a program against specified properties. During the execution of a program, the changes in the state of the program are observed and are verified against the specified properties of the program.

We want to apply runtime verification to check sequences of events, which are distributed (they are generated in separate processes, which communicate with each other) and span multiple programming languages. There are currently two E-Chaser versions, which supply separate solutions for multiple language and distributed software. In this thesis we combine these two versions to create a new solution.

The solution is given in two parts. The first part of the solution is about the call graph approach, in which we try to generate call graphs of multiple language software. From a call graph we try to deduce structural information. Due to the problems related to combining call graph of different programming languages, a different approach was taken. The second part describes a new verification system design which extends the previous E-Chaser versions.

We have created two tools that adapt the source code of a program written in C and Java. We evaluate these tools against testing applications. The thesis is concluded with ideas for optimizing the solution.

Table of Content

List of Figures

List of Tables

List of Code Fragments

1 Introduction

Complex software usually consists of multiple subsystems, which are possibly implemented in different languages and/or are distributed across multiple processes. A behavior of such software is usually accomplished by a sequence of causally-dependent events, which span across multiple subsystems. The term **causality** refers to the relationship between an event (the cause) and a second event (the effect), where the second event is a consequence of the first.

An example of such software is embedded software in which subsystems are often written in different languages. Usually parts of the system are written in languages that are best suited for the task. For example: High level programming languages are used for user interaction and lower level languages for communicating with hardware and other low level components.

Another example is applications that use a network infrastructure, such as applications which use a service-oriented architecture. These applications consist of multiple subsystems, which can be accessed remotely.

This chapter first provides the necessary background on runtime verification tools. Second, it explains the example software that is used as the case study throughout the thesis. A discussion of problem statement is provided followed by the scope of the thesis.

1.1 State of the Art of Verification and Validation Techniques

There are several verification and validation techniques available. We will give a short overview of these techniques.

Static analysis is the analysis of code without executing the program. A strength of static analysis is that it can find errors that may only appear under very specific conditions when the program is executed. A weakness of static analysis is that it does not deal with a running program and it may not be able to verify properties depending on runtime context [37].

Model checking techniques verify models of the program against formally specified properties of the program. A weakness is that a model of the program is checked instead of the actual program.

Some properties of programs may not be verifiable in a model, since a model may not contain all the logic that the original program has [38].

Testing checks a result against an expectation of a program, during runtime. An expectation can be captured in a test case which is performed by a tester, who uses the program through its user interface. Another option is to write specific code for the purpose of testing the program. A strength is that the program is tested during execution, which allows the runtime context (e.g. user input) to be part of the testing. A weakness is that a test suite is almost never complete, since not all paths in a program can be easily tested. Another weakness of testing is that the program being tested is not executed in its target environment [39].

Runtime verification techniques are used to check the execution trace of a program against specified properties of the program. During the execution of a program, the changes in the state of the program are observed and are verified against the specifications. If any failure is detected, diagnosis and recovery actions may be performed. Runtime verification is introduced as a complement to the other techniques to verify a program in its target environment. A strength of runtime verification is that it functions during the execution of a program, unlike static analysis. It also works in the actual execution environment of the program, unlike testing. Another strength is that it tests the actual program, and not a model, like model checking.

Due to the complexity of software it is not possible to detect all failures of the software before its actual execution. Runtime verification techniques can be used to detect (and act upon) failures that are only visible during the execution of a program.

1.2 Architecture of Runtime Verification Tools

In the literature there are runtime verification tools that can be employed for the implementation of runtime verification techniques. As it is explained in [1], most of the tools adopt a two-layer architecture; a specification layer and a verification layer. Figure 1-1 shows such an architecture [1].

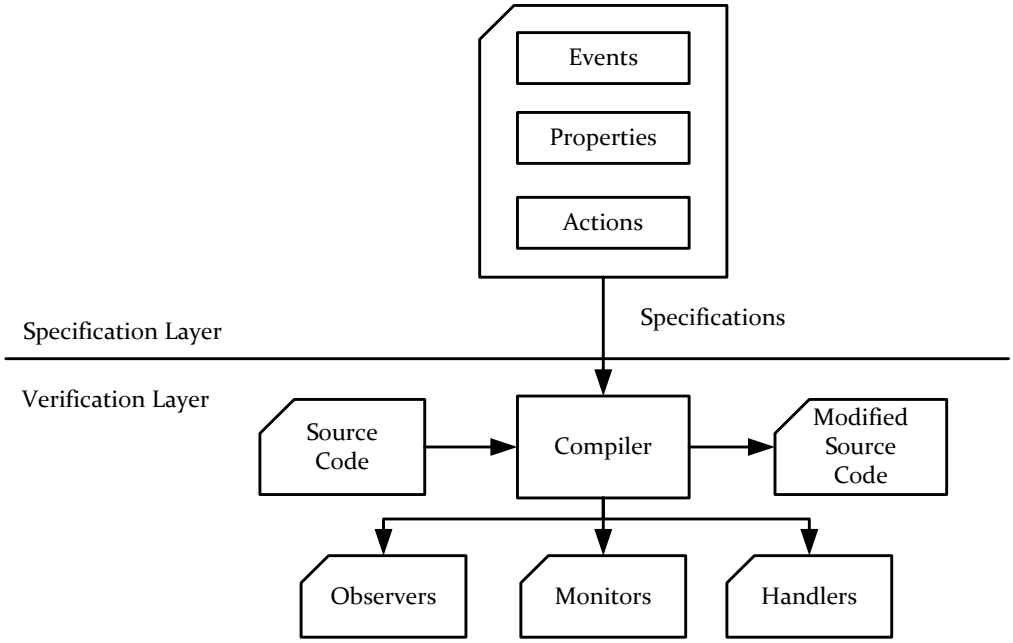


Figure 1-1: Typical architecture of a runtime verification tools

In the specification layer the events, properties and actions are specified. Events represent changes in the state of a program. Properties are logical predicates over events, expressed in a formalism. Actions specify functionality (usually written in code), which is executed once a property is validated or violated.

In the verification layer the compiler takes as input the original source code and specifications. The compiler generates the observers, monitors, handlers and modified source code. Observers are generated from events, monitors from properties and handlers from actions.

Observers are responsible of notifying monitors about the occurrence of the specified events. Monitors are in charge of verifying the occurred events against the logical predicates and depending on the result of verification they call the corresponding handlers, if there are any. Likewise, handlers are in charge of executing the functionality specified in actions.

1.3 Running Example

For the sake of illustration consider the following example. There is a document editing program which consists of two subsystems: document-client and document-server. The former provides a user-interface through which user can enter a filename and a text. The latter provides the core functionality of the program, which is saving the document. Its architecture is shown in Figure 1-2.

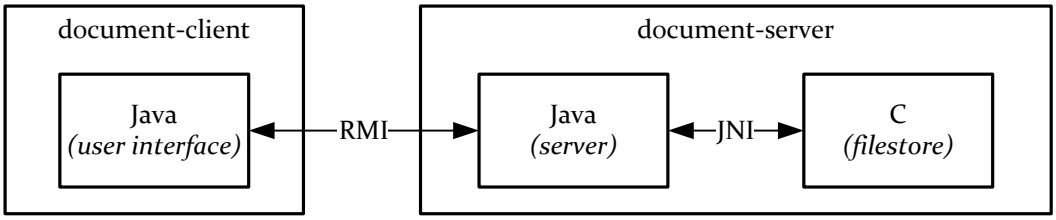


Figure 1-2: Architecture of the document editing sofware

The document-client is implemented in Java, and the document-server is implemented in Java and C. The C code of the document-server performs the saving of the document to disk, whereas the Java code acts as a server for the document-client. The C code in the document-server is also called the *filestore*. The document-client and document-server communicate through Java RMI. The Java and C code communicate through JNI.

Java RMI (Remote Method Invocation) [44] is the remote procedure calls API of Java. It allows remote procedure calls between Java processes. JNI (Java Native Interface) [41] is a layer that enables Java code running inside a virtual machine to call and be called from C.

Assume that we would like to verify at runtime that a request to save a document by the user eventually results in storing the document content on the file system. Figure 1-3 shows the sequence of causally-dependent events that handles such a request. The figure shows that document-client and the document-server processes. The sequence of events starts when the user invokes the functionality to store a document ("Store Document" arrow). Then the document-client invokes the document-server through the `saveDocument(filename, text)` invocation. This invocation uses RMI. The document-server then calls the filestore through a JNI call named `storeFile(filename, text)`.

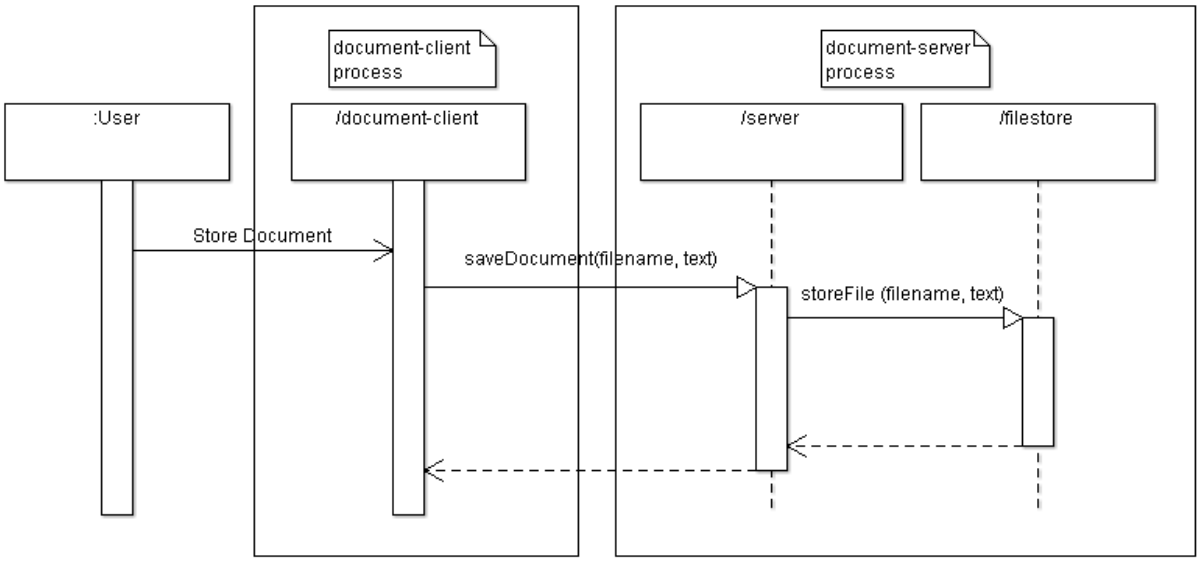


Figure 1-3: Sequence of events to store a document

This running example illustrates a program that is both distributed and written in multiple languages. It is an example of the complex software that we mentioned in the introduction of this chapter. The running example should be supported through changing the E-Chaser, which we have chosen as our verification system. The reasons for choosing E-Chaser are that there are two versions which separately support multilanguage programs and distributed programs. We will explain E-Chaser further, later in this chapter.

1.4 Problem Statement

To be able to apply runtime verification to check such sequences of events, a runtime verification technique must be able to deal with multiplicity of processes and languages, both in its specification layer and verification layer.

There are several runtime verification tools introduced in the literature as the study in [1, 2] shows, they fall short in dealing with the multiplicity of processes and languages.

E-Chaser is a runtime verification tool developed in the TRESE group, which aims at overcoming the shortcomings of the existing RV tools. E-Chaser is an extension to the aspect-oriented language and compiler Compose*.

Two versions of E-Chaser are available. The first version aims at supporting multiple language software, and has the following characteristics:

- The specification language is independent of the programming languages, because it is based on Compose*, which provides this feature.

- Since E-Chaser is based on Compose*, it can generate code for Java, .NET and C/C++
- It only supports the verification of multiple languages software, through a distributed setup, because the synchronization process always runs in a separate process.
- Specifications are not distribution transparent, because each separate module running in a separate process needs a separate specification.

The second version [2] aims at supporting distribution and has following characteristics:

- Distribution transparent specification. This means that only one specification is needed to for our case study example. If we wish to write a trace that matches the events in Figure 1-3, we will only need one specification instead of the two necessary in the previous E-Chaser version.
- A compiler that infers the distribution structure through code analysis, and accordingly generates code for observers, monitors and handlers.
- Only Java software which makes use of RMI is supported.

Current versions of E-Chaser *either* provide multiple language *or* multiple process (distributed) support. Because there are a lot of programs that are distributed *and* written in multiple programming languages, we wish to verify these. To achieve it we will combine the two versions of E-Chaser into a new version.

1.5 An Overview of the Solution

We wish to extend the second version of E-Chaser and integrate our new solution into it. We reuse the architecture of the second E-Chaser version, because 1) its specification is language independent and 2) its specification is distribution transparent.

The architecture of the second version of E-Chaser is shown in Figure 1-4. For a new version we want to be able to deduce the distribution of a program for multiple programming languages.

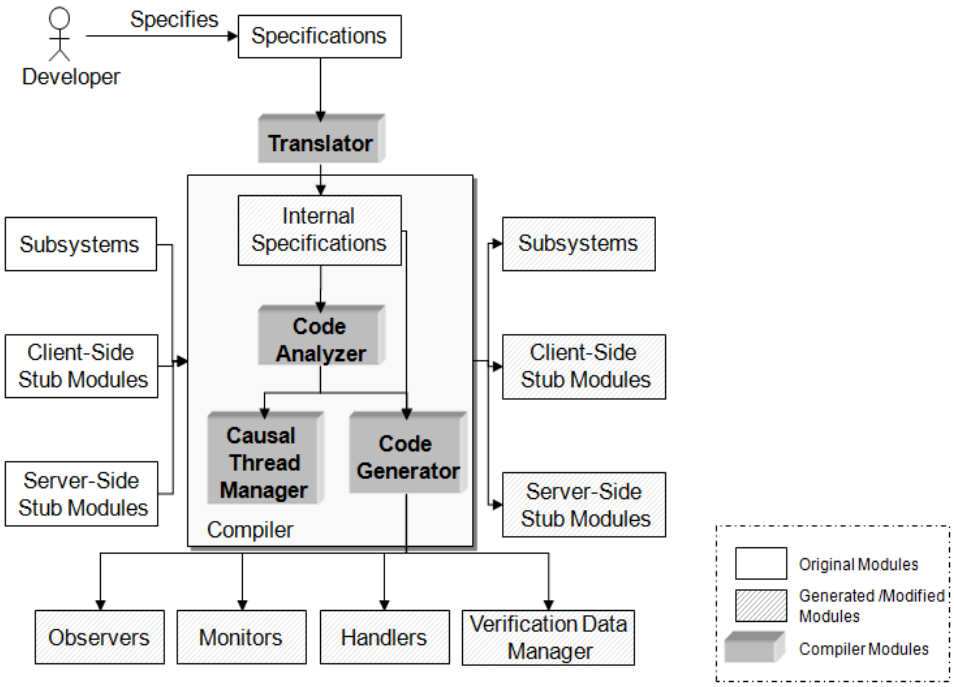


Figure 1-4: Architecture of the second E-Chaser version

In the second version of E-Chaser, the developer first writes a *specification* in which the events, properties and actions are defined. The *Translator* converts this specification to an *internal specification*. The internal specification is hidden from the developer. A more complete overview with additional information about the internal specification and the conversion can be found in [1] and [2].

The *Code Analyzer* extracts a call graph of the program and identifies remote methods. The *Causal Thread Manager* modifies the client and server-side stub modules of remote method invocations, so it can maintain the causal thread of execution. The *Code Generator* generates the code, based on the provided information by the *Code Analyzer* and the *Causal Thread Manager*. The generated code consists of the observers, monitors, handlers and the *Verification Data Manager*, which is executed in the central verification process. It also generates the (modified) subsystems, client and server-side stub modules.

To support this, we explored a solution in which we keep the functionality of the *Translator* and *Causal Thread Manager*. We want to change the *Code Analyzer*. It must generate call graphs of programs, in which remote method calls and the usage of JNI can be found. The *Code Generator* must use the call graph to create code, which works for a program that is distributed and written in multiple languages.

We were unable to implement the previously described solution, because we were unable to deduce the distribution of a program which uses the JNI layer. The distribution information was

necessary to generate optimized monitors for the verification system. Connecting Java and C call graphs was impossible, because the edges from C to Java depended on runtime information. Therefore we implemented a different solution.

The other solution creates (a part of) a verification framework that is distribution independent and thus does not rely on inferring the distribution of the program. In this solution the *specification*, *Translator*, *internal specification* and *Causal Thread Manager* keep the same functionality. The *Code Analyzer* still generates the information needed for the *Causal Thread Manager* to function, but it also has to find JNI calls in Java and C source code. The *Code Generator* has to generate adapted JNI calls and insert them at the location of the original JNI calls. For a complete implementation of the solution, additional changes to the *Code Generator* are needed. It is regarded as future work.

1.6 Scope of the Thesis

In this thesis we focus on programs that are written in Java and C and that use JNI for communication between the two languages. We also focus on the usage of Java RMI for remote communication and the creation of distributed programs.

Currently we have only implemented and demonstrated the use of our verification system in a single threaded environment. We do this to simplify the current implementation. For some language combinations sharing the causal thread id is harder, especially in a multilanguage and multithreaded environment. In such an environment the interlanguage layer may introduce additional complexity and change the thread of execution between the two programming languages.

2 Solution 1: Generating Optimized Monitors

We want to deduce the distribution structure of the program, so that we can generate optimized observers, monitors and handlers. The optimization lies in the generated architecture. If a non-distributed program is verified, then the verification system is generated as part of the program. If the program is distributed, then the verification system is partly run in a separate process.

This chapter explains a possible solution to extract a call graph of multiple language and distributed programs, which are developed in C and Java and make use of RMI and JNI. The call graph is used to deduce the structure of the program. After explaining the goal of the chapter and giving background information we will explain our approach step by step.

We want to support distributed and multilanguage software in both the specification and the verification layer. In the specification we want to keep 1) language independence and 2) distribution transparency (as provided by the second version of E-Chaser). In the verification layer we want to infer the distribution structure of the program and accordingly generate observers, handlers and monitors. If the program is not distributed, we generate a verification system that is added to the original program. If the program is distributed, we generate a verification system that (partially) runs in a separate process.

To infer the distribution structure, we extract the call graph and analyze each path in the call graph to see whether the specified sequence of events occur in the path. If so, we check the path to see whether it is multilanguage and distributed.

An example of a non-distributed program's architecture is given in Figure 2-1. It is the document editor of the case study.

