5
Exception-Flow Analysis for AspectJ Programs

To discover the exceptions that can flow from an *aspect advice* can become a daunting, if not infeasible, task to be manually performed - especially if we take into account the *unchecked* exceptions available in AspectJ. The exceptions that may escape from an advice can come from different sources: (i) they can be explicitly thrown by throw statements; (ii) they can also be implicitly thrown by operations (e.g., the division by zero throws an instance of `ArithmeticException`, and the access of a null reference throws an instance of `NullPointerException`); (iii) exceptions can be thrown by the JDK environment when an abnormal situation occurs (e.g., `OutOfMemoryException`); (iv) there are also AspectJ environment exceptions that can be thrown by the additional code included by the AspectJ weaver during aspect composition (e.g., `NoAspectBoundException`); and finally (v) exceptions can also be implicitly thrown by a library method call.

Usually, to know which exceptions may be thrown from a piece of code, the developer must rely on documentation that specifies the exceptions (checked and unchecked) that a method or advice may throw. However, most of the time, such documentation is neither precise nor complete (Sacramento et al., 2006; Cabral and Marques, 2007). This chapter presents SAFE (*Static Analysis for the Flow of Exceptions*), an *exception-flow analysis* tool that computes the exception flow of AspectJ programs. In Sections 5.1 and 5.2 we present the supporting ideas and the heuristics adopted in SAFE, and in Section 5.3 we describe its design and implementation details.

5.1.
Supporting Ideas

The exception-flow analysis resembles the data-flow analysis for finding *def-use* pairs (Myers, 2004); but, instead of using the control flow graphs, it traverses the program call graph (see Section 5.1.1). In the *exception-flow*

analysis, the places in which an exception is thrown (e.g., by `throw` statements) and handled (by an enclosing `try-catch` block) are equivalent to the places where a variable is defined and used in *def-use* analysis algorithms. When an exception is signaled it propagates along the dynamic call chain until a proper handler is found. If no handler is found, it remains *uncaught* and reaches the program entry point. Therefore, to compute the flow of program's exceptions, the *exception-flow analysis* algorithm traverses the program call graph *backwards* until a proper handler is found for each exception signaled inside the program.

Some tools have been proposed so far to calculate the exception flows of OO programs. However these tools cannot be used in a straightforward fashion to analyze AspectJ programs, since they do not interpret the characteristics of AspectJ source code nor the effects on *bytecode* caused by the AspectJ weaving process (Hilsdale and Hugunin, 2004). The advantage of working on Java *bytecode* level instead of the source code level is that the exception analysis algorithm can analyze the exceptions that flow from reused pre-compiled *libraries* (which are responsible for a large number of the exceptions that flow within current applications).

For that reason, we have implemented the tool called SAFE that analyzes the *woven bytecode* of AspectJ programs to (i) discover the exceptions that can be signaled from aspect advice (i.e., *exception interfaces*) and (ii) find out how such exceptions propagate in the base code (i.e., *exception paths*). This *exception-flow analysis* tool was implemented on top of the Soot framework for *bytecode* analysis and transformation. Next sections describe the supporting ideas and the heuristics adopted on the implementation of SAFE.

5.1.1. Advice Weaving in AspectJ

In the former versions of AspectJ (previous to version 1.2) the weaving process happened in the program source code. The additional behavior of an advice was, therefore, directly injected into specific points of the affected classes' source code. Currently, the weaving process happens at the *bytecode* level and produces the woven code as pure Java *bytecode* as a result.

The main idea of the *bytecode-level* AspectJ weaver is to convert aspects and advices into standard Java classes and methods, respectively. Advice parameters are converted into parameters of these new methods (Hilsdale and Hugunin, 2004). In order to coordinate aspects and non-aspects, the system code is instrumented and calls to the *"advice methods"* are inserted at specific join points - they are also called *static shadows* (Hilsdale and Hugunin, 2004). Furthermore, if the join points cannot be completely determined at compile time, the call to *"advice methods"* are guarded by dynamic tests to make sure that the pieces of advice are executed only when specific conditions are satisfied. Such guards are used to implement `cflow` and `cflowbelow` *pointcut* designators presented in Chapter 2.

Therefore, we can identify the places where the pieces of advice add new behavior in the *bytecode* (resultant from the compilation/weaving process) by observing the places where the *"advice methods"* are called in the *bytecode*. Listing 3 illustrates the source code of an aspect and an advised method and Listing 5 presents the decompiled[19] *woven bytecode* of the advised method.

```
1. public class ExampleClass {
2.   public void methodA (){
3.      System.out.println("Just an example");
4.   }
5. }
```

```
6. package aspects;
7. aspect ExampleAspect {
8.    public pointcut aMethodCall() :
9.        execution(public *  methodA (..));
10.    before() : aMethodCall(){
11.       System.out.println("Affected method");
12     }
13.   }
```

Listing.3. Example of an aspect that contains a before advice.

```
1. public void methodA (){
2. ExampleAspect.aspectOf().
      ajc$before$aspects_ExampleAspect$1$dcd6c0af(…);
```

```
3. System.out.println("Just an example");
4. }
```

Listing.4. Decompiled code of the advised method after weaving process.

In Listing 4 we can observe that the before advice was converted to a call of a method defined in the `ExampleAspect`. The `aspectOf()` used in line 2 is a static method available in all AspectJ aspects, which returns the singleton instance of an aspect (it is used by any class to call the public methods from an aspect) (Kiczales et al., 2001a). Generally, the name of an *advice method* is formed in the following way: `ajc$`*`advicetype`*`$`*`package_aspectName`*`$`*`methodID`*. Where the `advicetype` can be one of the following: `after`, `afterReturning`, `afterThrowing`, `before`, or `around`. `methodID` is a code generated by AspectJ weaver to uniquely identify each *advice method*. `before` and `after` advice modify the *bytecode* in a similar way: static calls to *advice methods* defined on Aspects are included in specific places in the affected code. The `around` advice, however, works slightly differently because of the execution of the `proceed` method (Kiczales et al., 2001a) – that proceeds the execution of the affected join point. When a method m1 is affected by an `around` advice, the original method body is extracted (block of *bytecode*) and replaced by a call to a static method defined in the affected class, which comprises: (i) the *advice method* body and optionally (ii) a call to the original method body (`proceed`).

Table 10 briefly illustrates the effect of aspects on the woven *bytecode*. The first column contains a *pointcut* expression associated with an advice type and the second column presents the partial code of the woven *bytecode* affected by it. `<AspectID>` represents the name of the aspect on which the advices were defined, `<ClassID>` represents the name of the affected class, and `<Id>` is the *advice method* unique identifier. This identifier can be formed by a sequential number, or a combination between a sequential number, and the identifier by the affected code (e.g., class name and method signature).

[19] We used Dava decompiler, a module of Soot framework, to decompile the *woven bytecode*.

Advice Type + *Pointcut* Expression	Effect on Decompiled Woven *Bytecode*

before():
execution(
public * *(..))
	public void method(){
	<AspectID>.aspectOf().ajc$before$<Id>(…);
	//original method body
	...
	}

after () returning :
execution(
public * *(..))
	public void method(){
	//original method body
	...
	<AspectID>.aspectOf().ajc$afterReturning$<Id>(…);
	}

after () throwing
(Exception e):
execution(
public * *(..))
	public void method() throws Exception{
	try{
	//original method body
	...
	} catch(Exception t){
	<AspectID>.aspectOf().ajc$afterThrowing$<Id>(…);
	throw t;
	}
	}

after():
execution(
public * *(..))
	public void method() throws Throwable{
	try{
	//original method body
	...
	<AspectID>.aspectOf().ajc$after<Id>(…);
	} catch(Throwable t){
	<AspectID>.aspectOf().ajc$after<Id>(…);
	throw t;
	}

around():
execution(
public * *(..))
	public void method(){
	<ClassID>.method_aroundBody<Id>$advice(…);
	}
	private static final method_aroundBody<Id>$advice(…){
	<ClassID>.method_aroundBody<Id>();
	}
	//original method body.It represents a call to proceed()
	private static final method_aroundBody<Id>(…){
	...
	}

Table 10. Representation of aspect advice on Java *bytecode*.

As illustrated above when an advice is associated with a *method execution pointcut* the call to the *advice method* is included at a specific point of the advised

method's *bytecode*. On the other hand, when an advice intercepts a *method call* pointcut, the *advice method* is included on the corresponding *bytecode* of every advised method's caller. In this work we only account for *method call* and *method execution pointcut* descriptors.

