Improving the precision and correctness of exception analysis in Soot

Sable Technical Report No. 2003-3

John Jorgensen

September 15, 2003
Revised February 10, 2004

Contents

1 Introduction

Several aspects of Java's exception mechanism complicate the static analysis of Java programs.

First, the Java Virtual Machine signals violations of its semantics via exceptions, so code which would constitute a basic block using a traditional machine model turns out to contain many implicit branches when exceptional control flow is taken into account.

Second, the identity of the next statement to execute after an exception is thrown depends not only on the instruction being executed, but also on the type of exception being thrown and the set of `catch` clauses active when the exception occurs. But the exception's type and the set of active handlers cannot, in general, be determined statically, complicating the analysis of control flow and multiplying the number of paths along which dataflow values must be propagated.

Third, Java specifies precise exception semantics which introduce extra dependences between instructions that have the potential to throw exceptions, restricting the scope for optimisations to reorder instructions.

Finally, when the Java bytecode verifier performs dataflow analyses to ensure that a class file contains valid code, it assumes that every instruction protected by an exception handler may transfer control to that handler, even if the instruction cannot throw the exception that the handler catches. As a result, some otherwise legitimate optimisations produce unverifiable code.

Soot, a framework for representing, analysing and transforming Java class files[12], makes only limited provision for Java exceptions. As of release 2.1.0, Soot contains no general mechanism for mapping each instruction to the exceptions the instruction may throw. When building control flow graphs, Soot includes an edge from each instruction within a `try` block to each associated `catch` clause, even if none of the exceptions that the instruction may throw match the type of the `catch` parameter.[1] Conversely, in the absence of `try` blocks, exceptional control flow paths which abort a method are ignored completely.[2] Soot has no mechanism for representing the dependences required for precise exceptions, though individual optimisations (for example. the `LoadStoreOptimizer`, the `DeadAssignmentEliminator`, and partial redundancy elimination) make ad hoc provision for preserving the semantics of exceptions. Soot does contain analyses to detect pointer and array references which run no risk of `NullPointerException`s or `Array-IndexOutOfBoundsException`s, respectively, but the results are recorded in annotations to be read by a VM, rather than used within Soot. For example, the `DeadAssignmentEliminator` and partial redundancy elimination refuse to move array references even when their subscripts can be proved to be in-bounds.

This report has two loosely connected parts. Section 2 reviews the Java exception mechanism and discusses the impact of exceptions on compiler analyses in general, with little specific reference to Soot. Section 3 describes an attempt to refine Soot's treatment of intraprocedural exceptional control flow while making minimal changes to existing analyses and transformations, simply by removing from control flow graphs the edges that lead to exception handlers from statements which cannot throw any exception caught by the handler. Essentially, Section 2 ranges widely, raising a number of questions that compiler writers need to ask about exceptions, but providing few answers. Section 3 describes the implementation of one answer to a single narrow question: how to prune unrealizable exceptional control flow from Soot's control flow graphs.

[1]In other words, Soot is just as conservative as the verifier.

[2]To be fair, this does not invalidate any of the intraprocedural analyses performed.

2 Exceptions and static analysis

This section describes some of the difficulties that Java's exception mechanism presents to static compiler analyses such as those implemented in Soot. It begins by reviewing the standard description of explicit exceptions presented to Java programmers, then describes how the implementation of exceptions in bytecode is less constrained than the facilities used by the Java language. It goes on to discuss the difficulties that exceptions present to compiler analyses under two broad headings: the large number of potential exceptional control flow paths, and the constraints imposed by Java's notion of precise exceptions. It ends by describing the interaction of the exception mechanism with the Java bytecode verifier.

2.1 Exceptions as seen by Java programmers

2.1.1 Explicit exceptions

Java textbooks present exceptions as an error reporting facility which ensures that exceptional conditions will not be ignored inadvertently, without requiring programmers to read special status flags or check for distinguished function results. Instead, code which detects an exceptional condition "throws" an exception object whose type is some subclass of `java.lang.Throwable` (or `java.lang.Throwable` itself). For example:

```
if (! (huey.before(dewey) && dewey.before(louie)))
  throw new DucksNotInARowException();
```
(1)

If users of code which might throw an exception are in a position to do something about the exceptional condition, they enclose the use within a `try` block accompanied by a `catch` clause whose parameter type matches the class of the thrown exception, or one of its superclasses. The `catch` clause contains code to handle the exception. A very simple example would be:

```
try {
    if (! (huey.before(dewey) && dewey.before(louie)))
        throw new DucksNotInARowException();
} catch (DucksNotInARowException e) {
    nephews.sort();
}
```
(2)

but such examples—where a single `throw` statement and matching `catch` clause both occur in a single method—are unlikely to occur in real code, since it would be simpler just to replace the `throw` statement with the text within the `catch` clause. For a more realistic example of an explicit `throw` whose exception is caught in the same method, imagine that there were a series of activities where the nephews might be discovered to be out of order—so that "`throw DucksNotInARowException`" appeared many times—and that the body of the `catch` clause contained the code to sort the nephews, instead of a function call—so that the recovery code would be too bulky to reproduce at every point where the error might occur.