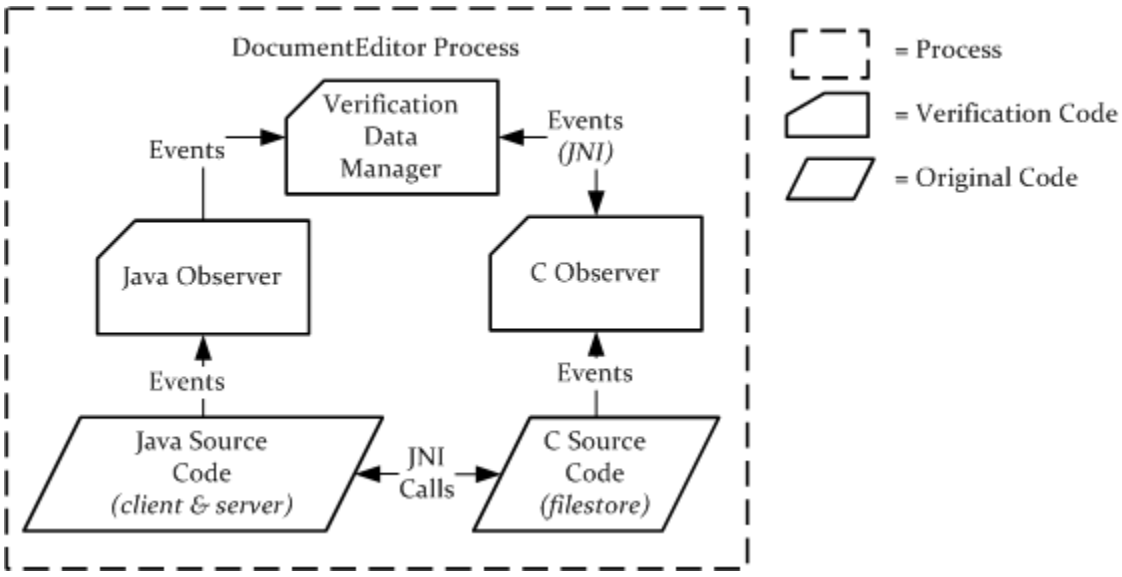


Figure 2-1: Architecture of a non-distributed program with inlined verification system

The document editor is shown as being a non-distributed system. In the original case study it was a distributed system, but in case we would put the *client* (GUI) in the same process as the *server* and *filestore*, the program would consist of one single process without any RMI calls. In such a program, we can run the *Verification Data Manager* in the same process as the whole program. The *Verification Data Manager* is a component that contains the verification information, such as the state of traces. The *Java Observer* gets the events from the *Java Source Code* and sends them to the *Verification Data Manager*. The *C Observer* gets its event from the *C Source Code*. The *C Observer* sends its events to the *Verification Data Manager* through JNI calls.

For a distributed program, the verification architecture is displayed in Figure 2-2. This program is used as running example throughout this thesis and is introduced in the first chapter, in the Running Example section.

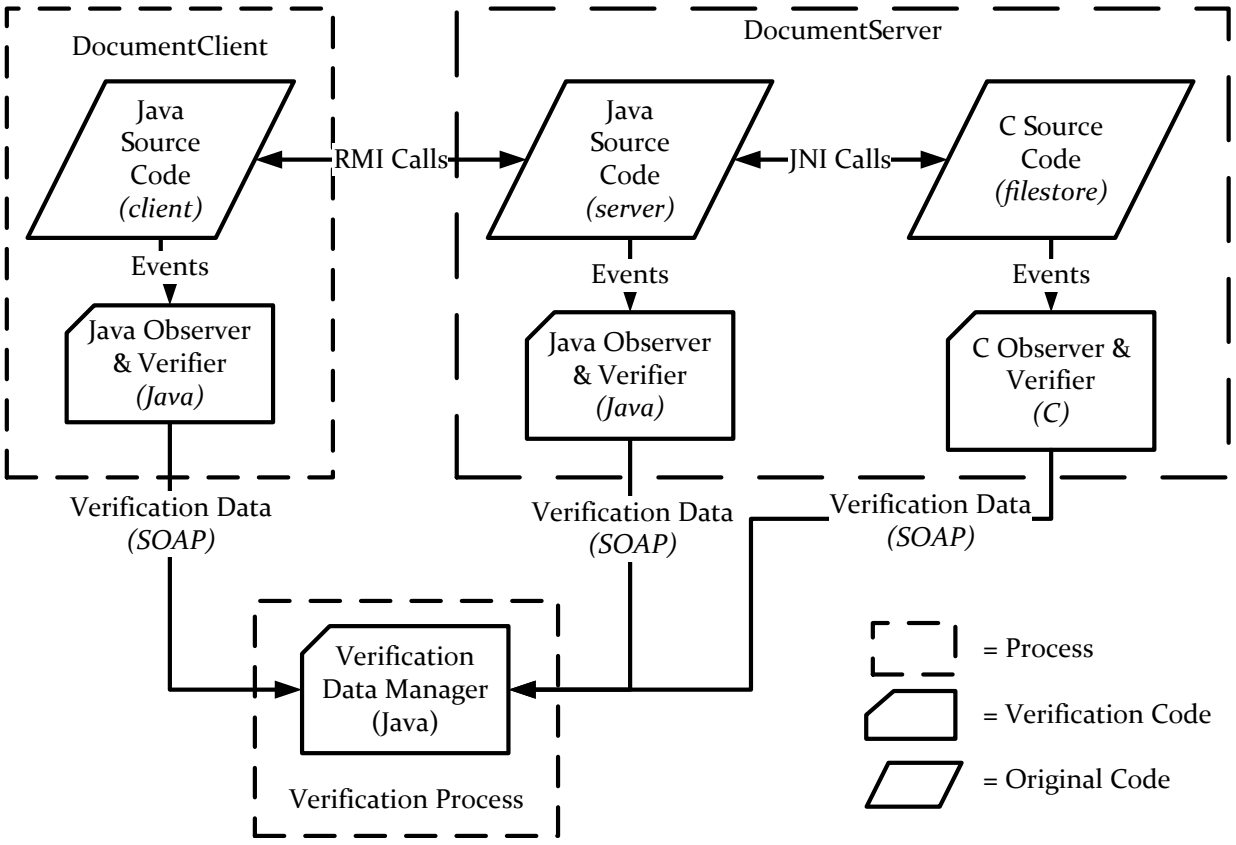


Figure 2-2: Architecture of a distributed program with external verification process

Both the *Java Source Code* on the client and the server report their events to their respective *Java Observer & Verifier*. The *Java Observer & Verifier* then retrieves verification data from the *Verification Data Manager* through RMI calls and checks the received event against the *verification data*. The *Verification Data Manager* runs in a separate process called the *Verification Process*. The *C Source Code* reports its events to the *C Observer & Verifier*. The *C Observer & Verifier* reports its events to a special piece of Java code, called the *Java Event Bridge & Verifier*, which retrieves the *verification data* and checks the events.

2.1 Background

2.1.1 Call Graph

A call graph is a directed graph in which the call flow of a program is depicted. Its edges represents the calling relationship between two functions or methods. Each node is a method or function and each edge *(f,g)* is the call of a function or method where function *f* calls function *g*. A call graph may contain cycles.

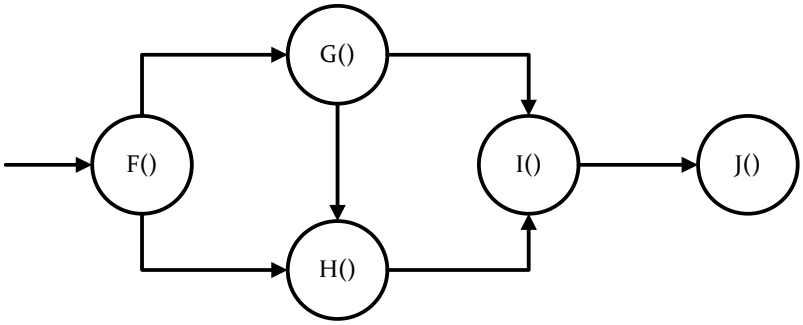


Figure 2-3: Example call graph

Figure 2-3 shows a very simple example of a call graph. It starts at a function call to F(), which in turn calls G() and H(). G() itself calls I(), but also H(). H() calls I() also. I() itself calls J().

There are two basic call graph types. Static and dynamic call graphs [40]. Static call graphs are generated without program execution. It may not contain all edges (when some edges cannot be statically determined, e.g. function pointers) or may not be totally precise and feature edges that are not actually executed (polymorphism). Dynamic call graphs are generated during the execution of a program and may not contain all edges, since it may be possible that not every statement of the program is executed.

2.1.2 Java RMI

The architecture of a RMI program is given in Figure 2-4. The server provides one or more interfaces that extend `java.rmi.Remote` interface. Inside these interfaces, the server defines the remote methods that can be called by clients. The server implements the interfaces.

The RMI compiler generates *stub* and *skeleton* classes for each of these remote interfaces. A *stub* is the client side code that is generated and it implements the remote interface and the remote communication for the developer. A *skeleton* does the same on the server side.

When a client invokes a remote method, the call goes to *stub*, where the generated code takes care of the remote communication (including marshalling the object). On the server side, the *skeleton* handles the communication and continues the flow of the program on the server side implementation object of the remote interface.

Any object that is registered to the RMI registry has to implement an interface that extends the `java.rmi.Remote` interface [12].

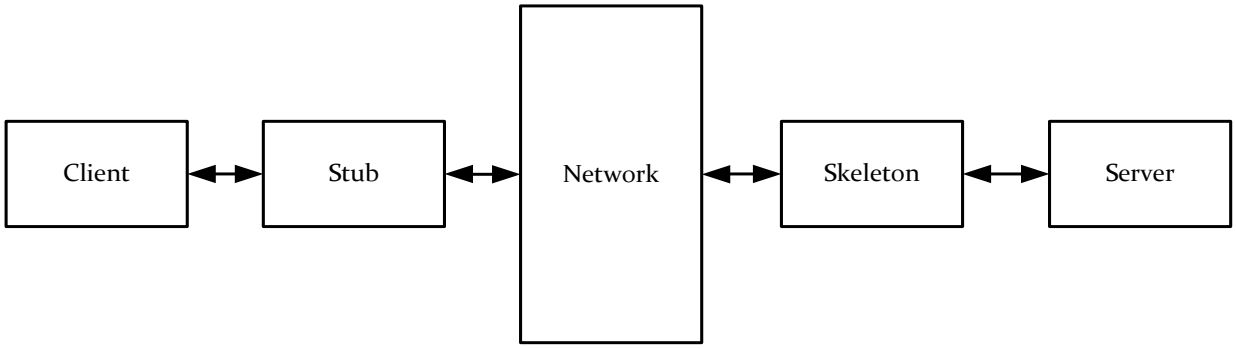


Figure 2-4: Architecture of a RMI program

Usage of Java RMI can be inferred from the Java code, because both sides of the communication access the `java.rmi.registry` package. This however does not mean that code from the package is actually used. Therefore, we need to look at statement that invokes methods that are part of an interface that extends the `java.rmi.Remote` interface. The code examples are from the case study that we introduced in the first chapter.

The code in Code Fragment 2-1 shows an example of a remote interface.

```
 1 package nl.utwente.rschutte;
 2
 3 import java.rmi.Remote;
 4 import java.rmi.RemoteException;
 5
 6
 7 public interface DocumentStore extends Remote {
 8
 9     public String storeDocument(String name, String text) throws RemoteException;
10
11 }
```

Code Fragment 2-1: Extension of the java.rmi.Remote interface

There are two important lines in Code Fragment 2-1. The first is line 7, in which the interface `DocumentStore` extends the interface `java.rmi.Remote` interface. At line 9, there is the `storeDocument` method declaration in the interface, with the name and text arguments of the type `java.lang.String`.

The implementation of the interface is shown in Code Fragment 2-2. Some irrelevant code has been left out for brevity.

```
 1 ...
 2
 3 public class DocumentStoreImpl extends UnicastRemoteObject implements
DocumentStore {
 4
       ...
 5
 6     public String storeDocument(String name, String text) {
 7            System.out.println("Stored: " + name + " | " + text);
 8
 9            FileStore fs = new FileStore();
10            String result = fs.storeFile(name, text);
11
13            return result;
14     }
15
       ...
16
17 }
18
```

Code Fragment 2-2: Implementation of the DocumentStore interface

At line 3, we see that the `DocumentStoreImpl` class implements the `DocumentStore` interface. Lines 6 – 14 shows the implementation of the `storeDocument(String, String)` method, which was defined by the `DocumentStore` interface.

The server registers objects in the RMI registry. This is done in Code Fragment 2-3 line 5. The variable `store` is registered under the name "store". In the same code fragment on line 4 the registry is located.

```
1 try {
2      DocumentStoreImpl store = new DocumentStoreImpl();
3
4      Registry registry = LocateRegistry.getRegistry();
5      registry.rebind("store", store);
6 } catch (RemoteException e) {
7      e.printStackTrace();
8 }
```

Code Fragment 2-3: The server side of Java RMI

Client-side code using the remote object is shown in Code Fragment 2-4. The registry is located on line 2. The client looks up the object that was stored by the server under the name "store" on line 3. The call that is distributed is the `store.storeDocument(…)` call (line 4), which can also be found in the interface definition (Code Fragment 2-1) and implementation (Code Fragment 2-2).

```
1 try {
2      Registry registry = LocateRegistry.getRegistry();
3      DocumentStore store = (DocumentStore) registry.lookup("store");
4      String result = store.storeDocument(this.getName(), this.getText());
5 } catch ( Exception e ) {
6      e.printStackTrace();
7 }
```

Code Fragment 2-4: The client side of Java RMI

2.1.3 JNI

JNI [41] is a layer that enables Java program to call and be called by C or C++ code. As such, there are two cases, Java calling C/C++ and C/C++ calling Java.

An example of Java to C source code is given in Code Fragment 2-5. The code is taken from our document editor case study.

```
 1 package nl.utwente.rschutte;
 2
 3 public class FileStore {
 4
 5     static {
 6            System.loadLibrary("filestore");
 7     }
 8
 9     public native String storeFile(String fileName, String text);
10
11 }
```

Code Fragment 2-5: Java JNI code

There are two points of interest about the JNI usage. In the class initializer the library "filestore" is included (lines 5 - 7). This loads the methods that are implemented in the "filestore" library. The method declaration in (line 9) is a JNI method (which is a method with a native implementation). It can be recognized by the *"native"* qualifier. The method declaration can be recognized by the "*native*" qualifier. The empty stub of the method can be found in Code Fragment 2-6.

The Java method shown in Code Fragment 2-5 stub has a C function as its implementation.

```
1 #include <jni.h>
2 #include "nl_utwente_rschutte_FileStore.h"
3
4 JNIEXPORT jstring JNICALL Java_nl_utwente_rschutte_FileStore_storeFile
5   (JNIEnv * jnienv, jobject obj, jstring filename, jstring text) {
6 }
```

Code Fragment 2-6: Filestore skeleton

An empty function can be generated by the *javah* tool [21], which is part of the standard JDK. Code Fragment 2-6 shows the empty function definition that has been generated from the Java JNI code in Code Fragment 2-5. The implementation is left out for brevity. The first line is a reference to `jni.h` which is the header file with the common JNI declarations. The other header file (line 2) contains a declaration of the function on lines 4 - 6. The function on lines 4 – 6 is the implementation of the method declaration given on line 9 of Code Fragment 2-5.

The opposite direction of communication is from C to Java. An example of an invocation of a Java method from C is given in Code Fragment 2-7. It shows the invocation of the main method of a program located in the `nl.utwente.rschutte.Editor` class from C code. From C, there are many functions which can be used to find methods in a Java program. A list of these functions can be found at [13].

```
 1 #include <jni.h>
 2
 3 int main() {
 4     JavaVM * jvm;
 5     JNIEnv* env = create_vm(&jvm);
 6
 7     jclass editorClass;
 8     jmethodID mainMethod;
 9
10     editorClass = (*env)->FindClass(env, "nl/utwente/rschutte/Editor");
11     mainMethod = (*env)->GetStaticMethodID(env, editorClass, "main",
12                  "([Ljava/lang/String;)V");
13
14     (*env)->CallStaticVoidMethod(env, editorClass, mainMethod, NULL);
15
16     (*jvm)->DestroyJavaVM(jvm);
17
18     return 0;
19 }
```

Code Fragment 2-7: C JNI code

In order to determine that a piece of C code invokes Java code, we look for functions that are part of JNI API. The JNI API has a subset of functions which are used to invoke Java methods. Just the inclusion or import of the JNI header (`jni.h` on line 1) does not indicate the invocation of Java from C.

In line 10 the `nl.utwente.rschutte.Editor` class is loaded. It contains the main method of the Java program. The main method is found in lines 11 and 12. The main method of the Java program is invoked via the function `CallStaticVoidMethod` in line 14.

In our example we invoke the main method of a Java program, but any Java method can be called from JNI. As such, the Java side of a C/C++ to Java invocation is an ordinary Java method.

2.1.4 Tool Reviews

For the extraction of call graph we use existing tools. A small overview of the available tools will be given to show the options. Some tools are not specifically aimed at generating call graphs; however they still generate information that can be used for this purpose.

The criteria for the tools are as follows:

- The tools needs to generate call graphs for either C or Java
- The generated call graph should be available to our tool in a format that allows further processing

A potential combination of Java and C tools should be available at a single platform, so they can be included in our solution.

Soot

Soot [15] is a Java optimization framework, which uses Java byte code as input. A call graph can be obtained by using the Class Hierarchy Analysis option. Soot has to run in whole program mode to be able to generate a call graph. Unlike most call graph tools, it uses the compiled code to generate a call graph. It can be used as a library for any other Java program. Soot only works for Java.

GNU CFlow

CFlow [16] is a tool specifically aimed at extracting call graphs from C source code. It generates a textual representation of the call graph. It can generate a caller-callee call graph as well as the reverse callee-caller graph. The textual representation of the call graph makes it easier for further processing. CFlow is only available for the Unix platforms (or similar, Linux, OSX etc.).

Doxygen

Doxygen [17] is a documentation system for several programming languages, including Java, C, C++ and C#. It generates documentation based upon a set of documented source files. However, it is also able to generate structural information of the program to aid a programmer in finding his way in a large source distribution. Doxygen works at the source code level.

Doxygen generates a call graph for each function/method or for each function/method that is annotated properly. A disadvantage of Doxygen is that the generation of the call or caller graph is not generated as text, but only as figures of graphs and immediately fed to dot (as part of GraphViz). The intermediate format needs to be available for our tool.