5.1.2. Program Representation

The program representation used by the SAFE tool to statically calculate the exception paths of AspectJ programs is a variant of a program call graph. A program call graph (Ryder, 1979) is a directed graph whose vertices and edges represent method fragments and method call relations, respectively. Each edge connects a *caller method* to a *target* method.

In the call graph used in this approach each vertex may represent: (i) a *application method* – method defined in a class (base code); (ii) an *aspect method*[20] – method defined in an aspect; (iii) an *intertype method* – method added to a class by an aspect's intertype declaration; or (iv) an *advice method* – representation of an advice in the byte code, since as illustrated above advices are converted into standard Java methods by AspectJ weaver (Hilsdale and Hugunin, 2004). In the rest of this document, for the sake of simplicity, unless it is explicitly mentioned we use the term "advice method" to refer to both, advice and intertype methods.

The advice method appears on specific points of the program call graph according to the AspectJ weaver rules detailed in the previous section (see Table 10). Figure 15 (a) illustrates how `before` and `after` affect a program call graph, and Figure 15 (b) illustrates how and `around` advice affects a program call graph. This figure illustrates a method (i.e., `method1()`) that is intercepted by such advices, and as a consequence calls to *advice methods* are included at specific points in the program, affecting the program call graph.

[20] Although *aspect methods* are represented on the program call graph used by SAFE tool, the exception analysis algorithm does not calculate the exception interfaces of such methods since they usually implement part of the behaviors defined by advices.

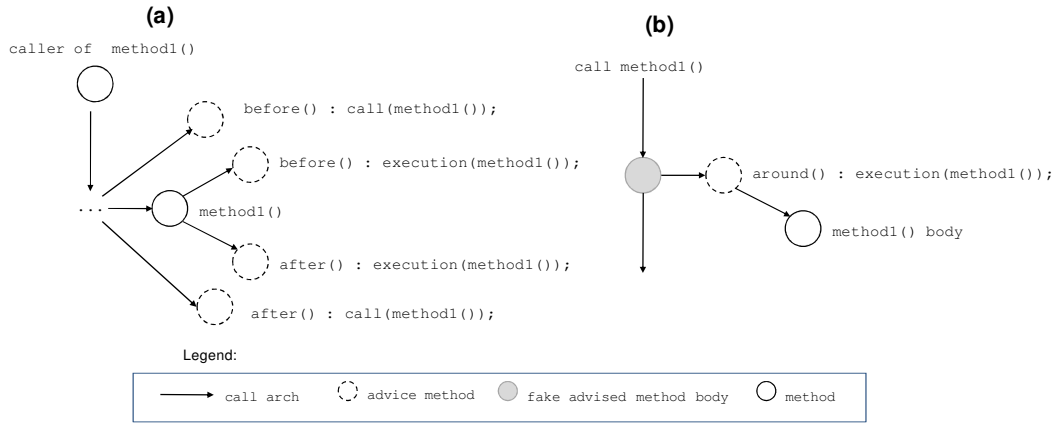

Figure 15. Advice methods on the program call graph.

Each vertex has an inner structure comprising *exceptional control flow* information of interest to our analysis. For the exception flow analysis, the key control-flow statements in a method are: `try-catch` blocks, `throw` statements and method calls. All other statements do not affect the exception-flow analysis. Moreover, the order of these statements within a method also does not influence the analysis. What matters is whether or not a throw statement or method call (which may signal an exception) is contained in a try block. Therefore, the inner structure defined by a method comprises: the list of nodes representing the statements that compose the method and arcs connecting every statement in a protected area (`try`-block) to the handlers associated with it (`catch`-clauses). This inner structure resembles a simplified representation of a method control flow graph (CFG) (Myers, 2004). Figure 16 depicts the program representation described here (see Section 5.4.2 for a concrete example of the inner structure of call graph vertices).

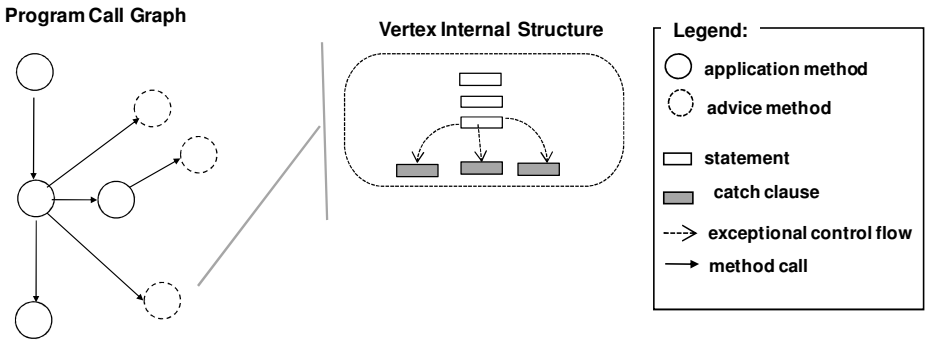

Figure 16. Program Representation.

5.1.2.1. Dealing with Dynamic Dispatch

In languages such as Java and AspectJ that allow dynamic dispatch (Gosling et al., 1996; Rountev et al., 2004), calling method *m* defined in the class *A* may represent, in runtime, any one of the redefining methods of *m* in the inheritance tree of *A*. To deal with dynamic dispatch in OO systems one of the most used algorithms for call graph construction is the *Class Hierarchy Analysis* (CHA) algorithm (Grove and Chambers, 2001). According to it, when a method *m1* defined in class *c* is overridden in *n* subclasses of c, every time a method *m2* calls the method c.*m1*, *n*+1 edges are included on the call graph: one from *m2* to *c.m1,* and *n* edges from *m2* to every subclass of c that redefines *m1*. The CHA algorithm is used by most inter-procedural analyses proposed so far (Robillard and Murphy, 2003; Vincenzi et al., 2003; Ishio et al., 2004; Sinha et al., 2004; Fu and Ryder, 2005).

5.2. Heuristics used by the Tool

Some questions arise when analyzing the exception flow of AspectJ programs: (i) *How should we assign the responsibility of an exception signaled in the system; was it signaled by an aspect or a class?* (ii) *How should the static analysis tool deal with the exception softening construct, which converts a checked exception into a* `SoftException`*?* (iii) *How should the static analysis tool deal with iterative code?* To answer such questions we needed to elaborate a set of heuristics. The next sections present the heuristics aimed at taming the complexity inherent to the exception flow in AspectJ programs.

5.2.1. Blame Assignment in Exception Handling Scenarios

When an exception is thrown by an element of a software system and is not caught inside it, such exception will reach the program entry point and lead to a software crash. When such failures occur, it is important to pinpoint the element responsible for signaling the *uncaught* exception (the guilty party) in order to understand and solve the exception handling fault.

One particular issue raised by the exception handling scenarios of AO systems is the ability to identify which element is responsible for signaling and handling exceptions: an aspect or a class. Such ability can, for instance, blame the aspect integration for breaking the existing *exception handling policy* implemented in the base code. Consequently, it will help the aspect designer to define the exception handling policy for the exceptions explicitly or implicitly thrown by aspects.

Figure 17 depicts exception handling scenarios in which aspects were directly and indirectly involved. These scenarios are used to explain the heuristics for *blame assignment* adopted in our approach.

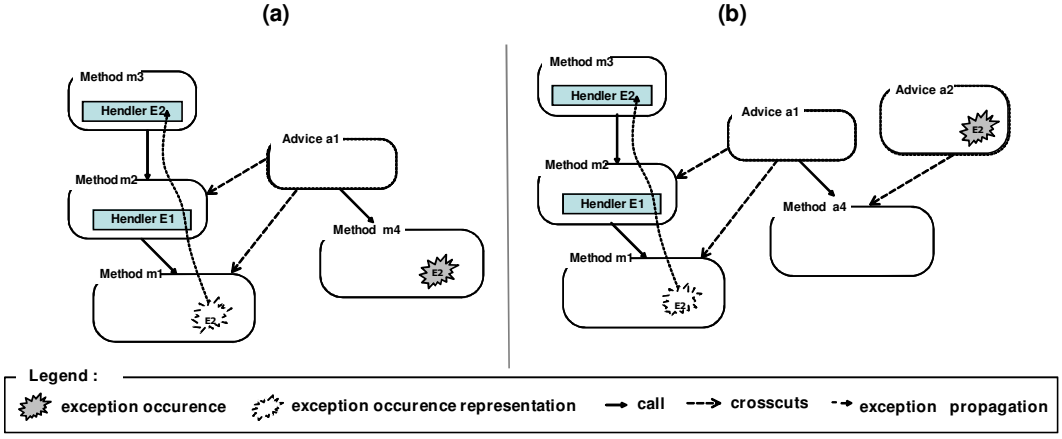

Figure 17. Scenarios where advice act as exception signalers.