A more typical example is probably

```
try {
    takeNephewsOnOuting()
} catch (DucksNotInARowException e) {
```
(3)

```
        nephews.sort();
        takeNephewsOnOuting();
    }
```

Here the test which throws `DucksNotInARowException` occurs somewhere in the method `takeNephews-OnOuting()`, where it is not located within any `try` block associated with a `catch` whose parameter matches `DucksNotInARowException`. When a `throw` statement is not statically enclosed by a `try` block that catches its exception, the runtime system starts unwinding the call stack of the active thread, propagating the exception to the callers of the method which threw it.

At each activation record, the runtime system checks if the call just unwound occurred within a `try` block associated with a `catch` parameter that can be assigned the thrown exception object. If the system finds such a `try`, control passes to the first statement in the matching `catch` clause, with the thrown object bound to the clause's parameter. So in our example, should `takeNephewsOnOuting()` signal its inability to deal with unordered ducklings by throwing `DucksNotInARowException`, the calling code will sort the nephews and try again.

If the runtime system unwinds the thread's entire call stack without finding a enclosing `try` with a `catch` clause that handles the exception, it terminates the thread. Computations in methods unwound from the stack are essentially aborted,[3] except that the runtime system does execute any `finally` clauses associated with aborted `try` statements, and it releases the locks associated with any aborted synchronised blocks or methods.

An explicit `throw`, then, is a control flow transfer where the destination depends on the run-time type of the object thrown (much as the destination of a virtual method call depends on the run-time type of the call's receiver), as well as on the set of active method calls represented by the run-time stack, since they determine the set of active `catch` clauses.

An aside on terminology A `catch` clause might also be called an "exception handler" or simply a "handler". The statements within a `try` block are sometimes described as being "protected", "guarded", or "covered" by the associated handlers. When discussing the bytecode that implements a `try` statement, the instructions corresponding to the `try` block might be called the handler's area or zone of protection. Finally, we will follow the *Java Language Specification*'s practice of using the uncapitalised word "exception" to refer to any object that may be thrown (that is, any `Throwable`) and reserve the capitalised "`Exception`" for speaking specifically about `java.lang.Exception` and its subclasses.

2.1.2 Implicit exceptions

The Java language specifies that exceptions may be thrown implicitly by statements other than `throw`, to signal violations of Java semantics or implementation errors discovered in the runtime system. For example, the statement

```
    this.array[13/--i] = new Thing();                                    (4)
```

has the potential to throw at least twenty-five different `Throwable`s defined by the standard Java libraries.[4]

[3]The specification uses the terminology " complete abruptly", rather than "abort".

[4]If `this.array` contains a null pointer, the assignment will produce a `NullPointerException`.

In practice, few programs are in a position to do much about these exceptions at runtime, so application programmers can usually ignore the exceptional control flow paths that implicitly exist in their programs. Compiler analyses cannot ignore these paths if they are to produce correct code, but the details of implicit exceptions are more easily discussed in the context of bytecode instructions than that of Java source, so they are deferred until after the next section.

2.2 Exceptions in bytecode

While the Java language specification ensures that exception mechanisms are well-structured, the Java Virtual Machine specification does not enforce these restrictions at the bytecode level.

The specification of Java's `try` statements ensures such things as [8, p. 161][7, section 4.9.5]:

- While one `try` block may be nested completely within another, `try` blocks never overlap partially.
- When a `try` block is nested within another, all of the inner block's `catch` clauses are also protected by the outer `try`.
- When a `try` block is nested within another, any thrown exceptions are matched against the `catch` parameters associated with the inner `try` block (in the order those `catch` parameters appear) before being matched against any of the `catch` parameters of the outer `try` block.
- The `catch` clauses associated with a `try` statement are never themselves within the statement's `try` block; that is, handlers cannot protect themselves.
- Code within a `catch` clause will only execute as a result of catching an exception.
- A single `catch` clause protects only one `try` block, and may specify only one type for its parameter (although the parameter will match all subtypes of its declared type).
- a method's `throws` clause lists the types of all `Throwable`s it might throw (even indirectly, via a call to another method), except for those which are "unchecked exceptions", that is, subclasses of `RuntimeException` or of `Error`.

In the bytecode that implements a method, explicit `throw`s are represented by the `athrow` instruction while `try` blocks and `catch` clauses are represented by a table of exception handlers. Each entry in that table contains four values:

1. The index within the method's code array of the first instruction protected by the handler.
2. The index of the instruction following the last instruction protected by the handler.

If `i` is originally 1, the division by zero will produce an `ArithmeticException`.

If $13/$`i-1` < 0 or $13/$`i-1` $\geq$ `this.array.length`, the assignment will produce an `ArrayIndexOutOfBoundsException`.

If `this.array` contains an array of a subclass of `Thing`, rather than `Thing` itself or one of its superclasses, the assignment will produce an `ArrayStoreException`.

If the attempt to allocate a new `Thing` exhausts the VM's store, the allocation will produce an `OutOfMemoryError`.