GProf

GProf [18] is a profiling tool for C programs. It performs dynamic analysis. GProf can generate call graphs based on the execution of a program. This is a drawback, because potentially not all functions in a program are covered in each run. An advantage of a GProf is that the call graph is represented as text and can be easily processed.

CScope

CScope [20] is a C source code browser, in which source code is presented in a structured way. The user must first index the source code that he wants to browse. The result is stored in a special database, which can be accessed by the CScope program. A call graph of the C source code is not immediately generated, however for each function the caller and callees can be found. As such, a call graph can be built if this information is used for every function. CScope is only available for the Unix platforms (or similar, Linux, OSX etc.).

Conclusion

For the Java call graph generation we will use Soot, as it can be included into the program through the usage of a Java runtime library and it can generate a complete call graph of the program that can be processed further.

For the C source code, we have chosen to use Cflow. It is easy to use for our goal as it can be run from the commandline and the output can be parsed, so it can be loaded into our tool.

GProf creates a call graph during the execution of a program and thus may not contain all edges of the call graph. CScope holds the call graph information in an internal database and thus does not allow an external program to reuse its information. Doxygen's call graphs are generated as an image which makes reuse of the information very hard.

2.2 Overview of the Approach

In this approach we will try to use call graphs to determine whether the program is written in multiple programming languages and whether remote communication through the usage of Java RMI takes place. Based on the call graph and the deducted information about the programming languages and RMI, we will generate observer, monitors, handlers and verifiers that are able to such verify programs. A *Verifier* is a piece of code that retrieves the verification data from the *Verification Data Manager* and checks the event against the verification data.

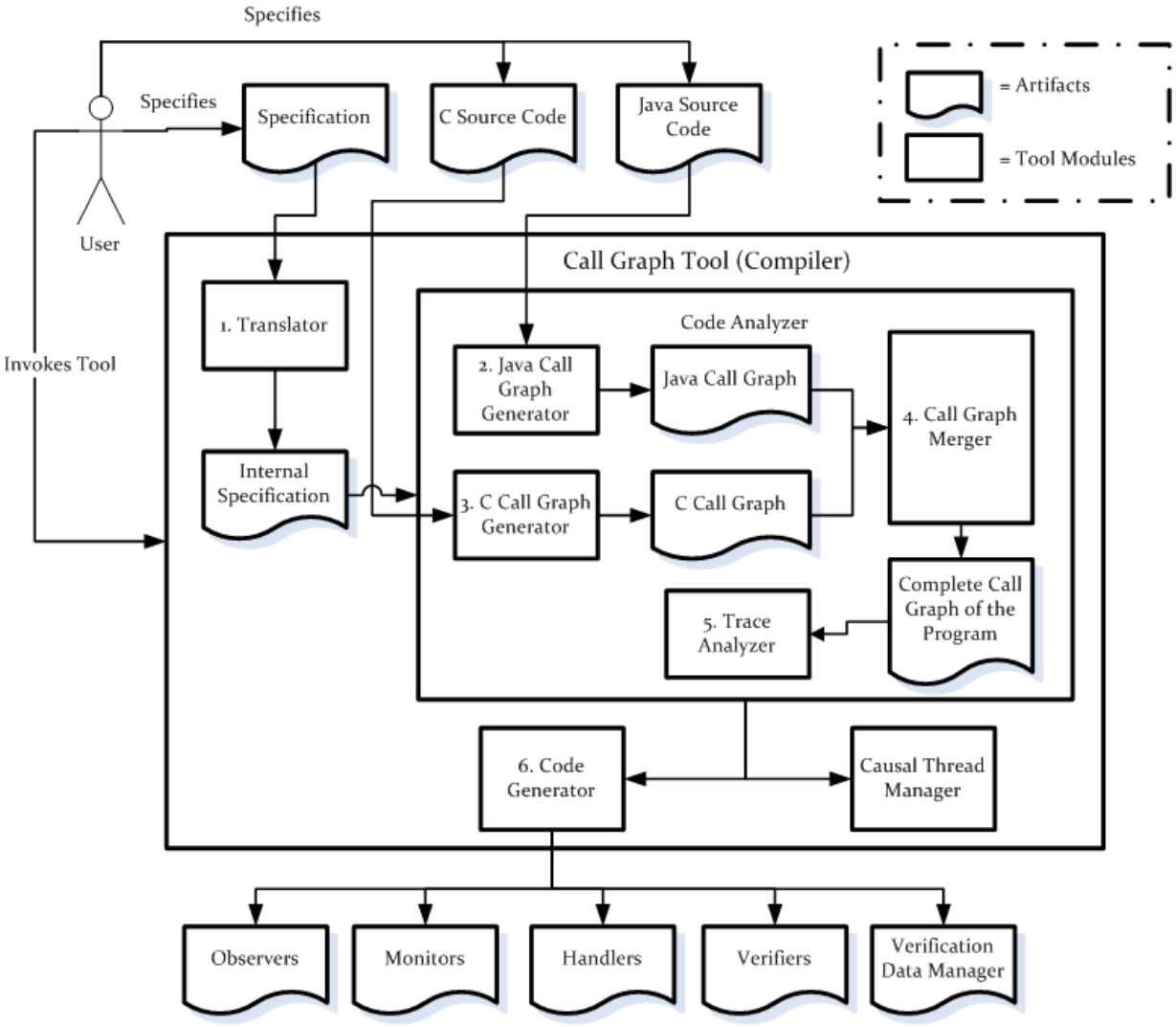


Figure 2-5: Flow of information in the call graph generation tool

The flow of information and the order of the executed tasks is shown in Figure 2-5. The inputs of the program are the specification by the user and the Java and C source code. Source code is specified by the Compose* BuildConfig.xml. As our tool will become an integrated part of the previous E-Chaser version, we can use its BuildConfig.xml file, which already contains a list of files that need to be compiled. We need to organize these files by programming language, so the files are analyzed by the proper tools. At the end of this step we provide the list of files in an organized manner for the rest of the tool.

In the first step the tool reads the specification to find the *events* and *properties* (1). The call graphs of the programs are extracted in the next two steps. For each programming language there is a separate call graph (Java (2) and C (3)). After separate call graphs are generated, they are merged into a larger call graph of the complete program (4). This call graph contains edges that

represent distributed calls (RMI) and edges that show usage of JNI. These edges need to be deduced, because they cross the programming language border and thus are not generated by the tools for Java and C source code. In order to obtain these missing edges, we need to analyze the call graph for invocations of native methods in Java and the usage of specific JNI functions in C. Through analyzing Java RMI invocations we can find distribution. RMI has a stub in the form of a method signature on an interface and the content of the method can be found at the implementing class.

In this complete program call graph must find the events in the paths of the call graph. For each path we will want to find whether it is distributed and whether it spans multiple programming languages (5). After deducting this information, we use it to generate monitors (6). In the following section, we explain each step in detail.

In comparison with the second E-Chaser's architecture (shown in Figure 1-4), there are no changes to the input and the output. The developer still defines the specification and the source code (for both Java and C). The *Translator* remains the same; it transforms the specification by the developer into an internal specification. It is explained in chapter 1. The handling of RMI code remains the same. The functionality of the *Code Analyzer* (with regards to RMI) and the *Causal Thread Manager* also remain the same. The *Code Analyzer*, however, gets extra functionality. The call graph extractor becomes part of the *Code Analyzer*, and its information is given to the *Code Generator*. The *Code Generator* will still generate the usual RMI code, but now also takes into account the structure of the program. It will try to generate one of the two architectures mentioned in Figure 2-1 or Figure 2-2, based on the information provided by the call graph extractor.

2.3 Call Graph Generator

The *Call Graph Generator* consists of three component. The *C Call Graph Generator*, the *Java Call Graph Generator* and the *Call Graph Merger*. The *Call Graph Merger* glues the Java and the C call graphs into one call graph of the whole program. In the next three sections, each component is described.

2.3.1 Call Graph Model

We use one model for both types (C and Java) of call graphs, which is shown in Figure 2-6. We do this to build a single call graph for the total program, consisting of two different programming languages. The result of the call graph generators is an instantiation of the call graph model for their respective languages (C and Java) after step 2 and 3 of the tool. After that, the C and Java call graphs are merged (or connected) at step 4.

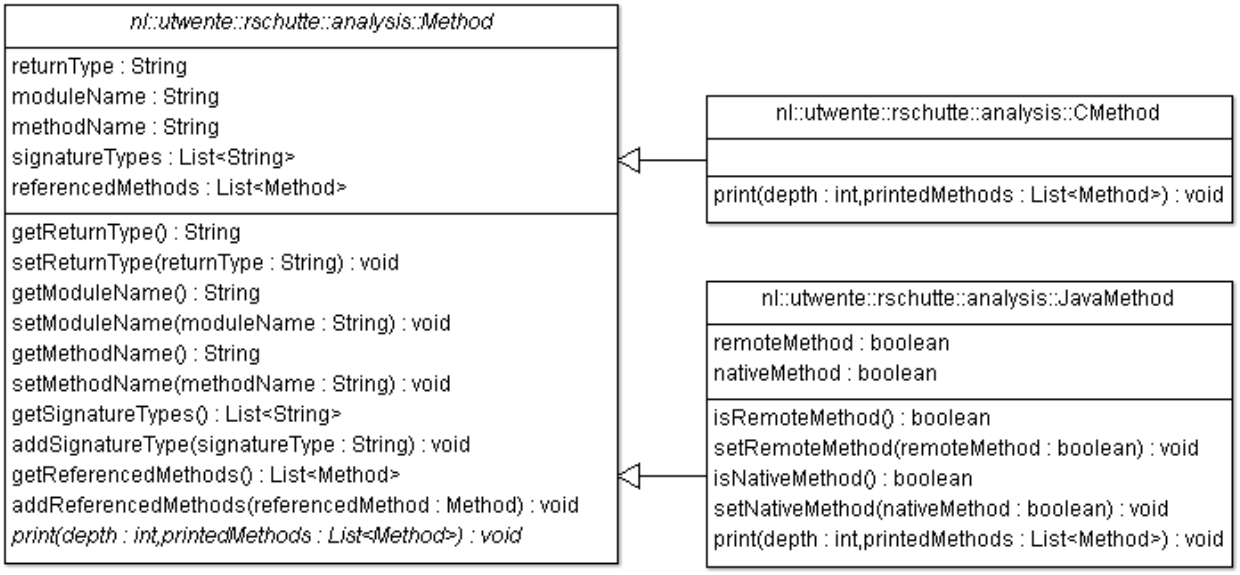


Figure 2-6: UML Class Diagram of Method, CMethod and JavaMethod classes

There is an abstract base class `Method` which holds data and methods that are applicable for most types of methods and functions. The `CMethod` and `JavaMethod` classes are realizations of the `Method` class. Both of them have specific methods and data that are being kept for their purposes. The `CMethod` only has a different print method, but the Java method also has flags for keeping track of native and remote methods. A `Method` has a set of referenced methods stored in the `referencedMethods` list, these are the edges of the call graph.

For both method types the returntype, the modulename, method- or functionname and the signature types are stored. The referenced methods list is an ordered list of methods or functions that is called from the C function of Java method.

Each Method class and its subclasses have a `print()` method. This method prints the call graph starting from this method, while excluding recursive calls. It is used to get an overview of the call graph that is generated by the tool.

2.3.2 Java Call Graph Generation

As the second step, the Java files are received by Java Call Graph Generator. It uses the Soot tool, which analyzes the program's source and extracts the call graph.

Soot allows us to specify the classes that must be included in the analysis. We then let Soot generate a call graph off all the included classes. When the classes are loaded, we can use the Soot classes explore the program's call graph from Java code. The call graph is given by Soot through classes from its API. The Java call graph generator then converts the call graph to the shared call

graph model, in which we can include both Java and C call graphs. This call graph model is given in Figure 2-6.

We do not explore a call graph solely from the main method and continue from there, since we want to get an overview of all the methods. This is necessary, because the Java source code can be used as a library from other languages, such as C. Therefore, some code may not be invoked from the main method of a program, but may still be used by source code outside of Java. Without including those Java methods, some source code may not be part of the call graph, while it is part of a multilanguage program.

Limitations of Java Call Graph Generation

Libraries
Soot is able to build a call graph of the internals of Java libraries, for example the standard JDK libraries, since it also works on compiled code. The C tools that we have reviewed can not analyze code after it has been compiled, it needs plain source code. Therefore the code in libraries can not be included in the call graphs of C programs. This shows a mismatch between the two approaches.

Analyzing the internals of libraries may be useful if we want to check the usage of certain classes of the standard Java library by the Java library itself as well as by the program that is being analyzed. A complete overview of the usage of a certain method can be given. For C this would be impossible, unless the actual source of the library is given.

Java Reflection
Method calls through Java reflection are not added to the call graph of the analyzed program. This is due to the parameterized nature of the calls. It is similar to other problems found, like C to Java JNI and function pointers. The actual targets of these calls are only known during runtime.

2.3.3 C Call Graph Generation

This chapter describes step 3 of the approach which is illustrated in Figure 2-5.

In order to get a more useful call graph we run Cflow with the following options.

- The `--number` option is used to get a number in front of each call graph node. These numbers are also used as a reference number in case of recursion or when the function has been previously explored (Cflow refers to this number).

- The `--brief` option. It is used to generate a more compact call graph, which requires less effort to parse.

- The `--reverse` option. It is used to generate a reversed call graph. A program may be used as a library and therefore it may not always start as the main function. Therefore, each

separate branch of the call graph needs to be explored. The reverse call graph makes sure any function in the program is listed, instead of just the functions that are called from the main function or from functions called indirectly from the main function.

```
 1 GetStringUTFChars():
 2     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17>
 3 Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
 4 NewStringUTF():
 5     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
 6 fclose():
 7     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
 8 fopen():
 9     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
10 fprintf():
11     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
12 strcat():
13     Java_nl_utwente_rschutte_FileStore_storeFile() <JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile (JNIEnv *jnienv,jobject obj,jstring
filename,jstring text) at nl_utwente_rschutte_FileStore.c:17> [see 2]
```

Code Fragment 2-8: Excerpt of example output of Cflow for the filestore written in C

The output of Cflow is then read by the C call graph generator. An example output of the Cflow tool is given in Code Fragment 2-8. The linenumbers are part of the output and are not just added for ease of reference. Children of function are distinguished through indentation.

Our example source code only has the `Java_nl_utwente_rschutte_FileStore_storeFile()` function, thus every root function is called by the `Java_nl_utwente_rschutte_FileStore_storeFile()` function. Only line 3 is different, it is an empty branch with just the `Java_nl_utwente_rschutte_FileStore_storeFile()` function. It shows that the function itself is never called by any other function. At the end of most lines containing a reference to the `Java_nl_utwente_rschutte_FileStore_storeFile()` function, there is a text `[see 2]`. It means that the function mentioned on that line is the same as the function on line 2. We can see it on line 3, 5, 7, 9 and 11.

We have created an ANTLR grammar to read and parse the output of Cflow, it can be found in Code Fragment B-8. The resulting Abstract Syntax Tree (AST) is then interpreted information is gathered. For a basic overview, see [43].

For a C function only the name of the function is necessary information, since it is identifying. In contrast to Java methods in which, the argument types and the defining class are also important. For a C function these are not necessary to identify a function, since a C function has to have a unique name.

The C Call Graph Generator then uses the information from the interpretation of the Cflow output to build a call graph according to the model given in Figure 2-6. It uses the same model as the Java Call Graph Generator.

Limitations of C Call Graph Generation

C Function Pointers
In C, function pointers are often used to achieve a form of abstraction, like simulating inheritance. The actual execution of a function is dependent on runtime values, which are used to select a function. Another use of function pointers is that of a callback function. An often used example is that of a sorting routine where the comparison function is provided through a function pointer.

Callback Functions
Callback functions in C are often implemented through the usage of function pointers. Therefore, they are not added to the call flow graph for the same reason as any other function pointer based function call.

In Java, callback methods are often implemented by anonymous inner classes. These can be added through the usage of the Soot tool. As such, it is a problem that is limited to C.

2.4 Merging of Call Graphs

This section describes step 4 of the approach which is illustrated in Figure 2-5 and it is performed by our call graph merger. For our current tool we use JNI as a way to communicate between Java and C (both directions), and Java RMI for distributed communication.

When the tool arrives at this step, there is a call graph, which contains nodes and edges from C and Java source code. However, they are not yet linked at RMI and JNI calls. This is what the tool attempts to do next.

2.4.1 RMI

RMI calls are denoted by the `isRemote` boolean property. It is a private variable of the `Method` class. If a method is remote, the client and server part will automatically be connected by the Java call graph generator. A method can only be declared once in a class and have only one implementation. On the client side, there will be no outgoing edges from the method, because there is no implementation. On the server side, the method is already added and outgoing edges can be added from the server implementation to the client.

2.4.2 JNI - C to Java

An example of a JNI call is given in Code Fragment 2-7. A small excerpt with the relevant lines is given in Code Fragment 2-9. We will show why the target node on the call graph of a JNI call cannot be found statically.

```
 1 int main() {
 2     JavaVM * jvm;
 3     JNIEnv* env = create_vm(&jvm);
 4
...
 5
 6     editorClass = (*env)->FindClass(env, "nl/utwente/rschutte/Editor");
 7     mainMethod = (*env)->GetStaticMethodID(env, editorClass, "main",
 8                  "([Ljava/lang/String;)V");
 9
10     (*env)->CallStaticVoidMethod(env, editorClass, mainMethod, NULL);
11
...
```

Code Fragment 2-9: C JNI code excerpt

Problem

The `create_vm(&jvm)` (line 3) call creates a JVM instance, which is omitted for brevity. The `(*env)->FindClass(env, "nl/utwente/rschutte/Editor")` function (line 6) call loads the `nl.utwente.rschutte.Editor` class. This class is then used as a parameter in the next line (line 7) which loads the actual method through using the `(*env)->getStaticMethodId(…)` call (line 7). In the next statement (`(*env)->CallStaticVoidMethod(env, editorClass, mainMethod, NULL);` on line 10) the JVM, the class and the method are used as parameters for actually executing the call. The NULL value is the value for the arguments for the function is getting called. In this case no arguments are necessary.

The exact target of the `(*env)->CallStaticVoidMethod(env, editorClass, mainMethod, NULL)` method call (line 14) can not be determined, since the values of `editorClass` and `mainMethod` cannot be statically determined. Their value is only known during runtime.

To still find the target of the function call there are several options available. We will discuss these next.

Program Slicing

Through program slicing one can determine the possible values of variables. A program slice is a set of statements from a program that influences the value of a chosen point of interest, for example a variable. A slice can be used to determine the values of the class and the method that is being invoked from C to Java. [42]

To find the value in the example of `editorClass` the possible values of the second parameter needs to be determined. The second parameter provides enough information about the actual class being loaded. The content can be used to deduce the fully qualified class name in Java. Once we can deduce the value of the `editorClass` variable we can use this to deduce the value of `mainMethod` variable.