Considering Figure 17 in scenario (a) the aspect advice a1 explicitly calls method m4 in order to achieve some part of its functionality. Since the code inside m4 could also be defined inside the advice, we consider m4 as part of the core behavior of a1, and call it an *auxiliary method* (i.e., a method defined inside the aspect or inside a class that is only used by aspects). In this scenario we can say that advice a1 bears the responsibility of the effects that exception E2 may cause in the base code. Therefore, in our approach, every time an advice calls an *auxiliary method* that throws an exception, the advice will take the responsibility for signaling the exception.

There are also scenarios in which m4 presented in Figure 17 scenario (a) represents a method defined in the base code (not just an *auxiliary method* of an

advice as described before), which is integrated with another method m1 (of the base code) through the advice a1. The advice responsible for *integrating* two existing functionalities from the base code is called "*integration advice*". If a method m4 throws an exception, this exception will flow through method m1, which may not necessarily handle it. Since such functionalities would not be related if the aspect *weaving* did not take place, in our approach the *integration advice* takes the responsibility of signaling the exception that will flow from m4 to m1.

Another interesting scenario is depicted by Figure 17 (b). It represents a scenario in which the method called by an advice is itself affected by another advice. In exception handling scenarios where more than one advice takes place, we assign the "*exception signaling*" responsibility to the advice that first adds a behavior that brings a new exception. Therefore, in Figure 17 (b) although method m4 implements part of the functionality of advice a1, in our approach the crosscutting concern responsible for signaling the exception is advice a2.

5.2.2. Exception Paths in AO Systems

As mentioned before, the *exception path* starts in the node *responsible* for signaling the exception and ends at the node that contains a catch clause that handles it. In case the exception remains uncaught, the last node of the *exception path* represents the program entry point. Thus, adopting the heuristics defined previously, the exception paths can be found in the following way:

- The signaler is a class when an application method (i) explicitly throws an exception or (ii) calls a method from a library that implicitly throws an exception - and it is not directly or indirectly used by an *advice method.*
- The signaler is an aspect when an *advice method* (or *intertype* method[21]), (i) explicitly throws an exception, (ii) calls a method from a library that throws an exception or (iii) calls an *application method* that throws an exception and no method that can be called from it is advised by an aspect.

[21] An aspect has the ability (through intertype declarations) to add public or private methods, field, or interface implementation declarations into a class. The *intertype methods* are therefore the methods added by an aspect into a class through intertype declaration.

The exception signaler is, therefore, the first advice on the *exception path* from the element that explicitly throws an exception to the element that handles it. Or, if no advice is found, the method that explicitly throws the exception is assigned the signaler responsibility. This heuristic is central to the SAFE tool implementation. Based on this heuristic, the SAFE tool classifies the exception paths according to their *Signaler-Handler* relationship. This relation assigns the responsibility for the exception signaling among the classes or aspects that define the application and can give an overview about what are the actions taken on the exceptions signaled by crosscutting concerns (directly through the throw statement or indirectly through library or application method calls).

5.2.2.1. Exception Paths Originating from Exception Softening

Another heuristic is applied when an exception is softened in a program. Figure 18 illustrates two scenarios, one in which the exception thrown by an application method is *softened* (see Figure 18 (a)); and other in which the exception signaled by an advice is softened (see Figure 18 (b)).

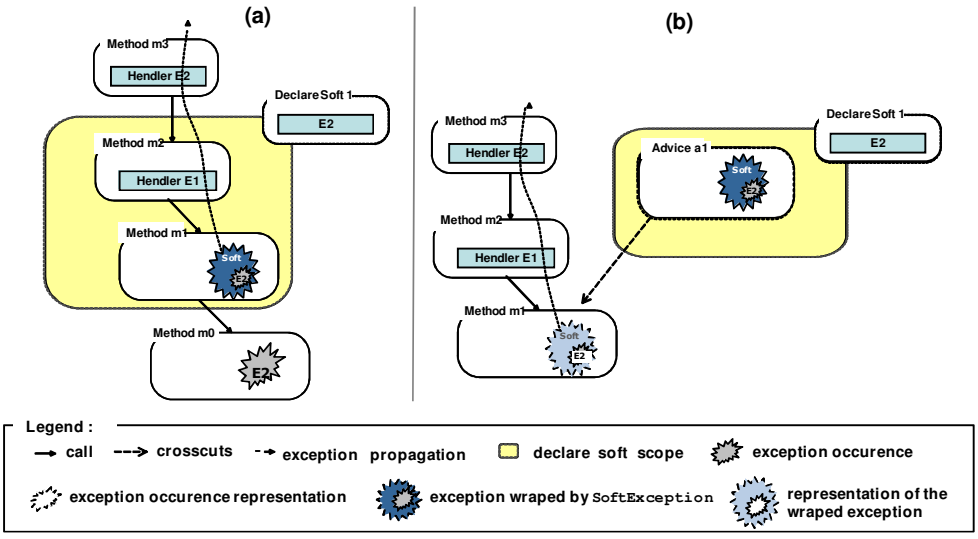

Figure 18. Exception softening scenarios.

As mentioned before, when an exception is softened within a specific scope (defined by a pointcut expression) an exception object is converted into an

instance of `SoftException`. In the woven *bytecode,* softening an exception corresponds to: (i) handling the original exception; and (ii) throwing an instance of `SoftException` that wraps the original exception.

Therefore, every time that one exception is softened, two exception paths are generated: (i) one from the site where the *original exception* is thrown until the point in the code where it is handled to be converted into a `SoftException`; and (ii) another one that starts where the instance of `SoftException` is thrown until it is handled somewhere in the code. Thus, in our approach, the exception paths reported for scenario (a) in Figure 18 are: <m0→m1> for exception E2 and <m1→m2→ m3→… > for the instance of `SoftException` that encapsulates E2. Similarly, in scenario (b), <a1> is the exception path of E2, and <a1→m1→ m2→ m3→… > is the exception path of `SoftException` instance.

5.2.2.2. Exception Paths Originating from Library Methods

When looking for the exceptions that may escape from an application or an advice method, the *exception-flow* analysis algorithm verifies each exception explicitly thrown inside the method or implicitly thrown by library method calls. When an exception escapes from a library method, we are not interested in the *internal library methods* that caused this exception – which is part of the exception interface of the library method. Figure 19 illustrates a library method *L1* whose *exception interface* is comprised of a set of exceptions signaled (and not handled) by the methods called from it.

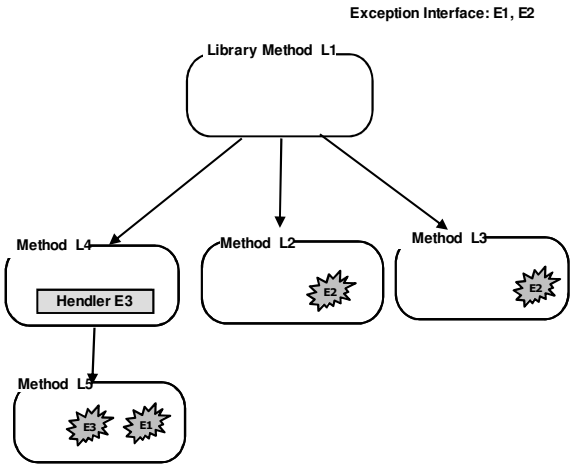

Figure 19. The exception types and the *exception interface of a method*.

In our approach, the *exception paths* calculated for exceptions that escape from library methods do not include the method call chain inside the library - which may contain the *internal library method* that actually signaled the exception. Hence, library methods, application methods and advice methods should be treated differently by the exception analysis algorithm: the exception paths that originate from aspect libraries may not include the actual exception signaler (i.e. the *internal library method* that signaled the exception).

5.3. The SAFE Tool

We developed SAFE (*Static Analysis for the Flow of Exceptions*) to statically analyze the woven *bytecode* of AspectJ programs in order to find out: (i) the *exception interfaces* of application and advice methods; and (ii) the *exception paths* of each exception signaled on them. To mine such information, the SAFE tool implements an *exception-flow analysis* procedure that implements the heuristics defined above (see Section 5.2).

Our tool is based on the Soot framework for static analysis of *bytecode* (Section 5.3.2). It uses SPARK, one of the call graph builders provided by Soot, (Section 5.3.3), used by other exception analysis tools (Fu et al., 2005; Fu and Ryder, 2007). The SAFE tool considers all checked and unchecked exceptions, explicitly thrown by the application or implicitly thrown (e.g. via library methods) by aspects and classes on the application code.