If resolving the reference to `this.array` requires loading a new class, any of thirteen subclasses of `LinkageError` may result, should the VM fail to find an implementation of the new class which matches the expectations of the code already loaded. Any of six other subclasses of `Error` might be thrown by static initialisers of the loaded class.

In addition, the asynchronous exceptions `InternalError` and `ThreadDeath` might occur at any point, since they do not depend directly on the state of this thread's computation.

3. The index of the first instruction of the handler.

4. The type of exceptions being caught.

Java virtual machines do not impose restrictions on the values for the three indexes that would correspond to the structuring constraints on Java source. So a class file that was generated from a language other than Java, or by a bytecode optimiser, or even by a Java compiler that aggressively seeks opportunities to share code among exception handlers, may include an exception table that does not satisfy the corresponding constraints on Java source. An exception table can specify partially overlapping protected regions, handlers that protect multiple regions, handlers that accept multiple parameter types, or handlers that protect themselves. Indeed, since the release of Java 1.3, Sun's `javac` compiler implements `synchronized` and `finally` by generating code where a handler is in its own protected area, and since Java 1.4, such handlers protect two, disjoint ranges of code.

Furthermore, the order in which the runtime system checks the class of a thrown exception against the type of `catch` parameters is dictated solely by the order of the exception table entries, without regard to how the corresponding protected areas are nested. Finally, while the contents of a Java method's `throws` clause are represented in the class file compiled from the source, the virtual machine does not prohibit methods from throwing unchecked exceptions which are not included in their `throws` clauses.

The arbitrary locations for protected blocks and handlers is likely to affect only compiler analyses that must reconstruct high level source language structures. Sophisticated analyses of loop nests, for example, would be complicated by the possibility that a handler or protected block might not nest cleanly within a given loop. The loss at the bytecode level of the information provided by `throws` clauses, though, affects a wider variety of analyses, since it implies that purely intraprocedural analyses must assume that all method calls have the potential to throw any `Throwable` whatsoever, regardless of their `throws` clauses.

2.2.1 Implicit exceptions[5] and the instructions who throw them

This section lists the classes of `Throwable`s which may be thrown by the VM even in the absence of an explicit `athrow` instruction. Appendix A provides a table showing which `Throwable`s might be thrown by each JVM instruction.

First note that

$$\text{implicit exceptions} \subset \text{unchecked exceptions}$$

but

$$\text{unchecked exceptions} \not\subset \text{implicit exceptions},$$

that is, while all classes of potentially implicit `Throwable`s are subclasses of `Error` and `RuntimeException` (so they need not be included in a method's `throws` clause), the converse is not true: Not only may programmers define their own subclasses of `Error` and `RuntimeException` which may only be thrown explicitly, but the JDK libraries already define several such necessarily explicit, unchecked exceptions.

The potentially implicit exceptions (with subclasses indicated by indentation) are:

Subclasses of `RuntimeException`

- `java.lang.ArithmeticException`

[5]Strictly speaking we should call them "potentially implicit exceptions", since nothing stops programmers from throwing them explicitly if they care to.

  - java.lang.ArrayStoreException
  - java.lang.ClassCastException
  - java.lang.IllegalMonitorStateException
  - java.lang.IndexOutOfBoundsException
    * java.lang.ArrayIndexOutOfBoundsException[6]
  - java.lang.NegativeArraySizeException
  - java.lang.NullPointerException

Subclasses of `Error`

  - java.lang.LinkageError
    * java.lang.ClassCircularityError
    * java.lang.ClassFormatError
      · java.lang.UnsupportedClassVersionError
    * java.lang.ExceptionInInitializerError
    * java.lang.IncompatibleClassChangeError
      · java.lang.AbstractMethodError
      · java.lang.IllegalAccessError
      · java.lang.InstantiationError
      · java.lang.NoSuchFieldError
      · java.lang.NoSuchMethodError
    * java.lang.NoClassDefFoundError
    * java.lang.UnsatisfiedLinkError
    * java.lang.VerifyError
  - java.lang.ThreadDeath
  - java.lang.VirtualMachineError
    * java.lang.InternalError
    * java.lang.OutOfMemoryError
    * java.lang.StackOverflowError
    * java.lang.UnknownError

Asynchronous exceptions *The Java Language Specification* and *The Java Virtual Machine Specification* describe `ThreadDeath` and at least one subclass of `VirtualMachineError` as asynchronous exceptions. These represent abnormal events which may occur at any point during a program's execution, regardless of the particular instruction being executed by the current thread. `ThreadDeath` errors are raised in the "victim thread" by the the deprecated library methods `Thread.stop()` and `ThreadGroup.stop()`, while `VirtualMachineError`s indicate that the VM has exhausted some resource or discovered an error in its own implementation.