The `mainMethod` variables content is based on three parameters. The second parameter is the class (in this case the `editorClass` variable) to find a method from. The third is the name of the method being called and the fourth which contains the parameters. The third and fourth variable both need to be deduced. If these three values are deduced the potential methods in the Java program can be found.

The advantage of this approach is that it is relatively precise. The set of potential methods can in most cases be severely limited even though there may still be some methods in the set that are not invoked. The disadvantage is that this approach is relatively expensive to compute and to implement.

Program Execution

A possibility would be to actually execute the program with additional code that analyzes the target of the JNI invocation. During execution information is retrieved for every instance of JNI invocation code.

The advantage of this method is that only methods are found that are indeed possible targets. The set of potential candidates is not polluted with methods that can never be invoked.

The main disadvantage is that there is no certainty about the completeness of the list of possible targets. There may be target methods that are never invoked during runtime, because of the reliance on unknown variables such as user input. All possible executions of the program may

need to be performed to form a complete list of candidate methods. This option also departs from the way that the rest of the program is analyzed, through static analysis.

JNI Method Name

Through analyzing the build-up of the name for the JNI methods we can deduct certain information about the actual method that is invoked. Example JNI methodnames are `CallIntMethod`, `CallNonvirtualObjectMethodV` and `CallStaticFloatMethodA`. The method names can be matched by the regular expression shown in Code Fragment 2-10. A complete list of the JNI methods can be found at [13].

```
Methodname ::= "Call" ["Static" | "Nonvirtual"]? <Type> "Method" ["V" | "A"]?
Type       ::= ["Object" | "Boolean" | "Byte" | "Char" | "Short" | "Int" |
                "Long" | "Float" | "Double" | "Void"]
```

Code Fragment 2-10: JNI function name build-up

The keyword `Static` means that a static method is being invoked. The `Nonvirtual` keyword means that the method is not invoked through the normal inheritance hierarchy and the actual method that is pointed to should be invoked, not an overriding submethod. The `<Type>` part is the return type of the method that is invoked. The last letter `V` or `A` denotes how the parameters are passed to the Java method and is of no consequence for the Java side of the invocation. However, if both `V` and `A` are missing it means that all the arguments have to be listed in the function call after the method id. If this is the case, the number of arguments can be measured.

Through this analysis we can find out whether the invoked method: is static, has a return type and what that type is, is called virtually or non-virtual and if it is invoked with arguments. If it is invoked through a list of arguments with a *"varargs"*-type invocation, the number of arguments can be determined. By taking these properties of the potential methods into account, all methods that do not match these criteria can be filtered out of the list of potential target methods.

A benefit of this method is that it leaves no methods out that may be potentially invoked. A disadvantage is that this method only narrows down the options and does not give a definitive answer. In some scenarios this approach may still yield a large amount of methods.

No Fix

Another option would be to not fix the problems posed by the JNI calls from C to Java. Each invocation of Java from C would be an outgoing edge from the C function in which the invocation is made to every Java method. The implication would be that it severely limits the use of the call graph. We may be able to use the separate parts of the call graphs that are extracted from the source code, but our tool would need to be adapted to this. Also, the call graph may not be easily

used by future projects as they may consider the call graph to be too unreliable to be used for further development.

Conclusion

We have not implemented any of the options discussed. The program slicing approach is too difficult and too large to implement for this thesis. The program execution option only provides edges that are actually executed. As such, not all edges may be explored. The analysis of the JNI function name leaves in many possible Java methods, which creates a call graph with many edges that do not exist. Not fixing the call graph and merging every Java invocation from C with every Java method is considered worse than the analysis of the JNI function name.

2.4.3 JNI – Java to C

The other direction of communication in JNI is from Java to C. To connect the graph, there are two steps that need to be taken. First we derive the C function name from the Java method signature and the second step is to connect the Java method and C function in the call graph.

Deriving C Function Name

To connect nodes from Java to C, we can match the name of the Java method stub with the C skeleton. C JNI headers are generated by the `javah` tool. Because the tool has consistent results, we can derive the name of the C function that is being called from the Java native method and we can use it to link a Java JNI call to a C function directly.

The name of the JNI function is deduced from the Java method name. A C JNI function name starts with "`Java_`". Appended to that is the Java package name in which the dots are replaced with underscores. The last part of the C function name is the actual Java method name; it is added with an underscore prefixed.

The arguments of a JNI C function are also derivable. The first argument is always the `JNIEnv` interface through which the JVM instance from which the JNI call is made can be found. The second argument is the Java object instance on which the JNI method was called. The remaining arguments are the arguments of Java method. Java primitives and Strings are converted to their JNI equivalents and Java objects are converted to the `jobject` type.

If there are JNI methods that are overloaded then the C functions get a different signature, since they would no longer be unique. The C functionname is generated in the same way as for non-overloaded functions, except that Java internal argument types are appended to the functionname to make it unique again. For an example, see Code Fragment 2-11. Two underscores are appended to the functionname as previously generated. After that each argument type is added (e.g. "I" for integer, "Z" for boolean etc.). Full class names are again separated by underscores instead of dots and preprended by an "L". Arraytypes are prepended by a "[".

```
JNIEXPORT jstring JNICALL Java_rik_Main_testMethod__Ljava_lang_String_2I
  (JNIEnv *, jobject, jstring, jint);
```

Code Fragment 2-11: Overloaded Java method's JNI signature

Connecting the Edges

Because the C function name can be determined from the Java method name, we can check every Java method in the Java call graph and check whether it has the *"native"* qualifier. It is stored in the `JavaMethod` object in the `nativeMethod` boolean. For such a method, we can find the C function name. This function is also available in the call graph, but as a `CMethod` instance. Since the name of a function is unique in C, we can easily determine the `CMethod` that needs to be added to the `referencedMethods` list of the Method object.

2.5 Trace Analyzer and Code Generator

Step 5 (Trace Analyzer) and 6 (Code Generator) of Figure 2-5 cannot be executed, since we have found that we were not able to connect JNI calls (Java invocations in C) from C to Java at step 4, because they rely on runtime values. A complete call graph of the program can therefore not be generated. We have shown alternatives and we have decided not to use them, because they are unfeasible, or too imprecise.

2.6 Conclusions

The Java and C language combination cannot be supported. More generally, any two languages that depend upon a layer that is dependent upon runtime variable values cannot be determined by a call graph approach. A different approach to verifying multilanguage and distributed program needs to be found which takes into account layers with these characteristics. As a result, we have only implemented step 2 (extracting a call graph from Java source code), 3 (extracting a call graph from C source code) and only a part of step 4 (merging of the call graphs in Java and C) of this solution.

As we have shown, call graph extraction by tools is problematic with respect to more advanced program structures, such as C function pointers and Java reflection. Another problem for our specific JNI case is related to the JNI interface itself, as it determines the target of the call based on variables of which the value is only known during runtime. For the JNI limitations we have shown alternatives, however they were not precise enough. Either too many edges would be included or some edges could be missing. We deemed the call graph with its limitations to be too imprecise to reach conclusions that can be used to generate monitors for multiple language and distributed programs.

3 Solution 2: Outline Monitors

As we have shown in Chapter 2, we were unable to extract the call graph of a program which is precise enough. This chapter explains an alternative solution to verify distributed programs written in multiple languages. Again we focus on programs written in C and Java and using RMI and JNI.

In this chapter we will give background information, explain our approach and the steps necessary to reach our goal.

3.1 Goal

The goal of this solution is again to verify programs that are distributed and written in multiple languages, but the approach is different. We will generate an architecture similar to the architecture of Figure 2-2, but with a few changes. Again the specification is: 1) distribution transparent and 2) language independent.

In the previous solution we could not determine the call graph of a multilanguage and distributed program. In this solution we will design a verification system that works independently of the distribution of the program.

3.1.1 Overview of the Approach

The architecture of the new E-Chaser version is given in Figure 3-1. The input is again the *Specification*, the and the *C* and *Java Source Code*. The source code is provided through the BuildConfig.xml file of the Compose* compiler. The output again consists of *Observers*, *Monitors*, *Handlers* and *Verifiers* for both Java and C, and the *Verification Data Manager*.

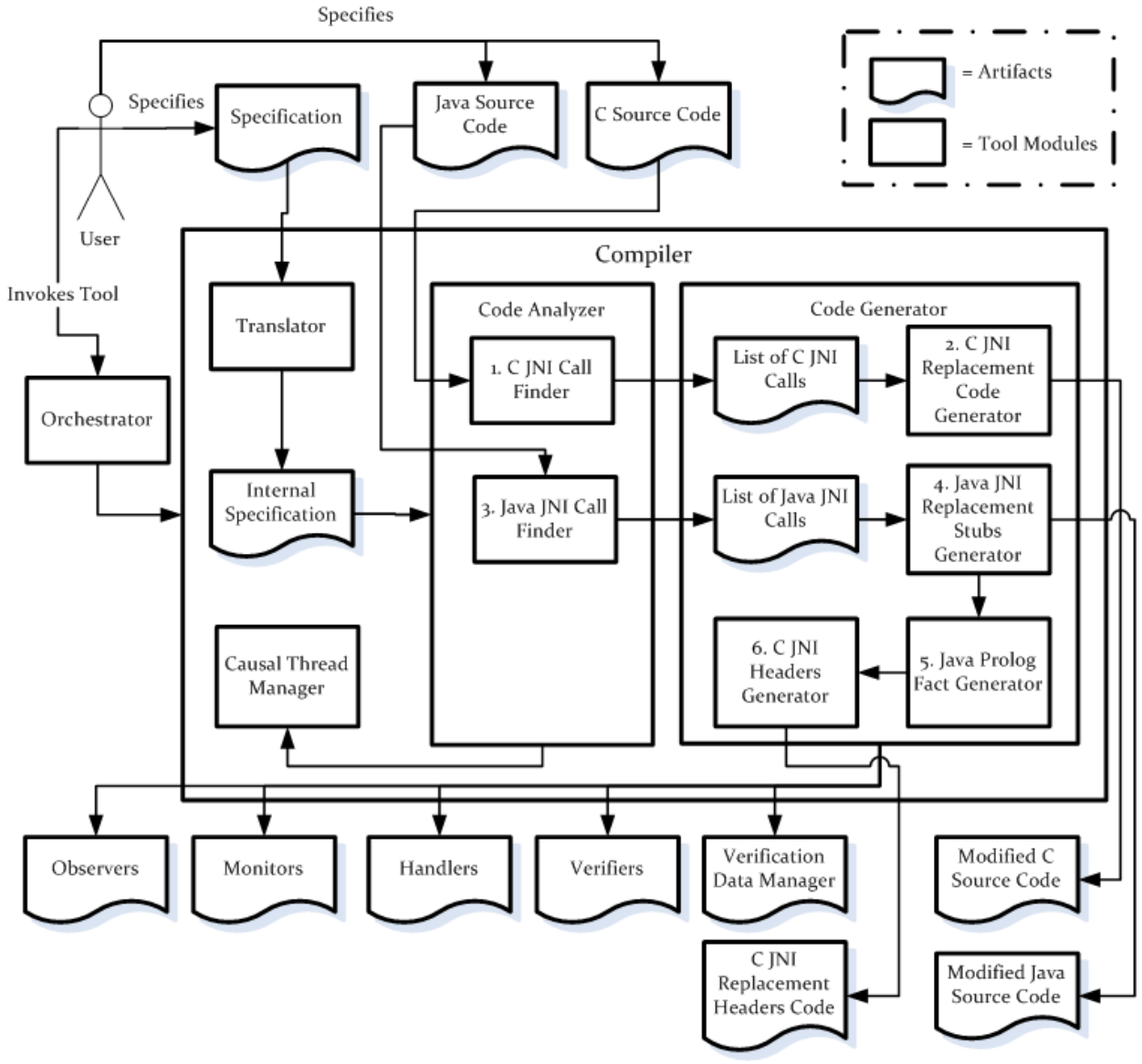


Figure 3-1: E-Chaser's architecture for the proposed solution

The *Specification*, *Translator* and *Internal Specification* are the same as the original E-Chaser as well as the previously proposed solution. There are two parallel paths for the C (steps 2 and 3) and Java source code (steps 4 - 7). The C path takes the *C Source Code* as input and outputs *Modified C Source Code*. The Java path takes *Java Source Code* as input and outputs *Modified Java Source Code* and *C JNI Replacement Headers Code*.

The C path consists of the *C JNI Call Finder* and the *C JNI Replacement Code Generator*. The *C JNI Call Finder* becomes part of E-Chaser's original *Code Analyzer*. It finds all the C JNI calls and provides them to the *C JNI Replacement Code Generator*, which is part of the *Code Generator*. The

C JNI Replacement Code Generator creates parameterization code which takes the place of the original JNI code. The result is *Modified C Source Code*.

The Java path consists of the *Java JNI Call Finder*, which passes a list of Java JNI calls to the *Java JNI Replacement Stubs Generator*. It is the only new module for Java in the *Code Analyzer*. The other steps are all part of the *Code Generator*. The *Java JNI Replacement Stubs Generator* creates Java JNI stubs that replace the original stubs, which results in *Modified Java Source Code*. Next, the *Java Prolog Fact Generator* creates Prolog facts and stores them in the Compose* prolog fact database. The *C JNI Headers Generator* creates the content, which is C source code, for the JNI stubs generated by the *Java JNI Replacement Stubs Generator*, which results in *C JNI Replacement Headers Code*.

The E-Chaser functionality regarding the usage of RMI stays the same as in the previous version of E-Chaser. The *Causal Thread Manager*, the *Code Analyzer* and *Code Generator* keep the original RMI functionality. The *Code Analyzer* and *Code Generator* are extended with the new functionality.

Steps 1 and 2 are related to C to Java JNI calls. These steps are described in chapter 4. The steps 3 - 6 are related to Java to C JNI calls and are described in chapter 5.

3.1.2 Orchestration

We create a single interface for the user to communicate with, to make the tool easier to use. We call it the *Orchestrator* module, see Figure 3-1. The interface orchestrates the C to Java and Java to C compilers. It also takes care of the dependencies for compilation and execution of the program. The separate binaries of the C and Java code also need to be configured and started with the right parameters to be executed.

3.1.3 New Program Flow

We want to parameterize the original JNI calls. Java methods with a native implementation can be found, because their signature contains the keyword “native”. JNI calls are statements that invoke a method with a native implementation. In C we can find calls that invoke Java methods, because there is a static list of JNI invocation functions through which Java methods can be called. Usage of such a function indicates a JNI call.

The resulting program’s architecture after the changes that are made by the tool is abstractly pictured in Figure 3-2. The tool generates a *Caller-Side Proxy* and a *Callee-Side Proxy*. The *Caller-Side Proxy* intercepts the *Original JNI Call*. After intercepting the *Original JNI Call*, the *Caller-Side Proxy* parameterizes the *Original JNI Call*. It then invoked the *Generated JNI Call*. The arguments of the *Generated JNI Call* are the parameterized *Original JNI Call* and the causal thread id. The causal thread id is an identifier for the causal thread in the program. The *Callee-Side Proxy* is the

receiver of the *Generated JNI Call*. It stores the causal thread id and continues the original call flow, by restoring the original JNI call from the parameters of *Generated JNI Call* and invoking it.

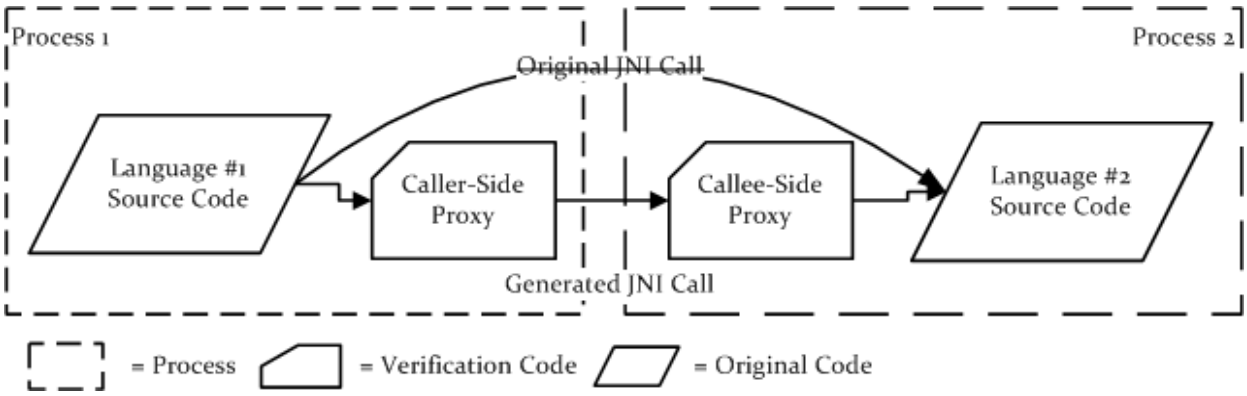


Figure 3-2: Call flow interception approach

3.1.4 Program Architecture

The architecture of the generated program is shown in Figure 3-3. The *RMI Calls* and *JNI Calls* include both the original and the newly created calls.