5.3.1. The Tool's Architecture

The SAFE tool comprises approximately 20.200 lines of Java source code structured in 84 classes[22]. Figure 20 depicts the main elements that comprise the SAFE architecture – it omits dependence relations between classes for the sake of simplicity. SAFE was developed on top of Soot a framework for Java *bytecode* analysis, and is structured in three main components: the *Call Graph, the Exception Analysis and the Exception Path Miner.* The *Util* package presented in Figure 20 contains auxiliary classes used by these three components.

The *Call Graph* component comprises the classes responsible for building the program call graph; this module re-uses the call graph builder defined in Soot framework. The *Exception Analysis* component comprises a set of classes responsible for traversing the program call graph finding the exception interfaces of methods and advices and calculating the exception flow of exceptions (`ExceptionAnalysis` class); it also contains an `ExceptionFilter` that excludes a specified set of exceptions from being analyzed (see Section 5.4.3). Moreover, there is an element responsible for storing the exception paths in a structured way (`ExceptionPathStorage` class). The *Exception Path Miner* component comprises the classes that classify each exception path according to its signaler (i.e., class method, aspect advice, intertype or `declare soft` constructs) and handler. Such classification helps the developer to discover the new dependencies that arise between aspects and classes on exceptional scenarios.

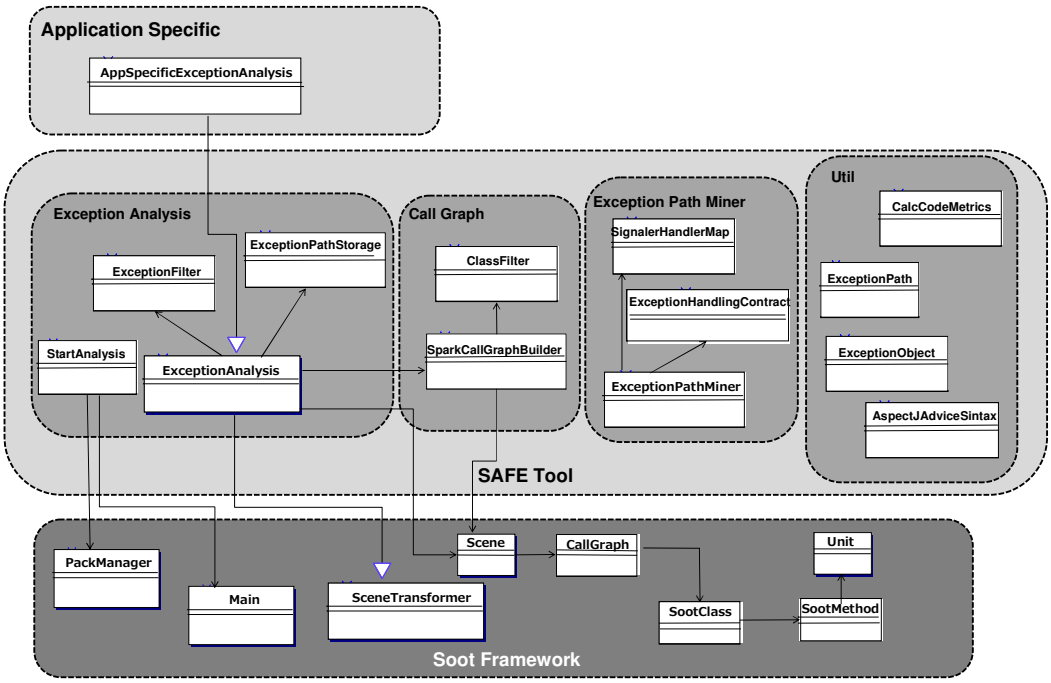

Figure 20. The SAFE tool architecture.

[22] This metric excludes the amount of Java code reused form libraries (e.g., Soot framework).

5.3.2. The Exception Analysis Component

The *Exception Analysis* component implements an *interprocedural* algorithm to find handlers for exceptions that escape from every application or advice method. The algorithm is divided into three major phases: (1) *constructing the program call graph*; (2) *finding the exceptions signaled by every application and advice method*; (3) *calculating the exception paths per signaled exception*. The pseudo code presented below briefly illustrates each one of these phases, omitting some implementation details.

The first step of the algorithm is the construction of a variation of the program call graph (as detailed in Section 5.1.2). The element responsible for building the program call graph is the `SparkCallGraphBuilder`. This element reuses one of the call graph builders available in Soot detailed in Section 5.3.3. Once the call graph was built, every *application method* and *advice method* is analyzed in order to discover which exceptions can be signaled from them. The set of exceptions that can be signaled by a method comprises: (i) the exceptions explicitly thrown by throw statements, and (ii) implicitly thrown by library method calls.

When a throw statement signaling exception *e1* is found in a method body *m1*, the algorithm checks whether it is contained within any try-catch block, and if there is any handler defined to *e1* (or to any of its *supertypes*). If the throw statement is not protected by a try-catch block or no handler is defined for *e1* in a try-catch block surrounding it, *e1* is included in the list of exceptions that can escape from the method. If *e1* is handled inside *m1* this *exception path* of *e1,* which starts and ends in *m,* is stored.

When a call to a library method is found, it is then recursively analyzed. After the *exception interface* of the library method is calculated, the algorithm checks whether the method call is contained within any try-catch block, and if there is any handler defined to the exceptions that comprise the exception interface of the library method (or to any of its *supertypes*). Thus, all exceptions not adequately handled inside the analyzed method are included on the list of exceptions that escape from it. At this step the algorithm uses the exceptional flow information defined on the inner structure of call graph vertices presented above

(Section 5.1.2). The algorithm does not recursively analyze the application methods called from another application method since every application method is analyzed on its own. Listing 5 offers an example to illustrate how this step works.

```
1. package myapplication;
2.  ...
3. int retrieveDataFromReport() throws FileNotFoundException{
4.   …
5.   FileReader fr = new FileReader(path);
6.   BufferedReader bf = new BufferedReader(fr);
7.   try{
8.     String    aux = bf.readLine();
9.     int sum = Integer.parseInt(aux);
10.     if (sum < 100) {
11.       throw new IncompleteRecordException();
12.     }
13.   }catch(IOException ex){
14.     io.printStacktrace();
15.   }
16.   LogLibrary.log("End of operation… ");
17. }
```

```
1. package loglibrary;
2. public class LogLibrary {
3. void log(String message) {
4.    if(!pingLogServer()){
5.      throw new LogFileUnavailableException();
6.    }
7.    …
8. }
```

Listing.5. Looking for the exceptions signaled by an application method.

In Listing 5, the method `retrieveDataFromReport` throws only one checked exception (i.e., `FileNotFoundException`) declared on its signature. To find out the unchecked exceptions that can be signaled from them, the algorithm analyzes every throw statement, and every library method call, and checks whether or not the exceptions signaled from them are handled inside the method. Thus, the exception-analysis algorithm analyzes every statement of the `retrieveDataFromReport` method looking for: (i) throw statements or library method calls. When a library method is found it is recursively analyzed: every

throw statement and method called from it is analyzed[23]. The algorithm stops when every throw statement and library method call is analyzed and reports the list of exceptions that can be signaled from the `retrieveDataFromReport` method: `LogFileUnavailableException`, `IncompleteRecordException`, and `FileNotFoundException`.

Through this example we can observe why the exception flow analysis cannot be performed as an intraprocedural analysis: it needs to recursively analyze the library code called from an analyzed method to discover which exceptions can be signaled from it.

After the algorithm finds every exception that can be signaled by application and advice methods, its third phase starts. For each of these exceptions the algorithm traverses the program call graph - in the reverse direction with respect to the execution flow - looking for a proper handler to the exception. This *backward* data-flow analysis simulates the exception propagation that happens during runtime. During the *backward* analysis, each method (vertex in the program call graph) in which the exception propagates is recorded as part of the *exception path*. Thus, the *exception interface* of each application or advice method is comprised of: the exception types that can be signaled from it (discovered on the second step of the algorithm) and the exception types that propagate from it (discovered when calculating the exception paths of each exception).

5.3.2.1. Integration with Soot Framework

As illustrated in Figure 20, SAFE tool is developed on top of the Soot framework for Java *bytecode* analysis. The Soot framework adopts the Pipes and Filters architecture style (Buschmann et al., 1996). It divides the *bytecode* processing task into a sequence of smaller, independent processing phases called *packs*. Therefore, the *bytecode* analysis triggers a sequence of processing packs, each performing a specific set of functions.

The first pack parses class or source java files and produces the Jimple code, an intermediate representation available on Soot (see Section 5.4.1) to be fed into

[23] To deal with interprocedural loops, the algorithm recursively analyzes each method in a call chain only once.

the other packs. Each pack comprises a set of sub-phases that can perform either *intraprocedural* analysis (i.e. act on individual methods) or *interprocedural* analysis (i.e., acts on the whole set of available classes). Each pack works on a specific intermediate representation (see Section 5.4.1).