It is not completely clear which subclasses of `VirtualMachineError` may be delivered asynchronously. The first editions of the specifications stated unequivocally that only `InternalError` could be delivered

[6] `java.lang.StringIndexOutOfBoundsException` is also a subclass of `IndexOutOfBoundsException`, but it cannot be raised implicitly by the VM.

asynchronously [4, sections 11.1, 11.5.2], [6, sections 2.15.1, 2.15.4]. The second editions replace the precise "`InternalError`" with the more ambiguous "internal error" in the initial description of which exceptions may be asynchronous, though the exception inventory still implies that `InternalError` is the only asynchronous subclass of `VirtualMachineError`, without actually stating that the others are synchronous.[7]

Happily this ambiguity is not critical for Soot, since Soot performs bytecode-to-bytecode transformations, rather than actually executing the bytecode. For a VM implementor, asynchronous exceptions are distinguished by the fact that they might occur at arbitrary points in the execution cycle of an individual instruction. For Soot maintainers, asynchronous exceptions are distinguished by the fact that they may be thrown during the execution of any instruction, regardless of its opcode. For our purposes, `UnknownError` is just as asynchronous as `InternalError`, since the specifications do not limit which instructions may throw an `UnknownError`. *The Java Virtual Machine Specification* does describe when `OutOfMemoryError` and `StackOverflowError` may occur in terms of internal operations of the virtual machine (sections 3.5.2 to 3.5.6), but it does not explicitly tie those internal operations to specific opcodes, the way that it does for other exceptions.

So as far as Soot is concerned, then, the four subclasses of `VirtualMachineError` which may be thrown implicitly by the VM might all be thrown by any instruction.[8] In Appendix A, the label "*async*" stands for `ThreadDeath`, `InternalError`, `OutOfMemoryError`, and `StackOverflowError`. Note that we pedantically do not designate as asynchronous `VirtualMachineError` itself, nor any subclasses other than the four explicitly mentioned in the specification. Nothing prohibits application programmers from declaring their own subclasses of `VirtualMachineError`, or explicitly throwing instances of those subclasses, but such user-defined classes could not be thrown implicitly by the VM, barring a change to the VM specification.

Linkage errors A number of instructions have the potential to trigger the loading of new classes, and the loading and linking of new classes, in turn, may result in any of a number of exceptions. In Appendix A, these instructions include *linkage* in the the "Exceptions thrown" column.

The label "*linkage*" stands for `Error` and its subclasses. Most of the potential exceptions during loading are instances of `LinkageError` and its subclasses, but one step in loading a class is to run its static initialisers, that is, to invoke its `<clinit>` method. In effect, instructions marked *linkage* may be the site of an implicit method call. Should `<clinit>` throw an `Error`, that exception will be raised in the context of the instruction which caused the class to be loaded. If the initialisers raise an exception that is not an instance of `Error` or one of its subclasses, that exception will be replaced with an `ExceptionInInitializerError`.

The VM specification categorises `LinkageError`s according to when they may occur in the process of readying a class for use:

1. **Loading**: Loading a class refers to finding a binary representation of the class. The possible excep-

[7] As evidence that others have found this detail of the specifications unclear, see the URL `http://www.ergnosis.com/java-spec-report/java-language/jls-11.3.2.html`, which recognises the ambiguity, but concludes that only `ThreadDeath` and `InternalError` may be delivered asynchronously. Because the second editions of the specifications are less precise than the first, though, one wonders if the authors might have deliberately introduced ambiguity to increase the latitude provided to implementors.

[8] Even without knowledge of the internals of the VM executing the bytecodes, it is probably safe to assume that `StackOverflowError` can only be thrown by the same opcodes as may throw linkage errors, since these are the opcodes which have the potential to invoke a method, directly or indirectly. It is more difficult, though, to limit the instructions during which a virtual machine might try, and fail, to allocate more memory, throwing `OutOfMemoryError` as a result. Given that few `catch` parameters are declared as subtypes of `VirtualMachineError`, we lose little by taking the safe route of treating all four of the implicitly throwable `VirtualMachineError`s as asynchronous.

tions are:

`ClassFormatError:` the data loaded as a representation of the class is malformed.

`UnsupportedClassVersionError:` the data loaded was in an unsupported class file format.

`ClassCircularityError:` the class or interface to be loaded would be its own superclass or superinterface.

`NoClassDefFoundError:` the class loader could find no definition for the required class.

2. **Linking**: Linking refers to incorporating the binary representation of the class into the runtime state of the VM so that the class may be executed. Linking involves:

   (a) **Verification**: ensuring that the loaded representation is properly formed, with a valid symbol table, and that it obeys the semantic requirements of the language definition. Verification is discussed further in sections 2.6 and 3.4.

   `VerifyError:` indicates that the class failed verification.

   (b) **Preparation**:

   Allocation of storage for the class's static fields and for VM data structures implementing the class. Presumably this might throw an **`OutOfMemoryError`**, but that possibility is not mentioned explicitly in section 2.17 of the specification.

   (c) **Resolution**:

   Resolving a class consists of checking symbolic references from the class to other classes and interfaces, loading the classes and interfaces referred to, and checking that the references are correct. Resolution errors result from incompatible changes made to the definition of the referenced class after the compilation of the referring class. They are signalled by an instance of `IncompatibleClassChangeError` or one of its subclasses, with the possibilities depending on the type of reference being resolved:

   - Class resolution errors:

     `IllegalAccessError:` signals an attempt to access a class or class member whose declaration makes it inaccessible.