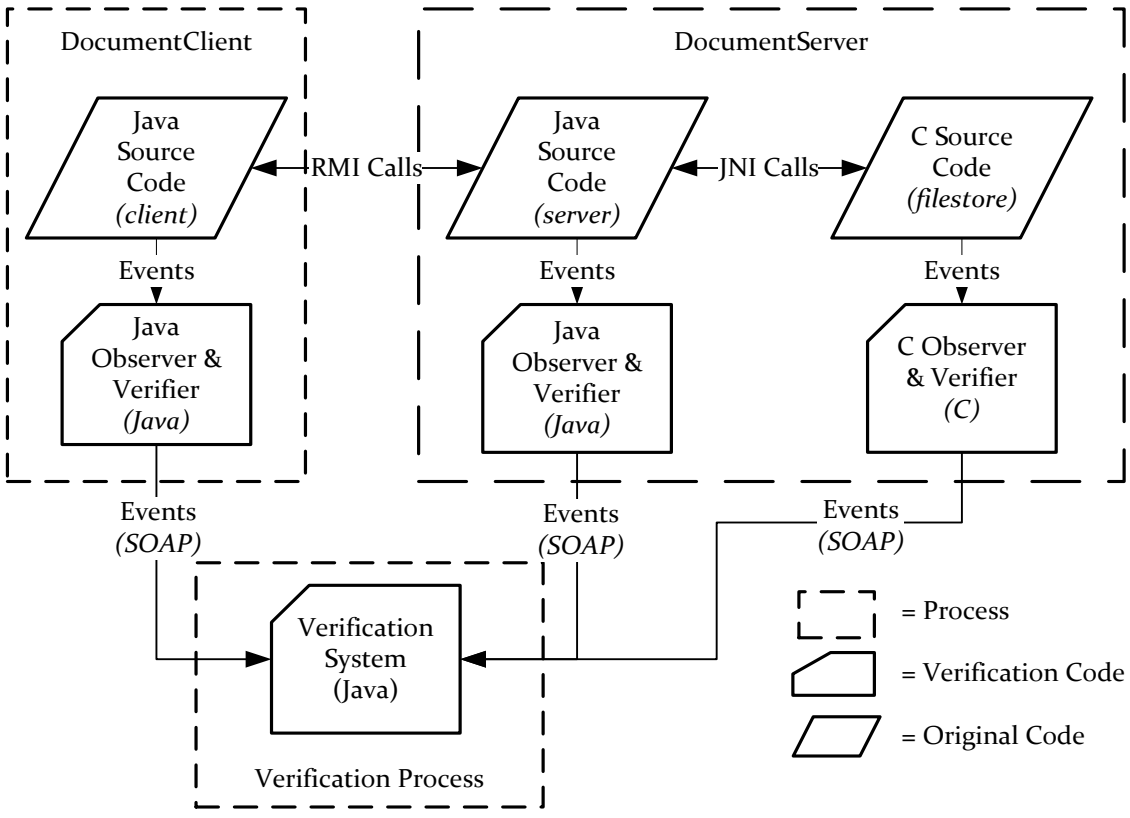


Figure 3-3: Architecture of a distributed program with external verification process

The *Java Observer & Verifier*s again retrieve the verification data from the *Verification System* and verify the generated event. The *C Observer & Verifier* does the same for C code. One of the changes in the architecture compared to Figure 2-2 is the direct communication of C source code (the *C Observer & Verifier*) with the remote *Verification System*. This can be done, because we have chosen to remove the usage of RMI from the verification system and chosen SOAP for the communication between the *Verification System* and the *Verifier*s.

3.2 Causal Thread Storage

The causal thread is stored and retrieved inside the Caller and Callee-Side Proxies. For each supported programming language (C and Java) different storage code is written. The causal thread id needs to be globally available. In Java we can use a static field or a singleton object. In C we can use a global variable.

For the causal thread storage to be able to support multiple threads of execution, the generation, storage and retrieval of the causal thread id needs to be thread safe. The implementation can use the `ThreadLocal` class of the Java API to achieve this.

3.3 Background

For the previous solution, we have described how we could find RMI and JNI calls, the same applies to this solution.

3.3.1 Function & Method Call Interception

As part of the original functionality of Compose*, it can intercept method and function invocations in the base program. This functionality can be reused by our tool. To intercept a JNI method or function call, prolog facts can be added to the repository. These prolog facts will instruct the compiler to generate code that will reroute invocations of these JNI methods and function calls to special handling code. The information necessary to achieve this are the method- or functionname that needs to be intercepted and the class or file in which the method or function is defined. Both can be retrieved from the source code and the file in which the source code is written.

3.3.2 Language Independent Protocol

The communication between the central *Verification System* and the *Verifier*s uses a language independent protocol, because the *C Observer & Verifier* cannot communicate through Java RMI. Many potential protocols are available and we have chosen SOAP as it is a widely documented protocol for exchanging information. It has libraries for many programming languages. A list of languages and libraries are given in Table 3-1.

Language	**Libraries**
Java	Apache Axis
C	gSOAP
C++	gSOAP, Apache AXIS
C#	ASP.NET Web Services

Table 3-1: SOAP libraries for several programming languages

Due to this change, the *Verification System* can be written an different language than Java. As such, execution environments that do not support Java can still be verified. The dependency on Java for every execution environment has been removed.

4 Maintaining the Causal Thread of Execution from C to Java

In this chapter we will describe how C to Java JNI calls are handled by our tool. These are steps 1 (*C JNI Call Finder*) and 2 (*C JNI Replacement Code Generator*) of Figure 3-1. Steps 1 and 2 together generate the *Caller-Side Proxy*. However, first we will explain the *Callee-Side Proxy*, which is a Java class that is the same for every Java program that is being verified, because it is the target of every parameterized JNI call from C to Java.

4.1 Compose/CwC Problems

Compose/CwC is the Compose* implementation for C. We were unable to use Compose/CwC for our implementation as the current version has compilation problems. For this reason, the control flow of the program is changed, by modifying the original source code and replacing the JNI calls instead of intercepting the call flow.

4.2 Overview

In this chapter we describe a tool that changes the C source code, so that the causal thread id is retained for JNI calls from C to Java. The overview of the tool's effect is depicted Figure 4-1.

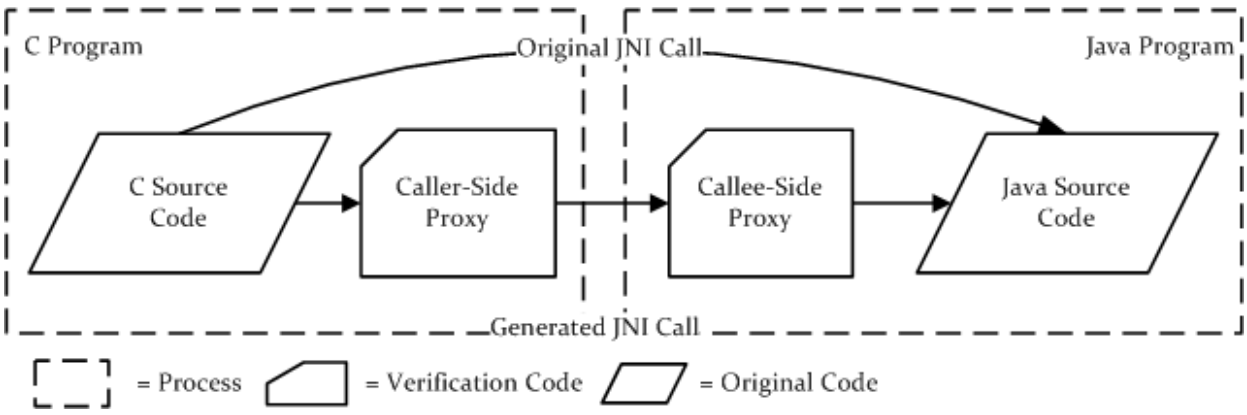


Figure 4-1: New control flow from C to Java

As shown, the *Original JNI Call* is removed from the program. It is replaced by the *Caller-Side Proxy*, which does a call to the *Callee-Side Proxy* (which is the *Java JNI Hook*), from which the original call flow continues. The new JNI call also holds the causal thread id.

The *Caller-Side Proxy* has to be generated based on the original JNI call.

4.3 Callee-Side Proxy (Java JNI Hook)

The *Java JNI Hook* is the implementation of the *Callee-Side Proxy*. It is a Java class, which is inserted in every Java program that is being verified by our tool. It receives the generated JNI function call. The *Java JNI Hook* receives the *Caller-Side Proxy*'s call and gets the causal thread id and the parameterized method call (target object, target method and parameters) as parameters. The *Java JNI Hook* stores the causal thread id and invokes the parameterized method call through Java's reflection API.

On the C side of an invocation of a method on the JNI hook is an ordinary JNI function invocation. An example of such code is given in Code Fragment 4-1, see line 9.

```
 1 JavaVM * jvm;
 2 JNIEnv* env = create_vm(&jvm);
 3
 4 jclass editorClass;
 5 jmethodID mainMethod;
 6
 7 editorClass = (*env)->FindClass(env, "nl/utwente/rschutte/Editor");
 8
 9 (*env)->CallStaticVoidMethod(env, editorClass, (*env)->GetStaticMethodID(env,
editorClass, "main", "([Ljava/lang/String;)V"), NULL);
10
11 (*jvm)->DestroyJavaVM(jvm);
12
13 return 0;
```

Code Fragment 4-1: Example JNI invocation from C to Java

The first parameter is the JNI environment. The Java hook class and method (dependant on whether it is static or not, as described earlier) need to be added as the second and the third parameter. The fourth parameter is the causal thread id which is saved in a standardized place. The fifth parameter is actual object or class (when invoking statically) on which the method is called. The sixth parameter is the reflected method which needs to be called. The seventh parameter is a `jobjectArray` with all the original function parameters converted to Java types.

The class `JNIHook` is the implementation of the *Java JNI Hook*. For each method on the `JNIHook` class the first argument is the causal thread id (named "causalThreadId"). This thread id holds the information about the causal thread of the application. The second argument is the actual object instance that the method needs to be invoked on. The third argument is a list of arguments for the original JNI method call. These will be passed to the original method call through Java's reflection API. For each Java return type a hook method is available.

Each method also has a static variant. The second argument of this type of function is the class on which the method is called instead of the actual object instance.

The implementation of the `objectJniHook` method is given in Code Fragment 4-2.

```
 1 public static Object objectJniHook(String causalThreadId, Object object, Method
method, Object[] args)
 2 {
 3     Object result = null;
 4
 5     try {
 6           result = method.invoke(object, args);
 7     } catch (Exception e) {
 8           e.printStackTrace();
 9     }
10
11     return result;
12 }
```

Code Fragment 4-2: Implementation of the objectJniHook method

These methods use the reflection API of Java to call the original target method as it was parameterized on the C side of the communication.

The `objectJniHook` and its static equivalent `staticObjectJniHook` are also called by any other hook method that has a return type. This is possible by casting the result to the proper type. An example is the `intJniHook` shown in Code Fragment 4-3.

```
1 public static int intJniHook(String causalThreadId, Object object, Method method,
Object[] args)
2 {
3     Integer result = (Integer) objectJniHook(causalThreadId, object, method, args);
4     return result;
5 }
```

Code Fragment 4-3: Implementation of the intJniHook method

The usage of the `objectJniHook` is shown in line 3. The result is also casted on the line to an Integer object, which is returned on line 4.

4.4 C JNI Call Finder

The C JNI Call Finder does two things. First it finds JNI calls and then it analyzes them. It extracts information from the source code that is necessary for generating the replacement code.

4.4.1 Finding JNI Calls

There is a set list of functions which can be used to invoke Java from C in the JNI API. Through analyzing the name of a C function call we can deduce if it is an invocation of a JNI function. Example JNI function names are `CallIntMethod`, `CallNonvirtualObjectMethodV` and

`CallStaticFloatMethodA`. The function names can be matched by the grammar in Code Fragment 4-4. A complete list of the JNI functions can be found at [13].

```
Methodname ::= "Call" ["Static" | "Nonvirtual"]? <Type> "Method" ["V" | "A"]?
Type       ::= ["Object" | "Boolean" | "Byte" | "Char" | "Short" | "Int" |
                "Long" | "Float" | "Double" | "Void"]
```

Code Fragment 4-4: Grammar of the JNI method names in C

4.4.2 Extracting Information

From the name of the C JNI function name that is used we can deduce information about the Java method that is invoked.

The keyword `Static` means that a static method is being invoked. The `Nonvirtual` keyword means that the method is not invoked through the normal inheritance hierarchy, but that the actual method that is pointed to is invoked, not an overriding submethod. The `<Type>` part is the return type of the method that is invoked. The last letter `V` or `A` denotes how the parameters are passed to the Java method and is of no consequence for the Java side of the invocation.

Through this analysis we can find whether the invoked method: is static, has a return type and what that type is, is called virtually or non-virtually and if it is invoked with arguments. If it is invoked through a list of arguments with a *"varargs"*-type invocation, the number of arguments can be determined.

4.5 C JNI Replacement Code Generator

The C JNI Replacement Code Generator gets its JNI calls information from the C JNI Call Finder. For each of the JNI calls, replacement code needs to be generated. When generating replacement code, we need to take into account the original JNI call. The original JNI call needs to be replaced with a proper JNI call to the *Java JNI Hook*.

We need to pass the parameters properly. Primitive types need to be converted to their Java wrapper types. In the `jvalue` array and `va_list` type of invocation, the parameters need to be retrieved from a data structure.

4.5.1 JNI to Java JNI Hook Matching

There are two different types of JNI function calls: Static and not static. The static method call has a class as second argument. For the not static function the second argument is an object.

The JNI function calls to Java are replaced with a function call to the JNI hook. The list of JNI hook replacements are given in Code Fragment 4-4. The original JNI function call is on the left and the replacement JNI hook method is on the right.

JNI Function Call	**Java JNI Hook**
CallVoidMethod	jniHook
CallObjectMethod	objectJniHook
CallBooleanMethod	booleanJniHook
CallByteMethod	byteJniHook
CallCharMethod	charJniHook
CallShortMethod	shortJniHook
CallIntMethod	intJniHook
CallLongMethod	longJniHook
CallFloatMethod	floatJniHook
CallDoubleMethod	doubleJniHook

Table 4-1: JNI function call and Java hook matches

The `<functionname>V()` and `<functionname>A()` functions are replaced by the same function as the normal replacement function (without the appended `V` and `A`). Parameterization code is needed for every JNI call to pass the parameters of the original JNI call, as objects to the Java hook method.

For static JNI functions the Java JNI Hook method name that is used is the same as in the table except with that the method name is prepended with `static`. For example: `staticObjectJniHook` instead of `objectJniHook`.

4.5.2 JNI Call Replacement Code Generation

We will now show for the three types of the JNI method calling functions the replacement code. In each example that follows, we only need to analyze the code of the line with the actual JNI invocation. The replacement code needs access to the variables used in this call. The original and replacement code can be found in the appendix.

Replacement Invocation

Each invocation is replaced by an invocation to the *Java JNI Hook*. The methods used to invoke the Java JNI Hook are listed in Table 4-1. An example of a replacement invocation is given in Code Fragment 4-5.

```
(*env)->CallStaticObjectMethod(env, hookClass, hookMethod, threadID, targetObject,
targetMethodObject, paramObjectValues);
```

Code Fragment 4-5: Example replacement invocation

There are six arguments to the function invocation. The first is the JNI environment in which the program runs. It can be taken from the original JNI call, as it is the first argument of that call also.

The second argument is the `JNIHook` class. The third argument is the method on the `JNIHook` class that is used. These two are constant. The causal thread id can be retrieved from a standardized location in the C program.

The `targetObject` argument is the second argument of the original JNI call. It is the object or class on which the original call was placed. The sixth argument is the `java.lang.reflect.Method` instance of the method that was originally invoked. Through reflection the `Method` object can be found. The seventh argument is the `paramObjectValues` argument. It is a `java.lang.Object[]` which is constructed for each type of JNI invocation. Each of these conversions is described in more detail.

Varargs

Code Fragment B-1 shows what the ordinary JNI call looks like. First a target object is instantiated (lines 1-6). Then the `retrieveConcatenation` method is retrieved (line 9). It is a testing method that simply concatenates the two given values and returns it. Next, there are two arguments constructed of the JNI types `jint` (a java primitive integer) and a `jstring` (an instance of `java.lang.String`) (lines 11 and 12). After that the function `CallObjectMethod` is used to call the Java method (line 15).

Code Fragment B-2 shows what the generated replacement code looks like. This code is placed at the location of the actual JNI invocation in Code Fragment B-1 (line 15). A static causal thread id is introduced for the calling of the JNI hook (line 1). In the actual code a dynamic causal thread id will be loaded. First the hook class is loaded and the proper Java function that replaces the original JNI call (lines 4 and 5). We explained earlier how the matching between the original JNI function call and the Java JNI hook is executed. Since the original class of the object is not used in the original JNI call, we retrieve is through the `GetObjectClass` function (line 8). Then we use the class to retrieve the reflected method (line 10). Next, we create a `jobjectArray` to contain the arguments as the last argument of the JNI hook is of the type `Object[]` (line 13). The size of the

array can be determined from the original JNI code; it is the original amount of arguments for the JNI call minus three (JNI interface, object/class and method). The first argument of the original JNI call was of the type `jint`, therefore it needs to be converted to a `java.lang.Integer` to be able to store it in an `Object[]`. In lines 15 - 23 the parameter types are determined. After the conversion of the parameter types (lines 25 - 112) we can call the JNI hook with the causal thread id, target object, target method and arguments array as arguments (line 115). The `CallStaticObjectMethod` function usage is deduced from the `CallObjectMethod` of the original JNI call.

To convert the parameter types a large if-else construct is used which checks each option (lines 25 - 112). A single helper function could not be used, since the actual type of the argument is not known to our compiler. However, the normal C compiler does know the type and checks for it. The conversion of the `double` and `float` types show this most clearly as it actually copies a part of the memory. The value is stored in a `jvalue` type and we reuse the `jvalue` conversion functions for the conversion.

Jvalue array

A `jvalue` array can be converted into an appropriate `jObjectArray` object by a helper function, which will be explained later. An example of a JNI invocation with a `jvalue` array as argument is found in Code Fragment B-3. The replacement code is given in Code Fragment B-4.

In the original code, in Code Fragment B-3, we first construct an instance of our test class (lines 1 - 4), load the testing method (line 6), create the `jvalue` array to store the arguments (line 9 - 20) and then invoke the Java method (line 23).

In the replacement code, shown in Code Fragment B-4, most of the work is done in the `convertJvalueArrayToJobjectArray` function. It is a function which converts the `jvalue` array to a `jobjectArray` object. We have a static causal thread id, which we can later replace with a dynamic causal thread id (line 2). The replacement code first gets the reflected method of the original JNI invocation call (lines 5 and 6). Next, it converts the `jvalue` array to a `jobjectArray` object (line 8), based on the reflected method and the original arguments. After that, it loads the hook class (line 11) and hook method (line 12) and subsequently invokes it with the proper arguments (line 15).

Va_list

The code in Code Fragment B-6 is called by the `testVAList` function call shown in Code Fragment B-5. It shows how a variadic list call is made in C and how a `va_list` is instantiated. An example JNI invocation through the variadic function option of JNI is shown in Code Fragment B-6. Its replacement code is shown in Code Fragment B-7.

In the original code we first construct an instance of our test class (lines 6 - 8), load the testing method (line 11), set up the `va_list` (line 14) and invoke the Java method with the `va_list` argument (line 15).

In the replacement code most of the work is done in the `convertVAListToJobjectArray` function (line 17). It is a helper function which converts a `va_list` to a `jobjectArray`. We create a static causal thread id (line 2) and we get the reflected Java object of the target method that is called in the original code (lines 5 and 6). Next, we load the JNI hook class and the appropriate JNI hook method (lines 9 and 10). After that the `va_list` is converted to a `jobjectArray` (lines 15 - 18). At the end, we invoke the original method through our JNI hook (line 21) and we check for an exception that may be thrown (lines 24 - 26).