Soot is extensible by adding new phases that perform a required analysis or program transformation. Soot provides the `BodyTransformer` abstract class to represent all intraprocedural sub-phases, and the `SceneTransformer` abstract class to represent the interprocedural sub-phases. Since the exception flow analysis needs a whole program representation (i.e., the program call graph), we extended the `SceneTransformer` class as illustrated in Figure 20. Listing 6 below presents the partial code of the `ExceptionAnalysis` class. It defines the `internalTransform` method that implements the exception analysis algorithm detailed in previous section.

```
public abstract class ExceptionAnalysis
                                    extends SceneTransformer{
   abstract boolean isAnAspect();
   abstract String[] getApplicationPackages();
   ...

  protected void internalTransform(String phase, Map arg1) {

    //1. Call graph construction
    cg = Scene.v().getCallGraph();
    Iterator itsce = Scene.v().getApplicationClasses().iterator();

    //Iterate per class
    while (itsce.hasNext()) {

         SootClass currentClass = (SootClass) itsce.next();
                ...
        Iterator itmet = currentClass.getMethods().iterator();

          while (itmet.hasNext()) {
            ExThrowableSet metExceptions = new ExThrowableSet();
            SootMethod met = (SootMethod) itmet.next();
            ...
           metExceptions =findEscapeExceptions (b,met,callChain);
            Iterator itExc = metExceptions.iterator();

            while( itExc.hasNext() ){

                SootClass excecao = (SootClass) itExc.next();
                 if(!shouldFilter(excecao.getType().getClassName()){
                      ExecutionStack stack = new ExecutionStack();
                     boolean escapa = true;
                      lookforHandler (met,excecao,stack);
                }//if
            }// while
        }// while method
    }//while class
  }
}
```

Listing.6. Code snippet for `ExceptionAnalysis` class.

The `ExceptionAnalysis` also defines two abstract methods, `isAnAspect` and `getApplicationPackages`, which should be implemented by the `ExceptionAnalysis` defined per analyzed application (see `AppSpecificExceptionAnalysis` in Figure 20). Each application should specify the application packages and the aspects that comprise it. The `getApplicationPackages` is useful to distinguish between library methods and application methods. Such information is needed to implement the heuristics defined in Section 5.2.

To execute the exception analysis, one instance `ExceptionAnalysis` is created and added as a new sub phase of a Soot pack. The code snippet below illustrates the piece of code responsible for adding `ExceptionAnalysis` in one of the Soot packs.

```
1. public static void main(String[] args) {
2.
3.    if(args.length == 0) {
4.    System.exit(0);
5.    }
6. PackManager.v().getPack("wjtp").add(newTransform("wjtp.SAFE",
            AppSpecificExceptionAnalysis.getInstance()));
7.
8.    soot.Main.main(args);
9. }
```

Listing.7. Adding a new phase to Soot.

The `ExceptionAnalysis` transformer is added to a Soot pack called `wjtp` (whole Jimple transformation pack). The `wjtp` pack comprises every transformation or analysis that works on Jimple intermediate transformation and needs a whole program representation (i.e., the program call graph) to perform an interprocedural analysis.

5.3.3. The Call Graph Builder

The component responsible for building the call graph in the SAFE tool uses the Soot Pointer Analysis Research Kit (SPARK)(Lhotak, 2002), one of the

call graph builders provided by Soot. The SPARK framework uses a points-to analysis algorithm to improve the precision of call graph construction (Lhotak, 2002).

The goal of points-to analysis is to compute the set of possible targets that can be referred by a *calls site*. In AspectJ, as in Java, instance methods' invocation is based on dynamic dispatch (Gosling et al., 1996; Colyer, 2004) - the target method of a call site is selected at runtime depending on a run-time value of a reference variable. This means that the whole program analysis requires an approximation of the program *call graph*. The simplest algorithm called Class Hierarchy Analysis (CHA) (Grove and Chambers, 2001), presented in Chapter 2, assumes that every reference might point to every other variable of its subtype. However, it is conservatively sound (Grove and Chambers, 2001) and not accurate. The SPARK is based on Andersen's points-to analysis algorithm, which provides a more accurate analysis despite requiring more processing time and memory (Grove and Chambers, 2001; Lhotak, 2002).

SPARK's algorithm is *field-sensitive* (Mine, 2006) (it distinguishes between different fields of an object), *context-insensitive* (Liang et al., 2001) (it does not distinguish between different calling contexts of the same method) and *flow-insensitive* (Foster et al., 2002) (the analysis does not take into account the order in which program statements are executed). For more details about SPARK's points-to analysis algorithm please refer to (Grove and Chambers, 2001; Milanova et al., 2002; Lhotak and Hendren, 2006). Empirical studies (Grove and Chambers, 2001; Lhotak, 2002; Rountev et al., 2004) have shown that a call graph generated by SPARK algorithm contains less *spurious* method call chains when compared to other *context-insensitive* call graph construction algorithms such as CHA – which is used by most inter-procedural analysis.

In the empirical study reported by Rountev et al. (2004) the call graph generated using CHA reported approximately 26,5% of *spurious method call chains* (i.e., infeasible sequence of method calls). And, in the same study, the call graph build using SPARK's algorithm was comprised of 6,4% of spurious call chains.

5.3.4. The Exception Path Miner

The *Exception Analysis* component presented in Section 5.3.2 calculates the *exception paths* of every exception that escapes from aspect advice and application methods. The exception paths are stored in structured files as illustrated in Listing 8 below. The Listing 8 presents one exception path found in Health Watcher system described in Chapter 3, Section 3.1.1.

```
<exceptionPath exception=org.aspectj.lang.SoftException>
  <calltchain>
    <method>healthwatcher.aspects.persistence.HWTransactio
    nManagement:voidajc$afterThrowing$healthwatcher_aspects
    _persistence_HWTransactionManagement$3$a03b16aa()</method>
    <method>healthwatcher.business.HealthWatcherFacade:lib
    .util.IteratorDskgetSpecialityList()</method>
    <method>healthwatcher.view.servlets.ServletGetDataForSear
     chBySpeciality: void doGet_aroundBody0(healthwatcher.vi
     ew.servlets.ServletGetDataForSearchBySpeciality,javax.se
     rvlet.http.HttpServletRequest,javax.servlet.http.HttpSer
     vletResponse)</method>
   </calltchain>
   <action type=”ExceptionSubsumption” detail=”org.aspectj.
   lang.SoftException capturedBy java.lang.Exception”/>
</exceptionPath>
```

Listing.8. Structured representation of an *exception path.*

The `ExceptionPathMiner` class (see Figure 21) parses the exception path files and classifies each exception path according to its *Signaler-Handler* relationship defined by the heuristics presented in Section 5.2.

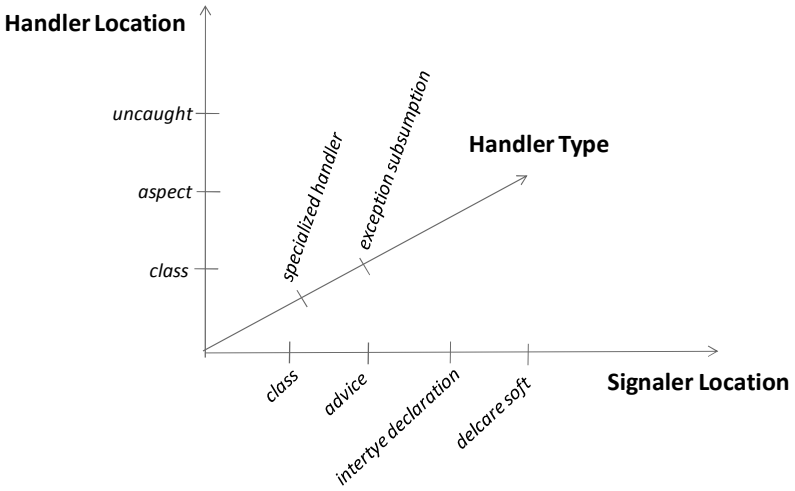

Figure 21. Three-dimension classification for exception paths.