     `InstantiationError:` signals an attempt to instantiate an abstract class or interface.
   - Interface resolution errors:

     `IllegalAccessError`
   - Method resolution errors:

     `IllegalAccessError`

     `NoSuchMethodError`
   - Field resolution errors:

     `IllegalAccessError`

     `NoSuchFieldError`

   (d) **Initialisation**: Initialisation consists of initialising any static fields and executing any static initialisers. The possible exceptions include:

   `ExceptionInInitializerError:` one of the initialisers has thrown an exception that is not an instance of `Error` or one of `Error`'s subclasses.

   `OutOfMemoryError`

any other `Error`: which is raised by an initialiser.

In principle, the potential exists to rule out linkage errors if one can prove that all the classes involved must already be loaded at this program point. In practice, two features of the JVM specification make it all but impossible to realize this potential.

First, the specification allows implementations to perform resolution early, as classes are loaded, or late, as references are used, or some combination of the two:

> The only requirement regarding when resolution is performed is that any errors detected during resolution must be thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error.
>
> [7, Section 2.17.1]

This alone makes reasoning about the possibility of linkage errors very difficult without knowledge of the VM that will execute the bytecode (ruling out such optimisations for bytecode-to-bytecode transformers like Soot). An initial `getstatic`, performed when the VM considers itself too busy to anticipate future loads, might link only the bare minimum of classes required to execute the instruction, while a later `getstatic` on the same field might decide to pull in the definitions of classes that are referred to in the classes already loaded, but have yet to be referenced.

Second, the specification allows the bytecode verifier to delay some of its checks until the first time that the relevant instruction is executed, instead of performing them all immediately when the class is loaded [7, Section 4.9.1]. For example, if class `A` contains an instruction to read field `f` from class `B`, but field `f` is now private, at least some VMs[9] will not report a `IllegalAccessError` unless and until the `getfield` instruction accessing `B.f` is actually executed. So `LinkageError`s cannot be ruled out even when one is certain that the referenced classes have already been loaded.

`IllegalMonitorStateException` and unsynchronised methods Appendix A lists `Illegal-MonitorStateException` as a possibility for all return instructions. Note that this includes returns from unsynchronised methods, since, according to section 8.13 of the JVM specification, a VM has the option to enforce proper nesting of locks within each procedure invocation. For example, a VM could throw an `IllegalMonitorStateException` when returning from an unsynchronised method which contained a `monitorenter` but no corresponding `monitorexit`. It is conceivable—given a sufficiently perverse reading of the specification—that a VM might even throw an `IllegalMonitorStateException` upon returning from an unsynchronised method which itself contained no monitor statements, but which called another method that was guilty of unbalanced locking.

2.3 Exceptional control flow

The existence of exceptions affects compiler analyses by increasing both the number of paths which control may take through a program and the difficulty of identifying those paths at compile-time. Thus exceptions make analysing control flow more difficult, and the resulting control flow graphs (CFGs) more complex. More complex CFGs, in turn, complicate the dataflow analyses which push facts along CFG edges and the dependence analyses which characterise relationships between CFG nodes [10, p. 4].

There are two aspects to the more complicated control flow:

[9]That is to say, all the VMs that we tested.

- The existence of implicitly thrown exceptions means that many instructions that one normally thinks of as purely sequential—always falling through to the next instruction—may, in fact, branch to an exception handler or terminate the executing thread completely.

  In terms of the control flow graph: the node corresponding to such an instruction becomes the source of multiple edges rather than a single edge.

- When an exception is thrown, the identity of the next instruction to be executed depends on the type of the exception object and—unless the exception is caught within the method that threw it—on the current stack of executing methods, which determine the set of active handlers. Both factors are, in general, unknown until runtime, though static analyses may sometimes be able to narrow the possibilities.

  In principle, this factor increases the number of edges in the control flow graph by providing multiple destinations for edges corresponding to a particular exception thrown by a particular instruction. In practice, the difficulty of determining the set of possible handlers for exceptions which escape a method may result in all of them being approximated by a single destination node.

Corresponding to these two aspects are two questions that an exceptional control flow analysis must answer:

- what exceptions might each instruction throw?
- what handlers might be active when they are thrown?

The answers to these questions must be represented in some sort of data structure, which prompts a third question:

- how do you represent the transfer of control from an excepting statement *A* to the handler beginning at statement *B*, given that not all the work represented by *A* is necessarily performed before the transfer to *B*?

The next three sections address these questions in turn.

2.3.1 What exceptions might an instruction throw?

A first approximation of the set of exceptions which an instruction may throw implicitly can be based entirely on the opcode of the instruction, using *The Java Virtual Machine Specification*'s descriptions of which exceptions each virtual machine instruction may throw (summarised in Appendix A).