Around the conversion of the `va_list` to a `jobjectArray`, that is the call of the `convertVAListToJobjectArray` function, the macros `va_start` and `va_end` are used (lines 16 - 18). These macros start and end the modification and traversal of a `va_list`. The implementation of these macros is compiler and architecture specific.

4.6 Helper Functions for Parameter Conversion

The helper functions are used by the *C JNI Replacement Code Generator*, so that the amount of code that needs to be generated is smaller. The helper functions can be reused.

The helper functions often use the JVM to gather extra information on the data that needs to be converted. Examples of it are the `java.lang.Class` instance of a type or the size of an array. For every helper function, the first argument is the JNI environment in which the program runs.

4.6.1 Invocation Types

As mentioned before, there are three invocation types possible on the JNI API and they each have a different way of passing the arguments. These are the varargs, the jvalue array and va_list invocations. For the varargs invocation, each arguments needs to be converted to a Java type and added to the `jobjectArray` that will be passed to the *JNI Hook*. For the jvalue array invocation, each `jvalue` needs to be converted to a Java type and also added to the `jobjectArray`. To do so, we use the `jvalueArray` conversion function (`convertJvalueArrayToJobjectArray`). A `va_list` is converted to a `jobjectArray` by the variadic list conversion function (`convertVAListToJobjectArray`).

4.6.2 Primitive Conversion

All JNI types can be reused (including array types) from the original JNI call, except in cases where primitive types are passed. In such cases, we need to convert primitive JNI types to their Java

wrapper types to call the JNI hook. The conversion takes place according to Table 4-2. A list of JNI types and their hierarchy can be found at [14].

JNI type	**Java type**
jboolean	java.lang.Boolean
jbyte	java.lang.Byte
jchar	java.lang.Char
jshort	java.lang.Short
jint	java.lang.Integer
jlong	java.lang.Long
jfloat	java.lang.Float
jdouble	java.lang.Double

Table 4-2: JNI primitives and their equivalent Java wrapper types

A `jboolean` gets converted to an instance of the class `java.lang.Boolean` which is of type `jobject` in C. The jobject type can be stored in a `jobjectArray`, which is necessary for our JNI hook. The autoboxing feature of Java will resolve the objects to Java native types when necessary.

4.6.3 JValue Conversion

There is another helper function named `convertJvalueToJobject`. The function signature is given in Code Fragment 4-6. The conversion only takes place when necessary. If the third argument is already a `jobject` it is returned in the same state as it was passed.

```
1 jobject convertJvalueToJobject(JNIEnv* env, jobject paramType, jvalue paramValue)
```

Code Fragment 4-6: Signature of the convertJvalueToJobject function

The second argument is the type to which the `jvalue` should be converted. This is the Java `java.lang.Class` instance that represents the type that the third argument should be converted to. Through an if-else construct the right conversion function is found.

4.6.4 JValue Array Conversion

Another helper function is the `convertJvalueArrayToJobjectArray` function. This function converts a given `jvalue` array to a `jobject` array. This helper is used for the JNI invocations that use a `jvalue` array to pass the arguments. The signature is given in Code Fragment 4-7.

```
1 jobjectArray convertJvalueArrayToJobjectArray(JNIEnv* env, jobject
targetMethodObject, jvalue jvalues[])
```

Code Fragment 4-7: Signature of the convertJvalueArrayToJobjectArray function

The second argument is the `java.lang.reflect.Method` instance of the targeted method. The third argument is the original `jvalue` array that is passed to the JNI call (represented by the second argument).

Through the `java.lang.reflect.Method` instance, more information about the target method can be found, such as the parameter types of the method. These types are used to invoke the `convertJValueToJobject` function. Each parameter type is separately converted. The size of the parameter array is found through the `java.lang.reflect.Method` instance, so we know how many parameters we need to convert and what array positions are used.

This function uses the `convertJvalueToJobject` function to convert the individual values provided in the `jvalue` array. The return type of the function is a `jobjectArray`. It will hold all the values as `jobject`s.

4.6.5 Variadic List Conversion

The variadic list (`va_list`) conversion function is used in the case of a JNI call that passes a variadic list to the JNI call. The signature of the helper is given in Code Fragment 4-8. It is similar to the `convertJvalueArrayToJobjectArray` function. A variadic list is created through varargs function calls such as `printf`.

```
1 jobjectArray convertVAListToJobjectArray(JNIEnv* env, jobject targetMethodObject,
va_list* listArgs)
```

Code Fragment 4-8: Signature of the convertVAListToJobjectArray function

The second argument is the `java.lang.reflect.Method` instance of the targeted method. The third argument is a pointer to the variadic list. It is assumed that the `va_start` and `va_end` macros are used outside this function, because applying them twice may cause errors.

Just like in the `convertJvalueArrayToJobjectArray` function, the `java.lang.reflect.Method` instance is again used to retrieve information about the target method, such as parameter types and the size of the parameter array.

The conversion and retrieval of the values are dependent on each other for this type of parameter conversion. For the retrieval of the value from the variadic list, the size of the actual parameter in memory needs to be known. The information that is gathered from the reflected method is used for this. An if-else construct is used to test each separate case, only performing the right case during runtime based on these tests. It uses primitive conversion helper functions to convert primitive values.

The return type of the function is a `jobjectArray` which holds the converted values. Only values that need conversion are converted. Others are returned as provided.

4.7 Limitations

One of the limitations of the parameterization of the JNI function calls are the nonvirtual function calls. This category of function calls does not take into account the inheritance hierarchy of objects in the JVM. A method that is overwritten by a subclass on a certain object can still be called through the nonvirtual JNI functions even though this is not possible in Java. As such, calls can be made from C to Java that cannot be replicated inside a JVM.

5 Maintaining the Causal Thread of Execution from Java to C

This chapter describes the modules of our implementation performing the steps (see also Figure 3-1):

- JNI Call Finder (step 3)
- Java JNI Replacement Stubs (step 4)
- Java Prolog Fact Generator (step 5) (instructs Compose* to generate interception code)
- C JNI Headers Generator (step 6) (generates the Callee-Side proxy)

The implementation is written in two Compose* modules, *StubInserter* and *CStubGenerator*. The *StubInserter* executes the steps 3, 4 and 5. The *CStubGenerator* executes step 6. The *StubInserter* is immediately executed after the Compose* has loaded the specifications. The *CStubGenerator* is executed at the end of the compilation process as it needs the Java binary code to generate the C JNI headers. The *StubInserter* takes care of the Java side of the JNI call (*Caller-Side Proxy*) and the *CStubGenerator* of the C side of the JNI call (*Callee-Side Proxy*).

The third part that completes the Java to C method call interception and rerouting is a *Java JNI Interceptor*, which receives the original JNI calls and which then invokes the *C Replacement Skeleton* by calling the *Java Replacement Stubs*.

5.1 Overview

An overview of the new control flow that is created by the modules in Figure 3-1 is given in Figure 5-1.

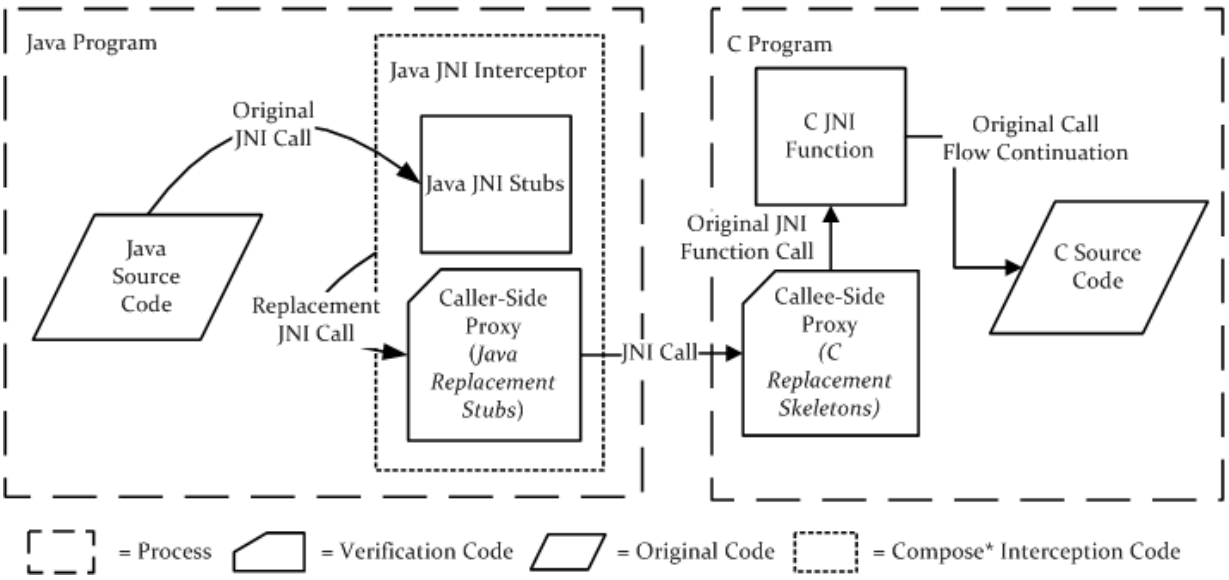


Figure 5-1: New control flow from Java to C

As shown, the *Original JNI Call* calls the original *Java JNI Stubs*. It is intercepted by the *Java JNI Interceptor*, which invokes the *Caller-Side Proxy* (*Java Replacement Stubs*). It inserts the causal thread id. The call flow will then continue to the *Callee-Side Proxy* (*C Replacement Skeletons*), which takes out the causal thread id. The replacement skeletons will invoke the target function of the *Original JNI Call* from where the original call flow continues.

5.2 Java JNI Call Finder

The *Java JNI Call Finder* is part of the *StubInserter* module of Compose*. It finds the JNI methods in the Java source code.

JNI methods can be found because of the usage of the *native* qualifier. An Abstract Syntax Tree (AST) is created from the source code files. For this we use ANTLR. The grammar can be reused from the DUMMER module that is already part of Compose*. If Compose* is updated with a new version of the Java syntax, it will automatically use the new version of the Java lexer and parser.

We traverse the AST and each method is checked for the `native` qualifier. For each method the return type, method name, annotations and parameter types are retrieved. The packagename of the class which contains the native method is retrieved from the AST, as it is used to find the file name for the code that will be generated in later steps.

The information about JNI methods is stored in a map together with the file in which they are defined. In later steps, this information is used for further processing.

5.3 Java JNI Replacement Stubs Generator

The *Java JNI Replacement Stubs Generator* is also part of the *StubInserter* module that is added to Compose*.

For each native method, found by the *Java JNI Call Finder*, a new method stub is generated. It is annotated with the `@ReplacementMethod` annotation. This annotation is added to prevent generation of multiple replacement methods for one native method when a file is compiled multiple times.

The import of the `ReplacementMethod` annotation is also added to the file. If the analysis process has not found any previous `ReplacementMethod` annotation on any method in the class it is automatically assumed that no `ReplacementMethod` import exists also, which is then added.

An example of a native method definition is given in Code Fragment 5-1. Its replacement code is given in Code Fragment 5-2.

```
public native String storeFile(String fileName, String text);
```

Code Fragment 5-1: Original Java code

```
public native String storeFile(String fileName, String text);

@ReplacementMethod
public native String storeFile_hook(String causalThreadID, String fileName, String
text);
```

Code Fragment 5-2: Java code after generation of extra method

The original method stub is kept, after the generation of the new code. This will make sure that the original program will still function if a different compiler is used.

The replacement method differs from its original version in three ways. The `@ReplacementMethod` annotation is added and the name of the method has "_hook" appended. It is appended to differentiate the name of the method from its original which makes generation of the C code easier, because the C JNI function name is easier to deduce.

Another difference is the first argument of the calls. In the replacement code the causal thread id is added as the first argument. This is necessary for the maintenance of the causal thread between the two parts of the program as described before.

5.4 Java Prolog Fact Generator

The *Java Prolog Fact Generator* is the last step that is part of the *StubInserter* Compose* module. This and the next section uses terminology of Compose* and its specification language. More information can be found at [19]. For every syntactic keyword like concern, inputfilter, filtermodule etc. there is a class equivalent, which is created when reading the specification.

In order to instruct Compose* to intercept calls to the original stub methods and reroute them to the newly defined stub methods, we define new concerns in the repository of Compose*.

The COPPER module reads the configuration files and creates concern objects in the repository from the concern definition files that are supplied in `BuildConfiguration.xml`. We can replicate this behavior by generating these concerns not from a configuration file, but based on information that is gathered in the analysis of the base program and then generating the concerns programmatically. This is done when finding JNI methods and creating replacement methods. Concerns need to be generated to reroute every call from an original JNI call to its replacement method.

For each class one concern is created. In each concern all the native methods of that class are intercepted. An `inputfilter` is defined per intercepted method. There are two main parts per concern, the `filtermodule` in which the `inputfilter`s and `selector`s are defined and the `superimposition` part where the `filtermodule`s are superimposed upon the class of which methods need to be intercepted.

An `external` is defined for each concern that refers to a singleton object. This object implements the functionality for invoking the replacement method based on the original target method of the invocation. It generates a new `filter.selector` property for the `inputfilter` that intercepts the original method call. The `filter.target` property refers to the earlier mentioned `external`. The type of `inputfilter` that is used is the Meta filter.

During the collection of the JNI methods, the class' fully qualified name was also retrieved. This classname is used in the `superimposition` part of the generated concern. In the `superimposition` a `selector` is generated that superimposes upon the class for which the concern was generated. The `filtermodule` that contains all the `inputfilter`s for each JNI method is then superimposed upon the `selector`. This will result in all methods matching the `inputfilter`s and their `selector`s to be intercepted and rerouted.

A `concern` is built according to the Composite design pattern. Each part has a common interface called `RepositoryEntity`. All these parts of the concern need to be registered in the repository. Just adding to the repository is not enough, since the repository can be queried in a multitude of ways. The repository can be queried by just retrieving all the `filtermodule` objects. The `concern` which contains the `filtermodule` does not need to be retrieved first.

5.5 Java JNI Interceptor

The *Java JNI Interceptor* is a Java class which is the same for every Java program that is verified. It is inserted in every program that has JNI calls from Java to C.

When a method is intercepted it is rerouted to a new location a special purpose object. The object is a singleton which is defined as an external in the prolog facts generation process. The `filter.selector` and `filter.target` properties are changed so that they are forwarded to this object. The signature of the method that the calls are rerouted to is given in Code Fragment 5-3.

```
public void replacementMethod(ReifiedMessage msg)
```

Code Fragment 5-3: Signature of new target method

The `ReifiedMessage` is the only argument of the method. The `ReifiedMessage` object provides information about the original call, such as the target object and the name of the method.

The `replacementMethod` method uses the name of the original target to find the replacement method stub. The new replacement stub is found through the usage of Java's reflection API. It adds the causal thread id as first argument for the replacement stub and takes all the original arguments and appends them to the causal thread id. Through reflection the new target method is invoked (which is the *Java Replacement Stub*). The result of the method call is stored on the `ReifiedMessage` object.

In order to prevent the original call flow to continue (as it would normally with the Meta inputfilter), the `target` and `selector` properties of the `ReifiedMessage` object are changed to a dead-end method.

We used the Meta Filter instead of the Dispatch filter, because the Dispatch Filter does not allow changes to the arguments and its values through its syntax and the underlying object model. Therefore we used a separate method to do it. In the future we can replace the Meta Filter and the separate class with the replacement method with a special Compose* filter and additional syntax to change the arguments and their values. As a result, we would also be able to add this new filter programmatically.

5.6 C JNI Header Generator

The CStubGenerator Compose* module consists solely of the *C JNI Header Generator* module.

The CStubGenerator module is is executed as the last module of Compose*/Java. It uses the fully compiled code to generate two C code files. The header files with the signatures of the native

methods, including the replacement methods, and a C source code file which contains implementations for the replacement methods.

To generate the header file, the module invokes the *javah* utility that is part of the JDK, on the class files that contain the native methods. The names of these class files were originally retrieved when analyzing the source code for native method calls (step 3, the *Java JNI Call Finder*). The same information is again used in this module.

To invoke the *javah* tool, our tool uses the `CommandLineExecutor` class located in the `Composestar.utils` package. The output directory is the same as that of the Compose*/Java compiler. The `javah` tool is invoked with the arguments `-jni` and `-o`. A default outputfile name is given with the `-o` option.

```
javah -jni -o nativesignatures.h <list of classnames>
```

Code Fragment 5-4: Invocation of the javah tool

In Code Fragment 5-4 the commandline instruction is shown. In `<list of classnames>` the classes from which the native signatures should be generated are given. This list can be generated from the information gathered by the *Java JNI Call Finder* (step 3). As a result of the execution of the *javah* tool, all the native signature headers for the whole program are gathered into one file.

The signatures file is included in another file which is also generated by the *CStubGenerator*. This second generated file provides implementations for the replacement methods. An example replacement method implementation is given in Code Fragment 5-5. It is the replacement method implementation for the replacement method in Code Fragment 5-2.

```
JNIEXPORT jstring JNICALL
Java_nl_utwente_rschutte_FileStore_storeFile_1hook(JNIEnv * env, jobject
object, jstring causalThreadId, jstring arg0, jstring arg1)
{

const char * threadId = (*env)->GetStringUTFChars(env, causalThreadId, 0);
(*env)->ReleaseStringUTFChars(env, causalThreadId, threadId);
return Java_nl_utwente_rschutte_FileStore_storeFile(env, object, arg0, arg1);

}
```

Code Fragment 5-5: Example of a stub implementation

As shown, there are only two lines in the implementation of the replacement method. The first line retrieves the causal thread id and stores it in a local variable. The second line invokes the original target function. The causal thread id is however not passed.

5.6.1 Java to C Name Matching

The name of the original method can be found since the derivation of the C function name from the Java method is static. The syntax of the C target function that can be derived from the Java native method stub.

```
Java_<fqn>_<method_name>
```

Code Fragment 5-6: Syntax of target function

Every JNI method starts its name with Java_. The `<FQN>` part stands for the fully qualified class name. The periods in the fully qualified classname are however replaced with underscores. Appended to it are an underscore and the original method name. An example of a JNI method name can be found in Code Fragment 5-3. Currently overloading of a method is not supported.

6 Demonstration

6.1 Example Programs

For testing we have two test programs. One for the Java to C direction of communication and one for C to Java. In the future we want to test the tool on real software, but due to the immaturity of the current tool, we only test it on our own test programs.