Figure 21 presents a three-dimension classification, in which each axis represents one property of the exception path for which a set of values are permitted. The first axis specifies the exception signaler, which can be: a class method, an advice, a method defined as intertype declaration, or a `declare soft` construct. The second axis defines the *exception handler* associated with the exception path. The exception can be handled by a class method, by `after` and `around` advice (Filho et al., 2006; Filho et al., 2007), or the `declare soft` construct. The `declare soft` can play both roles (i.e., exception signaler and handler) because the moment an exception is softened the declare soft construct *handles* the exception that was softened, and *signals* an instance of `SoftException` that wraps the original exception. When no handler is defined for an exception it remains *uncaught* and reaches the application entry point (i.e., `main` method), causing a software crash. The third axis represents the handler action. An exception occurrence can be caught in two basic ways. It can be caught by a *specialized handler* when the `catch` argument is the same type of the caught exception type. Alternatively, it can be caught by *subsumption* when the `catch` argument is a supertype of the exception being caught.

The `ExceptionPathMiner`, that classifies the exception paths according to the properties presented above, also allows the SAFE tool user to specify simple *exception handling contracts* (see the `ExceptionHandlingContract` class in Figure 20). These *contracts* define the *Signaler-Handler* relation that exceptions

thrown in the program should obey. An *exception handling contract* comprises the following information: (i) the name of the signaled exception; (ii) the signature of the method that signals it; (iii) the method signature that handles it; and (iv) the handler type, which can be `same_exception` when the exception is handled by a handler of the same type, or `subsumption` when the exception is handled by *subsumption*.

```
<exception type="">
   <signaler signature="" />
   <handler signature=""  type="" />
</exception>
```

Listing.9. Structure of an exception handling contract.

These contracts can be automatically checked during the exception path classification. After parsing and classifying the exception paths, the *Exception Path Miner* may calculate additional information concerning all exception paths, such as: (i) the number of exceptions thrown by aspects that remained uncaught; (ii) the number of exceptions thrown by classes that remained uncaught; (iii) the number of exceptions thrown by aspects and caught by *subsumption* on classes; (iv) the number of exceptions thrown by classes and handled by aspects; (v) the most frequently thrown exceptions; (vi) the exceptions most frequently thrown by aspects; (vii) the exceptions most frequently thrown by classes.

The `ExceptionPathMiner` output (the exception paths classification, the contract checking and the extra quantitative information) is reported on .xls files that can be used by developers to the guide the manual code inspections of the exception handling code.

5.4. Implementation Details

This section presents implementation details concerning intermediate representation of *bytecode* available in the Soot framework and used by the SAFE tool (Section 5.4.1). Moreover, it details the low-level design decisions made during the implementation of SAFE (Sections 5.4.2 and 5.4.3).

5.4.1.
Soot Intermediate Representation

The SAFE tool is implemented on top of the Soot[24] Java Analysis and Transformation Framework version 2.0.1 (Vallée-Rai, 2000). The static analysis is not performed on the raw *bytecode,* but on an intermediate representation for Java *bytecode* available in Soot, called Jimple (Vallée-Rai, 2000; Einarsson and Nielsen, 2007). Jimple is a typed, 3-address, statement-based intermediate representation. It is called 3-address because every expression in Jimple only references at most three local variables or constants. Every possible instruction in Java *bytecode*[25] are represented by the 15 statements available in the Jimple representation without loss of information (Einarsson and Nielsen, 2007). It results in a regular and very convenient representation for performing code analysis. Listing 10 presents the source code of a simple Java method. Listings 11 and 12 illustrate the *bytecode* representation and the Jimple representation of this method, respectively.

```
1. public static void main(String[] args) {
2.     Example f = new Example();
3.     int a = 2;
4.     int b = 3;
5.     int x = (f.method(4) + a) * b;
6. }
```

Listing.10. Code snippet of a Java method.

```
public static void main(java.lang.String[])
Max stack size: 2
Max local variables: 5
Code size: 24
   0 new #2 <Class test.Example>
   3 dup
   4 invokespecial #17 <Method Example()>
   7 astore_1
   8 iconst_2
   9 istore_2
  10 iconst_3
  11 istore_3
  12 aload_1
  13 iconst_4
  14 invokevirtual #21 <Method int method(int)>
  17 iload_2
  18 iadd
  19 iload_3
  20 imul
```

[24] http://www.sable.mcgill.ca/
[25] The java bytecode allows approximately 200 different intructions.

```
  21 istore_4
  23 return
```

Listing.11. The *bytecode* representation of a Java method.

```
 1. public static void main(java.lang.String[]) {
 2.   java.lang.String[] r0;
 3.   Example $r1, r2;
 4.   int i0, i1, i2, $i3, $i4;
 5.   r0 := @   parameter0: java.lang.String[];
 6.   $r1 = new Example;
 7.   specialinvoke $r1.<Example: void <init>()>();
 8.   r2 = $r1;
 9.   i0 = 2;
 10.  i1 = 3;
 11.  $i3 = virtualinvoke r2.<Example: int method()>(4);
 12.  $i4 = $i3 + i0;
 13.  i2 = $i4 * i1;
 14.  return;
15. }
```

Listing.12. Jimple representation of a Java method.

In the Jimple code illustrated in Listing 12 we can recognize the statement-based structure from Java, and the method invocation style is similar to the one in Java *bytecode*. The variables without the prefix $ represent user defined local variables, and the ones starting with a $ prefix represent stack positions. We can also observe that every expression obeys the 3-address form.

5.4.2. Exceptional Control Flow

As mentioned before, in AspectJ language, in addition to methods, an aspect may contain other modular units such as advice and inter-type methods. In this section we use the word “method” to refer to a piece of advice, an inter-type method, a method in an aspect or a method in a class - since advice and inter-type methods are converted to methods by aspect weaver. The Listings below show a Java method and its corresponding Jimple code.

```
1. public class CFGExample  {
2.   int m(int[] a, int i, int j) {
3.   int sum = 0;
4.   try {
6.       this.printStatus();
7.       sum += a[k];
8.   } catch (NullPointerException e) {
8.      return 0;
10.  } catch (RuntimeException e) {
```

```
11.     sum += 0;
12.  }
13. return sum;
14.}
```

Listing.13. Piece of Java method to illustrate the internal structure of a call graph vertex in our approach.

```
1. int m(int[], int, int) {
2. CFGExample r0;
3. int[] r1;
4. int i0, i1, i2, i3, $i4;
5. java.lang.NullPointerException r2, $r3;
6. java.lang.ArrayIndexOutOfBoundsException $r4, r5;
7. r0 := @this;
8. r1 := @parameter0;
9. i0 := @parameter1;
10. i1 := @parameter2;
11. i2 = 0;
12. i3 = i0;
13. virtualinvoke r0.<void printStatus()>();
14.    $i4 = r1[i3];
15. i2 = i2 + $i4;
16. label3:
17. $r3 := @caughtexception;
18. r2 = $r3;
19. return 0;
20. label4:
21. $r4 := @caughtexception;
22. r5 = $r4;
23. i2 = i2 + 0;
24. label5:
25. i3 = i3 + 1;
26. goto label0;
27. label6:
28. return i2;
29.
30. catch java.lang.NullPointerException from label1 to
    label2 with label3;
31. catch java.lang.RuntimeException from label1 to label2
    with label4;
```

Listing.14. Jimple representation of a Java method.

Figure 22 illustrates the internal structure of method *m* presented above in the call graph. The internal structure stores a partial representation of the method's control flow graph. Each statement is represented by a node in a linked list, which is linked to the next statement in the code. When the statement is protected by a try-catch block another link is added between the protected statement and the catch clause statement – the catch statement node specifies the type of the catch clause parameter (see the nodes identified with labels 13 and 14 in Figure 22). For the sake o simplicity this structure omits the statements defined inside catch clauses.

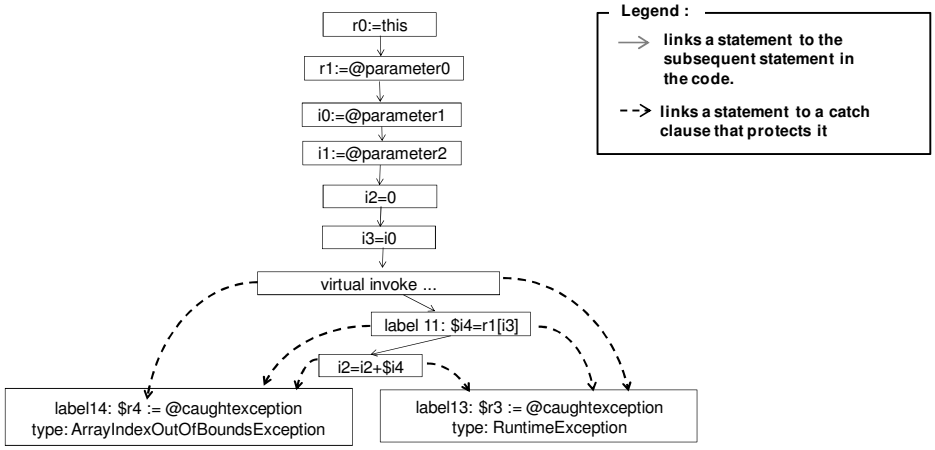

Figure 22. Call graph vertex internal structure.