Static analyses may be able to further reduce the set of possible exceptions by proving that particular instances of an instruction cannot throw particular exceptions. For example, the Java source

```
    int[] array = new int[len];
    array[0] = len;
```
(5)

might be implemented by the bytecode:

```
iload_0
newarray int
astore_1
```
(6)

```
4   aload_1
5   iconst_0
6   iload_0
7   iastore
```

In general, an `iastore` instruction might throw a `NullPointerException`, to signal that the array reference it is trying to index has the value `null`. But a static analysis can determine that the particular `iastore` on line 7 cannot throw a `NullPointerException`; line 7 can only be executed if the `newarray` instruction on line 2 succeeded in putting a valid array reference onto the stack.

For an `athrow` instruction, determining the set of possible exceptions also requires determining the possible classes of the object that it throws explicitly. Note the distinction between the type of the `throw` instruction's argument expression—which is known at compile time—and the classes that instantiate that expression at run-time.[10] The distinction matters because the choice of handler to execute depends on the class of each exception instance at run-time, not the type of the expression at compile-time. This is clearly illustrated by a contrived example[11]:

```
void pitchAndCatch(Exception t) {
  try {
    throw t;
  } catch (NullPointerException e) {
    handleNullPointer();
  } catch (RuntimeException e) {
    handleRuntime();
  } catch (Exception e) {
    handleException();
  }
}
```
(7)

If an analysis can somehow determine that all values passed as `t` at run-time will be instances of `NullPointerException`, then it can produce the CFG illustrated by Figure 1 (a). If it can prove that none of the possible values are instances of `RuntimeException`, then it can produce the CFG in Figure 1 (b). If, though, the analysis can establish nothing more specific than the compile-time type for `t`, it cannot produce a CFG more precise than that in Figure 1 (c).

The effect of an exception's class on the choice of handler is analogous to the effect of the receiving object's class on the choice of method implementation when performing a virtual call. Using type inference to narrow the set of potential virtual method implementations is a heavily-researched topic, and the same techniques could serve for estimating the possible run-time instantiations of thrown exceptions. Common Java idioms, though, reduce the need for sophisticated analyses of exception types. Empirical studies [10, p. 12] show that the vast majority of `throw` arguments in the wild are `new` expressions (as in our example "`throw new DucksNotInARowException()`"). The exact class of objects instantiating such expressions is directly available.

As another illustration of the importance of distinguishing run-time classes from compile-time types, contrast the two methods in the following example:

[10] I am adopting the nomenclature of the Java specifications[7, section 2.5.2]: rather than distinguishing compile-time types from run-time types, the specifications say that "A variable or expression has a type; an object or array has no type, but belongs to a class." (8)

[11] Recall that `NullPointerException` is a subclass of `RuntimeException`, which is a subclass of `Exception`.

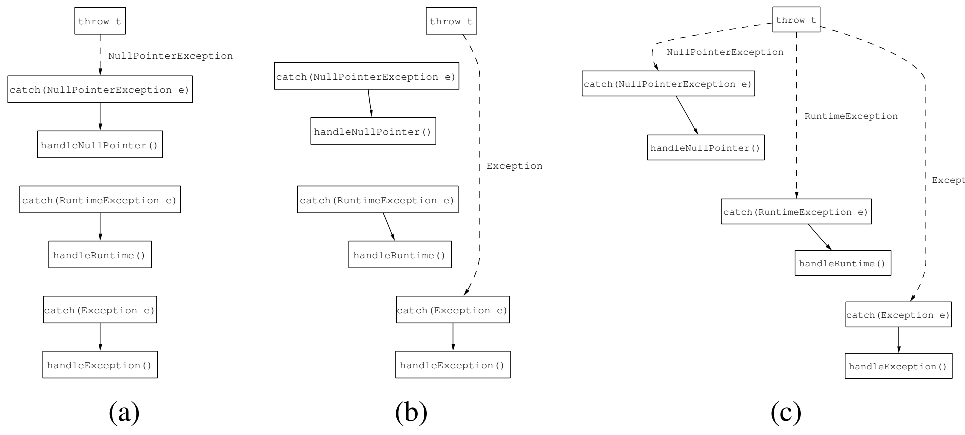

Figure 1: CFGs corresponding to different approximations for the class of `t`

```
import java.awt.print.PrinterException;
import java.awt.print.PrinterIOException; // Subclass of PrinterException

void generateException() {
  try {
    throw new PrinterException();
  } catch (PrinterIOException e) {
    kickPrinter();
  }
}

void filterException(PrinterException t) {
  try {
    throw t;
  } catch (PrinterIOException e) {
    kickPrinter();
  }
}
```

In both methods, the compile-time type of the object thrown is `java.awt.print.PrinterException`. In `generateException()`, though, we know that the instantiations of the exception will always be an instance of the `PrinterException` class itself, and not of a subclass. Without an interprocedural analysis of all potential callers of `filterException()`, on the other hand, we have to assume that `t` might be instantiated by any subclass of `PrinterException`, including `PrinterIOException`. So while we know that the `catch` clause in `generateException()` will never be executed, we have to assume that the one in `filterException()` might be.