For Java to C we have a test program that asks the user for a title and a text. The text is then stored under the given title. The `FileStore` class is the interface to the C part of the program. The C part writes the file to the disk. The Java part of the program handles the user input. The orchestration of the compilation process has to be done manually. That means that the `ThreadStore`, `ReplacementMethod` and `EventReporter` have to be added manually and that the generated C code also has to be moved by the user to the right location.

The user interface of the editor is shown in Figure 6-1.

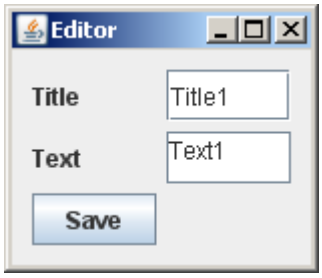


Figure 6-1: The editor's user interface

For C to Java we have written a test that does three calls to Java. (The Eclipse project *TestProjectCToJava*). The three calls match each type of JNI invocation (Var args, Jvalue array and var args list) and they vary with regards to being static and non-static. The contents of the test are initially in `main.c`. The generated code is put in `main_adapted.c`. There are two Java files: The JNIHook class, which is called by the generated code and the TestClass class, which holds a few methods for testing purposes.

6.2 Execution

6.2.1 Testing Java to C

The program is compiled by the Compose/Java compiler and the sole argument given to the compiler is the BuildConfiguration.xml in the documentLocal project. After compilation the Java classes are generated and two C files called `nativesignatures.c` and `nativesignatures.h` are generated. These have to be added manually to the C part of the program, by including these files during recompilation.

In each case, the original program and the adapted program, we create a file with the title "Title1" and the text "Text1". After entering the information we click the "Save" button.

The original program creates the output given in Code Fragment 6-1. This program is compiled without the modules that were added to the compiler as part of our tool. When we check the filesystem we can see that a new file is created at `<path to filestorage>\Title1`, which contains the text "Text1".

```
Stored document, response: H:\workspace\documentLocal\filestorage\Title1 | Text1
```

Code Fragment 6-1: Java to C original output

When we compile the program with Compose/Java including our new modules we get a different output. This output is shown in Code Fragment 6-2. The last line is the same as the line shown in the previous output. The first two lines are different.

The line denoted with `[REPLACED]` is output by the `EventReporter` class. It is output when an original JNI method is replaced by a generated method. In this case the method `storeFile` is replaced in the `FileStore` class.

The line denoted with `[CAUSAL THREAD]` is output in C code. It is output by a line which is part of the replacement method's C implementation. This is also where the causal thread id is extracted from the call flow.

```
[REPLACED] nl.utwente.rschutte.FileStore.storeFile(...)
[CAUSAL THREAD] storeFile - Thread ID: 22
Stored document, response: H:\workspace\documentLocal\filestorage\Title1 | Text1
```

Code Fragment 6-2: Java to C adapted output

6.2.2 Testing C to Java

First we compile the C to Java test, by invoking the Compose/CwC compiler and add as argument the location of the BuildConfiguration.xml of the project. The only file that is part of the C project (and as such, needs to be adapted) is the main.c file. The Java side does not need adaptation. When compiling the generated code, a few warnings will be given, but none that will stop the user from compiling the code.

After the Compose/CwC compiler is done. We have two files: main.c and main_adapted.c. The main.c file is the original code. When we run it, we get the output given in Code Fragment 6-3. The `[C]` denotes output in C and `[J]` denotes output in Java. There are three calls that are tested, the var args, jvalue array and va_list calls. The var args call (line 1 – 3) sums 10 and 15 and returns the result. The jvalue array call (line 4 – 6) passes two strings, "Hello " and "World!", and the native implementation concatenates them and returns the result. The va_list call (line 7 – 9) sums 100 and 250 and returns the result.

```
1 [C] TESTING VAR ARGS CALL
2 [J] 10 + 15 = 25
3 [C] Result = 25
4 [C] TESTING JVALUE CALL
5 [J] Result of concatenation: "Hello World!"
6 [C] Result: Hello World!
7 [C] TESTING VAR LIST CALL
8 [J] 100 + 250 = 350
9 [C] Result = 350
```

Code Fragment 6-3: Original C to Java test output

The adapted program has more lines of output. It is shown in Code Fragment 6-4. Again [C] is C output and [J] is Java output. In the adapted program there are lines that are not denoted with either of these. We can still find every line of output of the original program, but there are new lines added which are prepended with "--". These lines are generated by the JNI hook class, through which the calls are rerouted by the generated code. The most important lines are the lines starting with "-- CID:". They show the causal thread id, which was originally generated in the C part of the program. As shown, the original functionality is retained by the original program and the causal thread id has been passed to a different programming languages. In this case Java.

The var args call output is now shown on lines 1 – 14. The original output is there, but additional output has been generated. On line 3 – 8, the interception code outputs information. On line 7 the causal thread id is shown in Java, while it originated in C, thus showing that the causal thread id has been passed. The same happens for the jvalue array (line 15 – 23) and va_list (line 24 – 37) calls.

```
 1 [C] TESTING VAR ARGS CALL
 2
 3 -- JNIHook.staticIntJniHook(); --
 4 -- Trying to invoke sum
 5
 6 -- JNIHook.staticJniHook(); --
 7 -- CID: 2944
 8 -- Trying to invoke sum
 9 [J] 10 + 15 = 25
10 -- EXIT -- JNIHook.staticJniHook(); --
11
12 -- EXIT -- JNIHook.staticIntJniHook(); --
13
14 [C] Result = 25
15 [C] TESTING JVALUE CALL
16
17 -- ENTRY -- JNIHook.objectJniHook(); --
18 -- CID: 2944
19 -- Trying to invoke rik.TestClass.concatenate(...)
20 [J] Result of concatenation: "Hello World!"
21 -- EXIT -- JNIHook.objectJniHook(); --
22
23 [C] Result: Hello World!
24 [C] TESTING VAR LIST CALL
25
26 -- JNIHook.staticIntJniHook(); --
27 -- Trying to invoke sum
28
29 -- JNIHook.staticJniHook(); --
30 -- CID: 2944
31 -- Trying to invoke sum
32 [J] 100 + 250 = 350
33 -- EXIT -- JNIHook.staticJniHook(); --
34
35 -- EXIT -- JNIHook.staticIntJniHook(); --
36
37 [C] Result = 350
```

Code Fragment 6-4: Adapted C to Java test output

7 Conclusion & Future Work

The work can be divided into two solutions. The first solution is an attempt to deduce the distribution of a program through the generation of a call graph of the program and use it to generate optimized monitors. In the second solution we maintain the causal thread of programs, which are distributed and written in multiple languages, to be able to verify them. The first solution served as a motivation for the second part.

7.1 Call Graph Extraction

For the call graph extraction we have covered the basics. We have described how distribution through RMI usage can be detected. We have also determined how the combined usage of C and Java through JNI can be detected in the codebase.

For call graph extraction we have reviewed several tools and used two of them (Soot for Java, Cflow for C) to extract call graphs. We have tried to merge these call graphs, but we have shown that this is not possible. Fixes for the merging of the call graphs have been explained, but we found them to create a call graph that is too limited. We have written a prototype tool that extracts the call graphs of the Java and C languages and stores them in a uniform data structure.

The call graph approach has shown flaws in the design of the E-Chaser version for distributed systems, with regards to multilanguage support. The research showed that it can only work with an interlanguage layer that is not dependent on runtime context and values. An example of the problem was the JNI layer which we chose as the intermediate layer of our two target languages, Java and C.

7.2 Maintaining the Causal Thread ID

As a solution to the problems regarding multilanguage support in the second E-Chaser version's design, we designed an alternative approach that solves these problems. The design is based on reusing large parts of the Compose* infrastructure. The new tool is implemented as modules that

are added to Compose*. The generation of aspect oriented code is also taken from Compose* as well as the repository for storing information.

The new solution is based on the parameterization of the interlanguage calls, which can be generated without runtime information. We have designed a language independent verification infrastructure, which does not depend upon Java anymore when Java is not used for a program. For example a pure C program can be checked purely in C.

The solution consists of two parts. A C to Java part and a part for Java to C. From C to Java there are three different types of invocation which needed to be supported. Each of them requires a different code analysis and generation. The original JNI code is replaced by the parameterization code which calls a specifically created Java class (the JNI hook), which continues the original call flow.

For Java to C there are two modules that implement the functionality. In the first module the code base is analyzed and replacement stubs are generated. An internal concern file is generated and added to the repository, which reroutes the original method calls to the newly generated stubs. In the other module, which is the last module of the Compose* compiler, the stubs are implemented, by generating C headers and filling them with C code. The name of the original target can be found based on the original Java stub. An annotation called `ReplacementMethod` is added to the generated stubs to prevent stubs from being generated once they have been generated in a previous execution of the compiler.

For both Java and C a runtime library is developed. For C it is stored in one file called `helpers.c` which provides helper functions for the generated verification code. For Java, the JNI hook, the `ReplacementMethod` annotation, the thread storage code is part of the runtime library.

These parts together form a prototype of a runtime verification tool that implements the maintenance of the causal thread id.

7.3 Distribution Transparency

One of the requirements of the changes that we applied to E-Chaser was that the distribution transparency of the E-Chaser version for distributed program would be retained. We have achieved this by creating a distribution agnostic architecture in which the location of the events do not influence the functioning of the verification system.

7.4 Evaluation

In chapter 2 we already describe how MOP, TraceMatches, RMOR and Java-MaC compare with E-Chaser. As our work is an extension to E-Chaser, which was specifically designed to overcome

some of the problems in these tools, we will not evaluate our extension with these tools, but instead compare our extension with the previous versions of E-Chaser.

The new verification system maintains distribution support as delivered by the E-Chaser version for distributed systems. It does not rely on the Java programming language by design anymore. The communication from the local monitors to the verification infrastructure does not necessarily need to be written in Java as RMI is replaced with a language independent protocol.

The verification approach incurs runtime overhead costs. Some improvements can be made, by introducing structural information to the verification system. For example, through defining process structures.

In short, we have combined the first E-Chaser version which supports multilanguage programs, but not distributed and the second E-Chaser version which support distributed programs, but not multilanguage programs to a verification system which combines multilanguage and distribution support.

7.5 Future Work

7.5.1 Optimization of Monitoring

Runtime overhead in runtime verification systems can be significant. To decrease the amount of monitoring overhead, we can optimize the generation of monitors by letting the user provide the structure (a *process structure*) of the program. Such a process structure definition needs to define the process structure by grouping code. We can do this by grouping the code on a file path basis. In Java, file paths overlap with classes and their packages. We can select a single class by defining its path, or we can select a group of files through defining a path to a folder. These selected files are then considered one process structure.

An example program architecture is given in Figure 7-1. It is written in Java and C which communicate in a non-distributed manner.

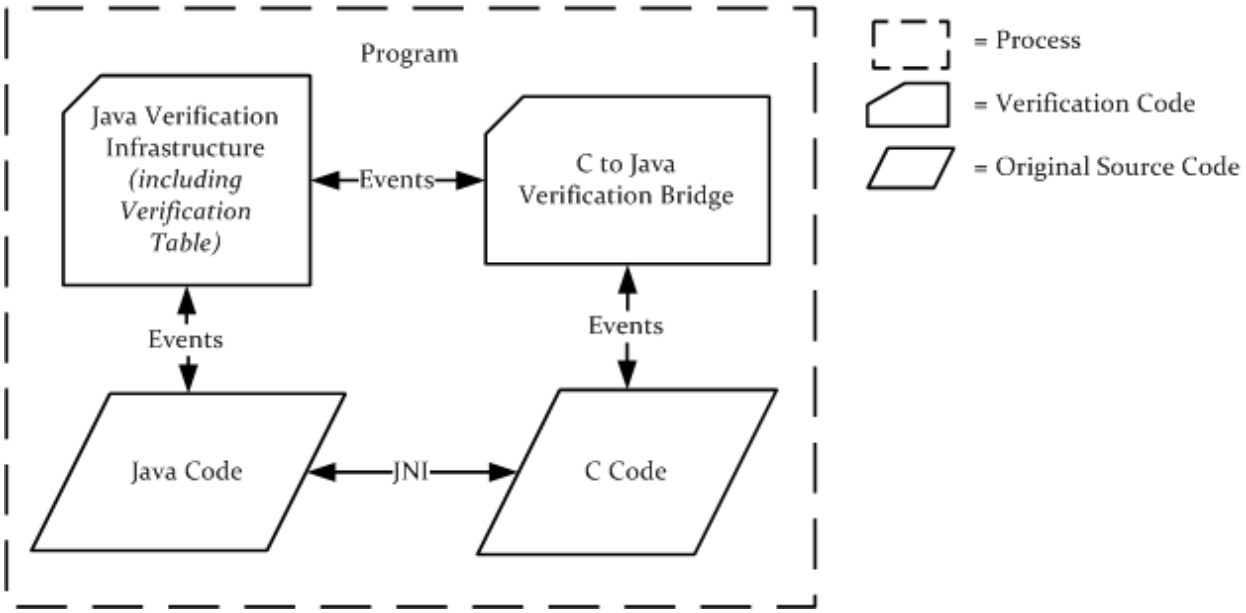


Figure 7-1: Optimized verification infrastructure for a non-distributed Java and C program

In this architecture it would be better to have only a single actual verification infrastructure to decrease remote communication. The *Java Verification Infrastructure* contains the verification table, which is retrieved from a separate process in a non-optimized infrastructure. The *C to Java Verification Bridge* is a piece of code, written in Java and C, which bridges the gap between Java and C through JNI. The same verification setup can be used for programs written in just Java, but then no verification bridge and no C source code would be part of the program.

7.5.2 Other Future Work

The verification system needs to be implemented further. Currently we have two separate tools. We have already proposed an Orchestrator to increase the ease of use for the end-user. Some modules (Translator, Causal Thread Manager) need to be integrated with the C to Java and Java to C tools to form a new E-Chaser version.

Currently only the Java and C programming languages are supported. Compose* also supports the C# programming language. This would be one of the easiest languages to include. However, also other languages could be included, such as PHP or Python. The support for other remote infrastructures can also be extended (e.g. SOAP).

We have only demonstrated our tools on small example programs. To be more confident about be the robustness of our tools, we need to test it on real applications, that use Java and C with JNI as interlanguage layer.

8 References

[1] Somayeh Malakuti, Christoph Bockisch and Mehmet Aksit, "Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software", ISSRE2009, Mysuru, India, 2009.

[2] Somayeh Malakuti, "E-Chaser: a Runtime Verification System for Distributed Software" [Presentation], April 2010, University of Twente.

[3] E-Chaser website. Available at: http://trese.cs.utwente.nl/e-chaser/

[4] F. Chen, and G. Rosu, "MOP: an efficient and generic runtime verification framework," OOPSLA, Montreal, Quebec, Canada, 2007.

[5] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ" OOPSLA, Oregon, USA, 2005.

[6] K. Havelund, "Runtime verification of C programs," TestCom/FATES., LNCS, vol. 5047, 2008.

[7] M. Kim, M.Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-MaC: a run-time assurance approach for Java programs," Formal methods in system design, vol. 24. Springer-Verlag, 2004.

[8] Dev Topics Website, 2008, "20 Famous Software Disasters". Available at: http://www.devtopics.com/20-famous-software-disasters/

[9] PC World Website, 2010, "11 Infamous Software Bugs". Available at: http://www.pcworld.com/article/205318/11_infamous_software_bugs.html

[10] aTunes website. Available at: http://www.atunes.org

[11] MPlayer website. Available at: http://www.mplayerhq.hu

[12] Java API, java.rmi.Remote class. Available at: http://java.sun.com/javase/6/docs/api/java/rmi/Remote.html

[13] Java JNI Documentation, "JNI Functions." Available at: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html#wp9502

[14] Java JNI Documentation, "JNI Types and Data Structures". Available at: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/types.html#wp9502

[15] Soot website. Available at: http://www.sable.mcgill.ca/soot/

[16] GNU Cflow website. Available at: http://www.gnu.org/software/cflow/

[17] Doxygen website. Available at: http://www.stack.nl/~dimitri/doxygen/

[18] GNU gprof website. Available at: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[19] Compose* website. Available at: http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/ComposeStarUsers

[20] Cscope website. Available at: http://cscope.sourceforge.net/

[21] Javah Tool, "javah - C Header and Stub File Generator". Available at: http://download.oracle.com/javase/6/docs/technotes/tools/windows/javah.html

[22] Compose* Documentation, "Repository Layout". Available at: http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/RepositoryLayout

[23] F. Chen and G. Rosu, "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation", Electronic Notes in Theoretical Computer Science 89 No. 2, 2003.

[24] K. Havelund and G. Rosu, "Java PathExplorer - A Runtime Verification Tool".

[25] Oleg Sokolsky, "Run-time Verification (with the MaC framework)" [Presentation], University of Pennsylvania, 2006.

[26] Ylies Falcone, "Introduction to Runtime Verification" [Presentation], INRIA, 2010.

[27] AspectJ tool website. Available at: http://www.eclipse.org/aspectj/

[28] E. Bodden, L. Hendren, P. Lam, O. Lothak and N.A. Naeem, "Collaborative runtime verification with tracematches", McGill University, Montréal, Québec, Canada and University of Waterloo, Waterloo, Ontario, Canada.

[29] K. Havelund and G. Rosu, "An overview of the runtime verification tool Java PathExplorer," in Formal methods in system design, vol. 24, 2004, pp. 189-215.

[30] E. Bodden and K. Havelund, "Racer: Effective Race Detection Using AspectJ", ISSTA2008, Seattle, Washington, USA, 2008.

[31] A. Milanova, A. Rountev and B.G. Ryder, "Precise Call Graphs for C Programs with Function Pointers", Kluwer Academic Publishers, 2003.

[32] D.C. Atkinson, "Call Graph Extraction in the Presence of Function Pointers", Proceedings of the 2002 InternationalConference on Software Engineering Research and Practice (SERP'02), Las Vegas, Nevada, June, 2002.

[33] Compose* Website. Composition Filters Documentation and MSc Theses. Available at: http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/ComposeStarDocumentation

[34] I. Nagy. "On the Design of Aspect-Oriented Composition Models for Software Evolution", PhD thesis, University of Twente, The Netherlands, June, 2006.