5.4.3. Exception Filtering

The SAFE tool considers all checked and unchecked exceptions. In languages such as AspectJ, where many unchecked exceptions can be thrown by the Java virtual machine (e.g., `NullPointerException`, `IllegalMonitorStateException`, `ArrayIndexOutOfBoundsExcep-tion`, `ArrayStoreException`, `NegativeArraySizeException`, `ClassCastException`, `ArithmeticException`) almost every operation may throw an unchecked exception (thrown by JVM). Thus, including every unchecked exception can generate too much information, which may affect the usability of the exception analysis (Chang et al., 2001; Jo et al., 2004). For this reason, the SAFE tool defines the `ExceptionFilter` component that enables the developer to filter the exceptions that should not be considered during the

analysis. The list of exceptions that should be filtered is defined on an XML file that is loaded just before the exception analysis begins, by the `ExceptionFilter` component.

5.4.4. Dealing with Exception-Related AspectJ Weaver Residues

Residues of AspectJ constructs explicitly affect the *exception interface* of advised methods. For instance, consider the method presented in Listing 15 (extracted from Health Watcher system presented in Chapter 3). This method should be intercepted by three different advices (i.e., a `before` advice, an `after returning` advice and an `after throwing` advice).

```
1. public void updateComplaint(Complaint complaint) throws
2.    ObjectNotFoundException, ObjectNotValidException {
3.      complaintRecord.update(complaint);
4.}
```

Listing.15. Code snippet of a method before the weaving process.

The code snippet below illustrates the decompiled *woven bytecode* of the method presented in Listing 15. As detailed before, every advice is converted into a static method that is called at specific points in the code (i.e., join points). Besides including method calls to the *advice methods* the AspectJ weaver also includes additional code on the advised methods. Such code is known as *residue of the match*. In the Listing below the *advice methods* are represented in gray and the *residues of the match* are underlined. Note that the list of exceptions declared on the method's signature changed in the woven bytecode.

```
1.  …
2.
3.  public void        (Complaint  r1) throws
4.        ObjectNotFoundException, ObjectNotValidException,
5.        Throwable, TransactionException {
6.
7.   try{
8.     HWTransactionManagement.aspectOf().ajc$before$healthwatcher
       _aspects_persistence_HWTransactionManagement$1$a03b16aa();
10.
11.     complaintRecord.update(r1);
```

```
12.
13.     HWTransactionManagement.aspectOf().ajc$afterReturning$healt
hwatcher_aspects_persistence_HWTransactionManagement$2$a03b16aa();
14.      return;
15. }
16. catch (Throwable $r6) {
17.
18.   HWTransactionManagement.aspectOf().ajc$afterThrowing$healthwa
   tcher_aspects_persistence_HWTransactionManagement$3$a03b16aa();
19.       throw $r6;
20. }
21.}
```

Listing.16. Code snippet of a method after the weaving process.

In this example the *residue of the match* is the additional code that handles and re-throws instances of `Throwable`. The residue of `after throwing` advice was to catch-clause in line 16 and the re-throw statement in line 19. A naive exception analysis algorithm would generate exception paths for the instance of `Throwable` re-thrown in line 19. However, this *exception handling and re-signaling pattern* is a *residue* of the `after throwing` advice. For that reason our exception identifies and ignores the pieces of code that represent *residues of the math* – which would generate misleading exception flow information.

The `declare soft` construct also generates similar *residues*. As explained before, this construct wraps an exception in an instance of `SoftException` and re-throws it. The code snippet below shows the decompiled woven *bytecode* of an advice affected by the `declare soft` construct[26].

```
1. public class HWTransactionManagement {
2.    public void ajc$before$healthwatcher_aspects_persistence_
3.              HWTransactionManagement$1$a03b16aa() throws
4.              lib.exceptions.TransactionException {
5.
6.        IPersistenceMechanism $r2;
7.        $r2 = this.getPm();
8.        try
9.        {
10.            $r2.beginTransaction();
11.       }
```

[26] This code was also from Health Watcher system.

```
12.       catch (TransactionException $r3) {
13.          if ( ! ($r3 instanceof RuntimeException)) {
14.                throw new SoftException($r3);
15.          }else{
16.                throw $r3;
17.           }
18.        }
19.    }
```

Listing.17. AspectJ weaver residues associated with the `declare soft` construct.

Since `SoftException` is an unchecked exception, it does not need to be explicitly declared on the methods signature as part of its *explicit exception interface* (lines 2-4). A naive exception analysis algorithm would calculate the exception path for the two exceptions thrown in this method body (lines 14 and 16). However, in this scenario only one exception will be thrown in runtime: an instance of `SoftException` if the exception being softened is not an instance of `RuntimeException` or the original exception. Our exception analysis algorithm keeps track of the type associated with each application exception, and in a scenario such as the one presented in Listing above only one exception occurrence is considered.

5.5. Tool's Performance

We have executed the SAFE tool to find the exception paths of the object-oriented versions and aspect-oriented versions of the three medium-sized systems described in Chapter 3. It was executed on a PC with an Intel Core2 2GHz processor using Windows XP with 1GB of RAM memory. The time spent to calculate the exception paths for each system is presented in Table 11. In Table 11 the *call graph size* comprises the number of application methods, advice methods and the methods reachable from them (e.g., methods defined on libraries); the line identified by the *"Complete analysis"* attribute comprises the processing time presented (in seconds) to complete the exception analysis; the line identified by *"Call graph construction*" comprises the time spent to build the program call graph before (the first step of the *exception analysis*).

	Health Watcher		Mobile Photo		JHotDraw
	V1	V9	V4	V6	V1
Object-Oriented					
Call graph size (# nodes)	94.738	48.263	44.933	46.241	88.199
Call graph construction (# seconds)	3.157,8	404,2	398	403,4	9.977,2
Complete analysis (# seconds)	3.354	453	587	441	10.720
Aspect-Oriented					
Call graph size (# nodes)	47.622	49.567	45.344	46193	89.794
Call graph construction (# seconds)	416,8	472,1	408,2	399,6	8.682,2
Complete analysis (# seconds)	596	540	676	422	9.610

Table 11. The exception analysis performance using the SAFE tool.

Listing 18 represents one of the exception paths found by the exception analysis of AJHotDraw[27] - that system whose exception analysis required the highest processing time (9.610 seconds). The *exception path* illustrated in Listing 18 includes both *application* and *library* methods. To distinguish between both kinds of methods, application method calls are shaded while method calls inside libraries are not.

```
1. org.jhotdraw.contrib.WindowMenu: void buildChildMenus()
2. org.jhotdraw.contrib.WindowMenu: void access$1()
3. org.jhotdraw.contrib.WindowMenu$3: void menuSelected()
4. javax.swing.JMenu: void fireMenuSelected()
5. javax.swing.JMenu$MenuChangeListener: void stateChanged()
6. javax.swing.text.DefaultCaret: void fireStateChanged()
7. javax.swing.text.DefaultCaret: void changeCaretPosition()
8. javax.swing.text.DefaultCaret: void handleSetDot()
9. javax.swing.text.DefaultCaret: void setDot()
10. javax.swing.text.DefaultCaret: void setDot()
11. javax.swing.text.JTextComponent: void
    setInputMethodCaretPosition()
12. javax.swing.text.JTextComponent: void processInputMethodEvent()
13. java.awt.Component: void processEvent()
14. java.awt.Container: void processEvent()
15. java.awt.Window: void processEvent()
16. java.awt.Component: void dispatchEventImpl()
17. java.awt.Container: void dispatchEventImpl()
18. java.awt.Window: void dispatchEventImpl()
19. java.awt.Component: void dispatchEvent()
20. java.awt.Container: void addImpl()
21. javax.swing.JFrame: void addImpl()
```