As an aside, concocting this example inadvertently illustrated another lesson about exceptions: their ubiquity. The original incarnation of the example used `RuntimeException` and `NullPointerException` in place of `PrinterException` and `PrinterIOException`. The difficulty is that one cannot be sure the `RuntimeException` constructor might not itself throw a `NullPointerException`, at least not without analysing the `RuntimeException` class, its superclasses, and the classes of any of their fields. Even the revised example avoids this problem only so long as one sticks to code generated from Java source, where the Java compiler enforces the guarantee that checked exceptions do not escape methods unless the exception is specified in a `throws` clause.

2.3.2 What handlers might be active?

Intraprocedurally, finding active handlers is a straightforward matter of checking each entry of a method's exception table in order. The basic algorithm is:

```
for each statement, $s$,
    for each exception, $e$, that $s$ may throw,
        for each handler, $h$ that protects $s$, in exception table order,
            if $h$'s catch type matches $e$'s class or one of its superclasses
                Record that $s$ throws $e$ to $h$.
                Break to the next exception.
        Record that $e$ escapes the method.
```

Indeed, that is the algorithm implemented in Soot during the course of this project, with some minor refinements—and aborted attempts at refinements—discussed in section 3.

In principle, turning the intraprocedural algorithm into an interprocedural algorithm to find handlers for exceptions which escape a method simply requires inserting another nested loop, albeit one that is called recursively:

```
for each statement, $s$, in method $m$,
    for each exception, $e$, that $s$ may throw,
        handlers$\leftarrow$ findHandlers$(m, s, e)$
        Record that $s$ may throw $e$ to any of the elements in the set handlers.
```

where `findHandlers()` is:

```
findHandlers$(m, s, e)$:
    for each handler, $h$ in $m$ which protects $s$, in exception table order,
        if $h$'s catch parameter type matches $e$'s class or a superclass
            return $\{h\}$ as the set of handlers catching $e$ when thrown by $s$ in $m$.
    // Otherwise $e$ escapes $m$:
    result$\leftarrow \emptyset$
    for each call site, $s_0$, in every method, $m_0$, which may call $m$
        result$\leftarrow$result $\cup$ findHandlers$(m_0, s_0, e)$
    if result remains empty
        result$\leftarrow$ {escapesThread}
    return result
```

There are glaring difficulties with the naive interprocedural algorithm.

First, the blithe words "for each call site, s_0, in every method, m_0, which may call m" gloss over an entire domain of active research: the approximation of the call graph of object-oriented programs. Sophisticated points-to analyses are required to estimate the possible classes of invocation receivers without being bogged down by the sheer volume of data. Moreover, even those sophisticated analyses require knowledge of the whole program, which is not available at compile-time for a language like Java, where classes are loaded—and even created—on-the-fly.

Second, the naive algorithm is woefully inefficient. As stated here, the recursive calls would compute the same information over and over again, so the algorithm at least needs to be augmented with some form of memoization or dynamic programming. The algorithm that Sinha and Harrold [10] present for constructing interprocedural control flow graphs which include explicit exceptional control flow is similar in spirit to this brute-force algorithm, but it produces a global CFG in one go by linking together the `throw` and `catch` statements of intraprocedural CFGs, avoiding the extravagance of the recursive calls used here.

All analyses implemented in the course of this project are strictly intraprocedural, so this report says no more about the problem of finding handlers for exceptions which escape a method.

2.4 What do control flow edges mean? Where should they occur?

This subsection might be entitled, "How what we implemented was a flawed idea to begin with." The modifications to Soot described in section 3 aim to provide more precise information about exceptional control flow by removing edges from the existing control flow graph structures, without modifying the existing analyses to distinguish exceptional control flow from regular control flow. This is a problematic exercise, because exceptional control flow differs from regular control flow.

In the absence of exceptions, the meaning of nodes and edges in a method's control flow graph is clearly established:

- there is a node for each instruction in the intermediate representation of the method;
- a directed edge e from node m to node n means that after the instruction corresponding to m is executed, the instruction corresponding to n may be the next to execute (*will* be the next to execute if there is only one edge out of m);
- if more than one edge leaves a node, the node must be some sort of conditional branch (an `if` or `switch`), so the value of the instruction's condition will determine which edge is taken.
- if edge e is taken, all the work of m's instruction is completed before n's instruction begins execution.

The last two points cease to be true in the presence of exceptions.

When some of the edges leaving a node are due to exceptional control flow, it is more difficult to describe the circumstances under which a particular edge is taken or (equivalently) to characterise what distinguishes the computational states at each of the destination nodes. This is because a different analysis is required to determine the circumstances under which each potentially caught exception occurs. To make this concrete, consider the following small example:

(9)

```
static int method(int i, int j) {
    Comparable[] array;
    switch (i) {
    case 0:
        array = null; break;
    case 1:
```