[35] J. Clark, "Calling Win32 DLLs in C# with P/Invoke", MSDN Magazine, July, 2003. Available at: http://msdn.microsoft.com/en-us/magazine/cc164123.aspx

[36] Microsoft MSDN, "Language Features for Targeting the CLR". Available at: http://msdn.microsoft.com/en-us/library/xey702bw.aspx

[37] Wikipedia. Static program analysis. Available at: http://en.wikipedia.org/wiki/Static_code_analysis

[38] Wikipedia. Model checking. Available at: http://en.wikipedia.org/wiki/Model_checking

[39] Wikipedia. Software testing. Available at: http://en.wikipedia.org/wiki/Software_testing

[40] Wikipedia. Call graph. Available at: http://en.wikipedia.org/wiki/Call_graph

[41] Java JNI Documentation. Available at: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html

[42] Wikipedia. Program Slicing. Available at: http://en.wikipedia.org/wiki/Program_slicing

[43] Wikipedia. Abstract syntax tree. Available at: http://en.wikipedia.org/wiki/Abstract_syntax_tree

[44] Java RMI Documentation. Available at: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Appendix A: Reviews of other Runtime Verification Systems

In this appendix the reviews of the other runtime verification systems can be found. They are also based on the findings of [1].

A.1 MOP

MOP [4] stands for Monitoring-Oriented Programming. It is a framework for the analysis of software in which the developer can define properties. MOP generates code to check whether these properties are satisfied or violated. JavaMOP is an implementation of MOP for Java. The concept is language independent, but an implementation is aimed at a single programming language.

Traces are specified as formulas of events. A logic can be specified, such as ERE (Extended Regular Expressions) and ptLTL (Past Time Linear Temporal Logic). Each trace can be given a validation and a violation handler.

JavaMOP provides handlers through executing code when a property is satisfied or violated. For convenience, it also provides the option of adding a raw specification (in the form of code) in cases where logical formalisms do not work well or for users that are not familiar with logical formalisms.

This framework can check systems written in a multitude of languages, if an implementation is written for those languages. Systems that consist out of parts written in multiple languages are not checkable by this framework. For distributed systems, no provisions are made.

A.2 TraceMatches

TraceMatches [5] is a system which is built upon AspectJ. It extends the functionality of AspectJ by adding advices which run only after a series of joinpoints are being encountered.

An event is written as a joinpoint and a trace is written as a series of events. The events thus match the joinpoint options (before, after, around advices) that AspectJ has to offer. A trace is defined as regular expression of events. A method body can be given that must be executed once a match of the trace has occurred.

TraceMatches is heavily intertwined with AspectJ and as result adapting it to other programming languages than Java may be difficult. Also, it does not take any provisions regarding programs that are potentially written in multiple programming languages. For distribution the same holds.

A.3 RMOR

RMOR (Requirement Monitoring and Recovery) [6] is a runtime verification framework for monitoring C programs against state machines. State machines are defined in a specification language. RMOR uses an aspect-oriented language to intercept the normal program flow. RMOR has the same setup as most verification frameworks.

In the specification, each state is written down. Exiting branches from each state are defined as events. This is in contrast to other systems where states are implicit steps between events. Events that should raise an error in a given state are also noted as outgoing branches of states. Events that do not cause a violation or a state change are not mentioned in a state; only relevant events are specified for a state. When the specification is violated a message is printed to the standard output and error handling functionality is executed if specified.

A centralized error handling function is provided in which the arguments provide information about the monitor, the state of the monitor (or state machine) and the kind of error that was produced.

RMOR is a checking system solely aimed at checking programs written in C. It also does not take any provisions for distributed programs.

A.4 Java-MaC

Java-MaC [7] is a runtime assurance system for Java programs based on the Monitoring and Checking (MaC) architecture. The MaC architecture is a general framework which is not limited to any specific programming language, however its implementations are.

Traces in MaC are written in terms of events and conditions. Events occur instantaneously and conditions hold during a certain period of time. Events are associated with a timestamp. A condition is started and ended when a specified event takes place. The correctness of a system is described through specifying safety properties and alarms. Safety properties are conditions

(mentioned earlier) that always need to be true. Alarms are events that should never be raised. Java-MaC has a similar setup as many other runtime verification systems.

MaC is a language independent architecture, but an implementation is not aimed at multiple language software. The architecture does not take into account distribution of programs.

Appendix B: C JNI Code Replacement

```
 1 // Load the target class
 2 jclass targetClass = (*env)->FindClass(env, "rik/TestClass");
 3 // Find the constructor
 4 jmethodID constructorMethod = (*env)->GetMethodID(env, targetClass, "<init>",
"()V");
 5 // Create a new object based on this constructor
 6 jobject targetObject = (*env)->NewObject(env, targetClass, constructorMethod);
 7
 8 // Get the non-static Java method which has to be called
 9 jmethodID targetMethod = (*env)->GetMethodID(env, targetClass,
"retrieveConcatenation", "(ILjava/lang/String;)Ljava/lang/String;");
10 // Create arguments for the Java method call
11 jint firstArg = (jint)100;
12 jstring secondArg = (*env)->NewStringUTF(env, "TestArgument");
13
14 // Call the Java method
15 jobject result = (*env)->CallObjectMethod(env, targetObject, targetMethod,
firstArg, secondArg);
16
17 // Throw Java exception if it has occurred
18 if((*env)->ExceptionOccurred(env)) {
19     (*env)->ExceptionDescribe(env);
20 }
```

Code Fragment B-1: Example varargs JNI invocation

```
  1 jstring threadId = (*env)->NewStringUTF(env, "AGeneratedThreadID");
  2
  3 // Loading hook
  4 jclass hookClass = (*env)->FindClass(env, "rik/JNIHook");
  5 jmethodID hookMethod = (*env)->GetStaticMethodID(env, hookClass, "objectJniHook",
"(Ljava/lang/String;Ljava/lang/Object;Ljava/lang/reflect/Method;[Ljava/lang/Object;)Lj
ava/lang/Object;");
  6
  7 // Get the reflected class of the target object
  8 jclass reflectedTargetClass = (*env)->GetObjectClass(env, targetObject);
  9 // Get the reflected target method based
 10 jobject reflectedTargetMethod = (*env)->ToReflectedMethod(env,
reflectedTargetClass, targetMethod, JNI_FALSE);
```

```
 11
 12 // Create an array to pass to the JNI function
 13 jobjectArray args = createObjectArgumentArray(env, 2);
 14
 15 // Retrieving the java.lang.reflect.Method.getParameterTypes method
 16 jclass methodClass = (*env)->FindClass(env, "java/lang/reflect/Method");
 17 jmethodID parameterMethod = (*env)->GetMethodID(env, methodClass,
"getParameterTypes", "()[Ljava/lang/Class;");
 18
 19 // Load the java.lang.Class.getName() method
 20 jclass classClass = (*env)->FindClass(env, "java/lang/Class");
 21 jmethodID getNameMethod = (*env)->GetMethodID(env, classClass, "getName",
"()Ljava/lang/String;");
 22 // Retrieving the actual parameter types of the given function
 23 jobjectArray paramTypes = (*env)->CallObjectMethod(env, reflectedTargetMethod,
parameterMethod);
 24
 25 // Storage variable
 26 jobject resultantArgument;
 27
 28 // For each parameter, get the type
 29 jobject paramType_0 = (*env)->GetObjectArrayElement(env, paramTypes, 0);
 30 // Get the name of the class
 31 jstring paramClassNameJString_0 = (*env)->CallObjectMethod(env, paramType_0,
getNameMethod);
 32 // Dont forget to release this string !!!
 33 const char * paramClassName_0 = (*env)->GetStringUTFChars(env,
paramClassNameJString_0, 0);
 34
 35 jvalue temp_0;
 36
 37 if(strncmp(paramClassName_0, "[", 1) != 0) {
 38    // It is a non-array type
 39    if(strcmp(paramClassName_0, "boolean") == 0) {
 40          temp_0.z = firstArg;
 41    } else if(strcmp(paramClassName_0, "byte") == 0) {
 42          temp_0.b = firstArg;
 43    } else if(strcmp(paramClassName_0, "char") == 0) {
 44          temp_0.c = firstArg;
 45    } else if(strcmp(paramClassName_0, "short") == 0) {
 46          temp_0.s = firstArg;
 47    } else if(strcmp(paramClassName_0, "int") == 0) {
 48          temp_0.i = firstArg;
 49    } else if(strcmp(paramClassName_0, "long") == 0) {
 50          temp_0.j = firstArg;
 51    } else if(strcmp(paramClassName_0, "float") == 0) {
 52          float temp_float;
 53          memcpy(&temp_float, &(firstArg), sizeof(jfloat));
 54          temp_0.f = temp_float;
 55    } else if(strcmp(paramClassName_0, "double") == 0) {
 56          double temp_double;
 57          memcpy(&temp_double, &(firstArg), sizeof(jdouble));
 58          temp_0.d = (double) temp_double;
 59    } else {
 60          // Apparantly its an object
 61          temp_0.l = firstArg;
 62    }
 63 } else {
 64    // It is an array type which is already an object
 65    Temp_0.l = firstArg;
 66 }
 67
```

```
 68 resultantArgument = convertJvalueToJobject(env, paramType_0, temp_0);
 69 (*env)->SetObjectArrayElement(env, args, 0, resultantArgument);
 70
 71 // For each parameter, get the type
 72 jobject paramType_1 = (*env)->GetObjectArrayElement(env, paramTypes, 1);
 73 // Get the name of the class
 74 jstring paramClassNameJString_1 = (*env)->CallObjectMethod(env, paramType_1,
getNameMethod);
 75 // Dont forget to release this string !!!
 76 const char * paramClassName_1 = (*env)->GetStringUTFChars(env,
paramClassNameJString_1, 0);
 77
 78 jvalue temp_1;
 79
 80 if(strncmp(paramClassName_1, "[", 1) != 0) {
 81    // It is a non-array type
 82    if(strcmp(paramClassName_1, "boolean") == 0) {
 83           temp_1.z = secondArg;
 84    } else if(strcmp(paramClassName_1, "byte") == 0) {
 85           temp_1.b = secondArg;
 86    } else if(strcmp(paramClassName_1, "char") == 0) {
 87           temp_1.c = secondArg;
 88    } else if(strcmp(paramClassName_1, "short") == 0) {
 89           temp_1.s = secondArg;
 90    } else if(strcmp(paramClassName_1, "int") == 0) {
 91           temp_1.i = secondArg;
 92    } else if(strcmp(paramClassName_1, "long") == 0) {
 93           temp_1.j = secondArg;
 94    } else if(strcmp(paramClassName_1, "float") == 0) {
 95           jfloat temp_float;
 96           memcpy(&temp_float, &(secondArg), sizeof(jfloat));
 97           temp_1.f = temp_float;
 98    } else if(strcmp(paramClassName_1, "double") == 0) {
 99           jdouble temp_double;
100           memcpy(&temp_double, &(secondArg), sizeof(jdouble));
101           temp_1.d = temp_double;
102    } else {
103           // Apparantly its an object
104           temp_1.l = secondArg;
105    }
106 } else {
107    // It is an array type which is already an object
108    temp_1.l = secondArg;
109 }
110
111 resultantArgument = convertJvalueToJobject(env, paramType_1, temp_1);
112 (*env)->SetObjectArrayElement(env, args, 1, resultantArgument);
113
114 // Call Java method through the JNI hook
115 jobject jniResult = (*env)->CallStaticObjectMethod(env, hookClass, hookMethod,
threadId, targetObject, reflectedTargetMethod, args);
116
117 // Throw Java exception if it has occurred
118 if((*env)->ExceptionOccurred(env)) {
119    (*env)->ExceptionDescribe(env);
120 }
```

Code Fragment B-2: Example varargs JNI invocation replacement code

```
 1 // Load target class for the method invocation and create an instance
 2 jclass targetClass = (*env)->FindClass(env, "rik/TestClass");
 3 jmethodID constructorMethod = (*env)->GetMethodID(env, targetClass, "<init>",
"()V");
 4 jobject targetObject = (*env)->NewObject(env, targetClass, constructorMethod);
 5 // Load the target method
 6 jmethodID targetMethod = (*env)->GetMethodID(env, targetClass,
"retrieveConcatenation", "(ILjava/lang/String;)Ljava/lang/String;");
 7
 8 // Create a arguments for the JNI function call
 9 jint firstArg = (jint)100;
10 jstring secondArg = (*env)->NewStringUTF(env, "TestArgument");
11
12 // Store them in two jvalue unions
13 jvalue args[2];
14 jvalue arg1, arg2;
15 arg1.i = firstArg;
16 arg2.l = secondArg;
17
18 // Store the jvalue unions in a jvalue array
19 args[0] = arg1;
20 args[1] = arg2;
21
22 // Call the Java method, note the append A to the function name
23 (*env)->CallObjectMethodA(env, targetObject, targetMethod, args);
24
25 // Throw Java exception if it has occurred
26 if((*env)->ExceptionOccurred(env)) {
27     (*env)->ExceptionDescribe(env);
28 }
```

Code Fragment B-3: Example jvalue array JNI invocation and its replacement code

```
 1 // Static causal thread id
 2 jstring threadID = (*env)->NewStringUTF(env, "CausalThreadID");
 3
 4 // Use JNI to get the java.lang.reflect.Method object for the method
 5 jclass targetObjectClass = (*env)->GetObjectClass(env, targetObject);
 6 jobject targetMethodObject = (*env)->ToReflectedMethod(env, targetObjectClass,
targetMethod, JNI_FALSE);
 7
 8 jobjectArray paramObjectValues = convertJvalueArrayToJobjectArray(env,
targetMethodObject, args);
 9
10 // Load Java JNI hook class and the appropriate method
11 jclass hookClass = (*env)->FindClass(env, "rik/JNIHook");
12 jmethodID hookMethod = (*env)->GetStaticMethodID(env, hookClass, "objectJniHook",
"(Ljava/lang/String;Ljava/lang/Object;Ljava/lang/reflect/Method;[Ljava/lang/Object;)Lj
ava/lang/Object;");
13
14 // Call the actual JNI method
15 jobject jniResult = (*env)->CallStaticObjectMethod(env, hookClass, hookMethod,
threadID, targetObject, targetMethodObject, paramObjectValues);
16
17 // Throw Java exception if it has occurred
18 if((*env)->ExceptionOccurred(env)) {
19     (*env)->ExceptionDescribe(env);
20 }
```

Code Fragment B-4: Example jvalue array JNI invocation replacement code

```
1 jint firstArg = (jint)100;
2 jstring secondArg = (*env)->NewStringUTF(env, "TestArgument");
3
4 testVAList(env, firstArg, secondArg);
```

Code Fragment B-5: Variadic function invocation example

```
 1 // A storage for the varargs list
 2 va_list listArgs;
 3
 4 // Load the target class and create an instance
 5 // of it through the constructor
 6 jclass targetClass = (*env)->FindClass(env, "rik/TestClass");
 7 jmethodID constructorMethod = (*env)->GetMethodID(env, targetClass, "<init>",
"()V");
 8 jobject targetObject = (*env)->NewObject(env, targetClass, constructorMethod);
 9
10 // Load the target method
11 jmethodID targetMethod = (*env)->GetMethodID(env, targetClass,
"retrieveConcatenation", "(ILjava/lang/String;)Ljava/lang/String;");
12
13 // Call the target method through a variadic function call
14 va_start(listArgs, env);
15 jobject vResult = (*env)->CallObjectMethodV(env, targetObject, targetMethod,
listArgs);
16 va_end(listArgs);
17
18 // Throw Java exception if it has occurred
19 if((*env)->ExceptionOccurred(env)) {
20     (*env)->ExceptionDescribe(env);
21 }
```

Code Fragment B-6: Example jvalue array JNI invocation

```
 1 // Creating a static causal thread id
 2 jstring threadID = (*env)->NewStringUTF(env, "CausalThreadID");
 3
 4 // Get the java.lang.reflect.Method object of the targeted Java method
 5 jclass targetObjectClass = (*env)->GetObjectClass(env, targetObject);
 6 jobject targetMethodObject = (*env)->ToReflectedMethod(env, targetObjectClass,
targetMethod, JNI_FALSE);
 7
 8 // Load the hook class and the appropriate hook method
 9 jclass hookClass = (*env)->FindClass(env, "rik/JNIHook");
10 jmethodID hookMethod = (*env)->GetStaticMethodID(env, hookClass, "objectJniHook",
"(Ljava/lang/String;Ljava/lang/Object;Ljava/lang/reflect/Method;[Ljava/lang/Object;)Lj
ava/lang/Object;");
11
12 // Set up the var args list and convert it to a jobjectArray
13 // This setting up wont be necessary when this code is actually replacing
14 // the original JNI call as it already does this
15 va_list list;
16 va_start(list, env);
17 jobjectArray argumentsArray = convertVAListToJobjectArray(env, targetMethodObject,
&list);
18 va_end(list);
19
20 // Call the actual JNI method through the JNI hook
21 jobject jniResult = (*env)->CallStaticObjectMethod(env, hookClass, hookMethod,
threadID, targetObject, targetMethodObject, argumentsArray);
22
23 // Throw Java exception if it has occurred
24 if((*env)->ExceptionOccurred(env)) {
25     (*env)->ExceptionDescribe(env);
26 }
```

Code Fragment B-7: Example replacement code for a jvalue array JNI invocation

```
 1 grammar Cflow;
 2
 3 options {
 4     language=Java;
 5     output=AST;
 6 }
 7
 8 // Every input line needs to be trimmed
 9
10 model      :      (line '\n'!)+;
11 line       :      INT tab* methodcall^ recursive? ':'? linereference?;
12 tab        :      '    '; // 4 spaces
13 methodcall :      functionname^ '()'! signature?;
14 signature  :      '<'! 'JNIEXPORT'? typeslist 'JNICALL'? functionname^ ('('!
arguments? ')'! | '()'!)! filereference '>'!;
15 functionname      :      STRING;
16 arguments  :      (argument ','!)* (argument);
17 argument   :      typeslist name^;
18 typeslist  :      (type | 'const' | 'unsigned')+;
19 type       :      (STRING (('*'*) | '&'));
20 name       :      STRING;
21 filereference     :      'at'! FILEPATH ':'! INT;
22 recursive  :      '(R)'! | ('(recursive: see '! INT ')'!);
23 linereference     :      '['! 'see'! INT ']'!;
24
25 INT :      DIGIT+
26    ;
27
28 STRING :   (LOWER|UPPER|DIGIT|'_')+
29   ;
30
31 FILEPATH: (STRING |'-' | '+' | '!' | '@' | '#' | '$' | '%' | '^' | '&' | '/' |
'..' | '.')+;
32
33 WS  :    ( ' '
34         | '\r'
35          ) {$channel=HIDDEN;}
36     ;
37
38 fragment DIGIT
39     :      ('0'..'9')
40     ;
41
42 fragment LOWER
43     :      ('a'..'z')
44     ;
45
46 fragment UPPER
47     :      ('A'..'Z')
48     ;
49
50 fragment LETTER
51     :      LOWER | UPPER
52     ;
```

Code Fragment B-8: ANTLR grammar for parsing Cflow output