[27] Parameters are omitted for readability.

```
22. java.awt.Container: void add()
23. javax.swing.JFrame: void setRootPane()
24. javax.swing.JFrame: void frameInit()
25. javax.swing.JFrame: void <init>()
26. sun.awt.im.InputMethodJFrame: void <init>()
27. sun.awt.im.InputMethodContext: java.awt.Window
    createInputMethodWindow()
28. sun.awt.im.CompositionArea: void <init>()
29. sun.awt.im.CompositionAreaHandler: void createCompositionArea()
30. sun.awt.im.CompositionAreaHandler: void grabCompositionArea()
31. sun.awt.im.InputMethodContext: void grabCompositionArea()
32. sun.awt.im.InputContext: void activateInputMethod()
33. sun.awt.im.InputContext: void changeInputMethod()
34. sun.awt.im.InputContext: boolean selectInputMethod()
35. sun.awt.im.InputContext: void <init>()
36. sun.awt.im.InputMethodContext: void <init>()
37. java.awt.im.InputContext: java.awt.im.InputContext getInstance()
38. java.awt.Window: java.awt.im.InputContext getInputContext()
39. java.awt.Component: void removeNotify()
40. java.awt.Container: void removeDelicately()
41. java.awt.Container: void setComponentZOrder()
42. javax.swing.JLayeredPane: void setLayer()
43. javax.swing.JLayeredPane: void setLayer()
44. javax.swing.JLayeredPane: void addImpl()
45. java.awt.Container: java.awt.Component add()
46. javax.swing.SwingUtilities: … getCellRendererPane()
47. javax.swing.SwingUtilities: void paintComponent()
48. org.jhotdraw.contrib.html.HTMLTextAreaFigure: float renderText()
49. org.jhotdraw.contrib.html.HTMLTextAreaFigure: void generateImage()
50. org.jhotdraw.contrib.html.HTMLTextAreaFigure: float drawText()
51. org.jhotdraw.contrib.html.HTMLTextAreaFigure: void draw()
52. org.jhotdraw.standard.CompositeFigure: void draw()
53. org.jhotdraw.standard.StandardDrawingView: void drawDrawing()
54. org.jhotdraw.standard.StandardDrawingView: void drawAll()
55. org.jhotdraw.standard.SimpleUpdateStrategy: void draw()
56. org.jhotdraw.standard.StandardDrawingView: void paintComponent()
57. javax.swing.JComponent: void paint()
58. org.jhotdraw.contrib.zoom.ZoomDrawingView: void paint()
59. javax.swing.JViewport: void paintViewDoubleBuffered()
60. javax.swing.JViewport: void paintView()
61. javax.swing.JViewport: void flushViewDirtyRegion()
62. javax.swing.JViewport: void setViewPosition()
63. javax.swing.ViewportLayout: void layoutContainer()
64. java.awt.Container: void layout()
```

```
65. java.awt.Container: void doLayout()
66. java.awt.Container: void validateTree()
67. java.awt.Container: void validate()
68. java.awt.Window: void show()
69. java.awt.Component: void show(boolean)
70. java.awt.Component: void setVisible()
71. javax.swing.JComponent: void setVisible()
72. javax.swing.JRootPane: java.awt.Component createGlassPane()
73. javax.swing.JRootPane: void <init>()
74. javax.swing.JInternalFrame: javax.swing.JRootPane createRootPane()
75. javax.swing.JInternalFrame: void <init>()
76. org.jhotdraw.contrib.MDIDesktopPane: … createContents()
77. org.jhotdraw.contrib.MDIDesktopPane: void addToDesktop()
78. org.jhotdraw.contrib.MDI_DrawApplication: void newWindow()
79. org.jhotdraw.samples.javadraw.JavaDrawApp$4: void execute()
80. org.jhotdraw.samples.javadraw.JavaDrawApp: void
    execute_aroundBody0()
81. org.jhotdraw.samples.javadraw.JavaDrawApp: void
    execute_aroundBody1$advice()
82. org.jhotdraw.samples.javadraw.JavaDrawApp: void
    executeCommandMenu()
```

Listing.18. Method call chain associated with an exception path found in AJHotDraw.

We can observe that most of the method calls presented in Listing 18 represent method calls inside Java libraries. Consequently, most of the AJHotdraw exception analysis processing time was dedicated to calculate the *exception path* inside such libraries (see Figure 23).

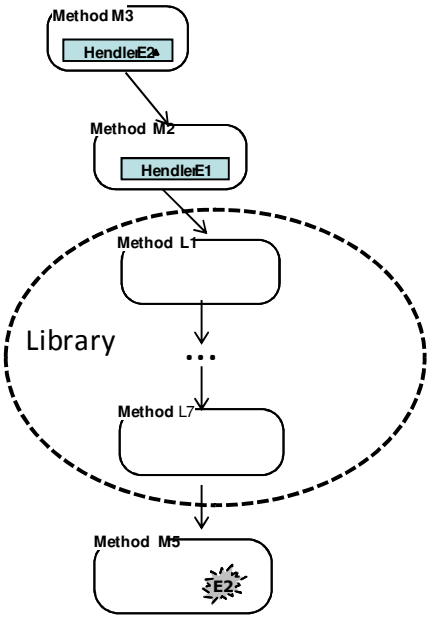

Figure 23. Library methods in an *exception path*.

Although the *exception path* may include both *application* and *library* methods, there is not much interest in the calling relationships among library methods for tools dedicated to software understanding, such as SAFE. The static analysis could run more efficiently if a summarized exception path could substitute the *complete* exception path presented above. We could reduce the size of such exception paths in two ways: (i) filtering library calls when calculating the exception paths or (ii) filtering such paths when reporting them.

The first strategy could lead to imprecise exception paths in the presence of *library callbacks* (Zhang and Ryder, 2007) (i.e. when the library calls back the application code) – see Figure 23. Our approach adopts the second strategy, as illustrated in Listing 19 below. In this listing some method calls were filtered after the exception analysis and the method calls to libraries were replaced by generic references to *library packages*. The summarized *exception paths* facilitate the manual inspection of the exception handling code and the exception handling contract checking (see Section 5.3.4).

```
1. org.jhotdraw.contrib.WindowMenu: void buildChildMenus()
2. org.jhotdraw.contrib.WindowMenu: void access$1()
3. org.jhotdraw.contrib.WindowMenu$3: void menuSelected()
4. library method calls: java.swing,java.awt
```

```
5.  org.jhotdraw.contrib.html.HTMLTextAreaFigure: float renderText()
6.  org.jhotdraw.contrib.html.HTMLTextAreaFigure: void generateImage()
7.  org.jhotdraw.contrib.html.HTMLTextAreaFigure: float drawText()
8.  org.jhotdraw.contrib.html.HTMLTextAreaFigure: void draw()
9.  org.jhotdraw.standard.CompositeFigure: void draw()
10. org.jhotdraw.standard.StandardDrawingView: void drawDrawing()
11. org.jhotdraw.standard.StandardDrawingView: void drawAll()
12. org.jhotdraw.standard.SimpleUpdateStrategy: void draw()
13. org.jhotdraw.standard.StandardDrawingView: void paintComponent()
14. javax.swing.JComponent: void paint()
15. org.jhotdraw.contrib.zoom.ZoomDrawingView: void paint()
16. library method calls: java.swing,java.awt
17. org.jhotdraw.contrib.MDIDesktopPane: … createContents()
18. org.jhotdraw.contrib.MDIDesktopPane: void addToDesktop()
19. org.jhotdraw.contrib.MDI_DrawApplication: void newWindow()
20. org.jhotdraw.samples.javadraw.JavaDrawApp$4: void execute()
21. org.jhotdraw.samples.javadraw.JavaDrawApp:                    void
    execute_aroundBody0()
22. org.jhotdraw.samples.javadraw.JavaDrawApp:                    void
    execute_aroundBody1$advice()
23. org.jhotdraw.samples.javadraw.JavaDrawApp:                    void
    executeCommandMenu()
```

Listing.19. Summarized exception path.

5.6. Summary

This chapter presented the supporting ideas and the implementation details of SAFE, a static analysis tool proposed in this work for discovering the exception flows in AspectJ programs. It is developed on top of the Soot framework (Vallée-Rai, 2000) for the static analysis of Java *bytecode*. The SAFE tool implements an exception-flow analysis algorithm that traverses a program representation - based on the program call graph and exceptional control flow information – to discover: (i) every exception that may escape from methods or advices (i.e., *exception interface*), and (ii) the *exception path* of each exception.

The *exception interfaces* and the *exception paths* are stored in XML files which are used, by specific modules of the SAFE tool, to mine information concerning the flow of exceptions in AspectJ programs. The *Exception Path Miner is* one of the SAFE tool components; this component parses the exception paths and classifies them according to the *Signaler-Handler* relationship. This

information is useful for discovering if the aspects are responsible for signaling the exceptions not adequately handled inside the system (e.g., uncaught exceptions).

The following chapter presents the SAFE tool dynamics in the context of a proposed verification approach for checking the reliability of the exception handling code of AspectJ programs. The approach uses the SAFE tool to (i) calculate the *exception paths* of AspectJ programs, (ii) automatically discover exception handling error-prone scenarios (e.g., *uncaught exceptions*, exception *subsumption*), and (iii) guide the manual inspection of exception handling code.