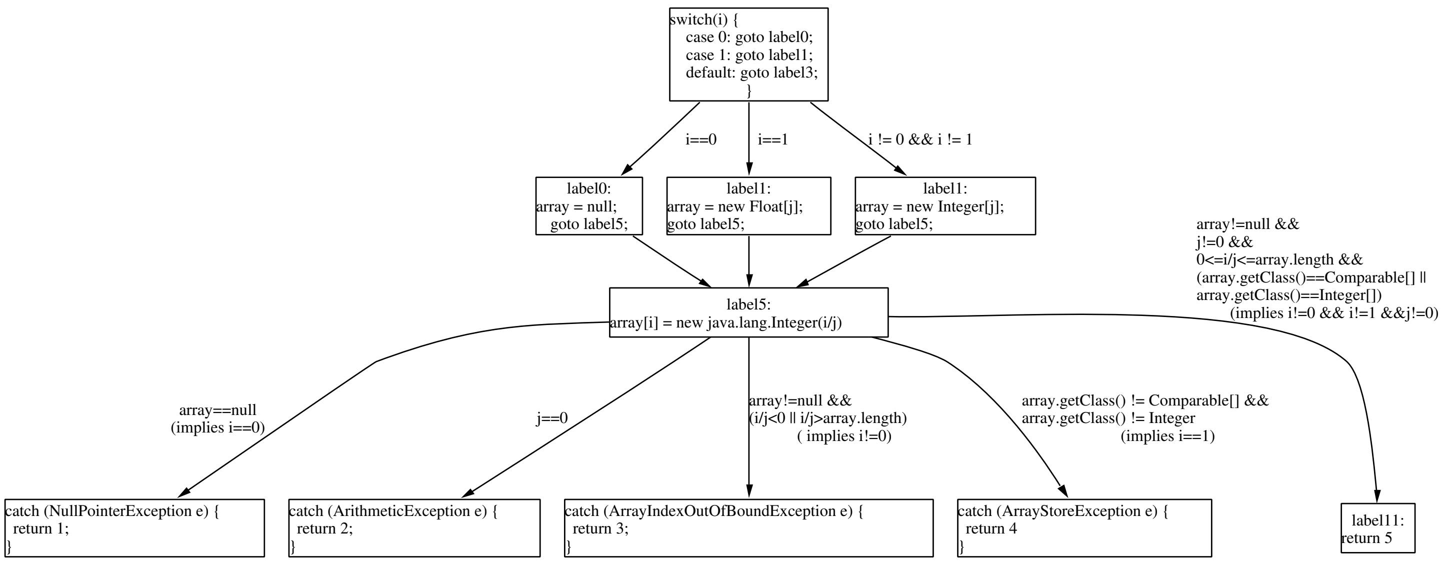

Figure 2: CFG for `method(int i, int j)`

```
            array = new Float[j]; break;
        default:
            array = new Integer[j]; break;
        }
        try {
            array[i] = new Integer(i/j);
        } catch (NullPointerException e) {
            return 1;
        } catch (ArithmeticException e) {
            return 2;
        } catch (ArrayIndexOutOfBoundsException e) {
            return 3;
        } catch (ArrayStoreException e) {
            return 4;
        }
        return 5;
    }
```

Figure 2 is a control flow graph for the example. The edges leaving the `switch` statement and the `try` block are labelled with the conditions which are true when that edge is taken. Note that the conditions on edges leaving the `switch` statement all represent different values for a single expression, while those on edges leaving the `try` block have little in common and can become quite complicated.

When an exception occurs, some or all of the excepting instruction's work will not have been completed. As a result, if we want client analyses using the CFG to be able to treat exceptional control flow edges as indistinguishable from regular control flow edges, it is not clear if the edges representing exceptional control flow should connect the excepting statement itself to the handler, the predecessors of the excepting statement to the handler, or both.

In the following code, for example, if line 3 throws a `NullPointerException` because `o` is `null`, it never assigns a new value to `l0`.

```
1      int l0 = 0;                                              (10)
2      try {
3        l0 = o.f * o.f + 1;   // rhs necessarily positive.
4        return l0;
5      } catch (NullPointerException e) {
6        System.err.println("NullPointerException");
7        if (l0 == 0) return 0; else return 1;
8      }
```

Thus, to represent control flow when the `NullPointerException` is thrown requires an edge to the handler from line 1, the predecessor of the throwing statement, rather than from the throwing statement itself. Representing exceptional control flow with a regular edge from line 3 to line 6 could mislead a sophisticated constant propagator into deducing that `l0` could not have the value 0 at line 7 so that the `if` statement could be replaced with `return 1`, when just the opposite is true: `l0` will *always* have the value 0 at line 7, so the test could be replaced by `return 0`.

It would be ideal if our representation of programs could guarantee that when a statement throws an exception, none of its work has been performed. Then only the predecessors of excepting statements would have CFG edges to the handlers that catch the exceptions. But it is impossible to design an intermediate representation for a non-functional language so that it guarantees that when an exception is thrown, no work has been performed by the excepting statement.[12] When a method call occurs in the scope of an exception handler and the called method might throw the exception that the handler catches, it is possible that any side effects of the called method may occur before it throws the exception.

To see how this observation bears on the construction of CFGs, let us change the field reference in our last example to a method call:

```
1      int l0 = 0;                                              (11)
2      try {
3        l0 = o.f() ;
4        return l0;
5      } catch (NullPointerException e) {
6        System.err.println("NullPointerException");
7        if (l0 == 0) return 0; else return 1;
8      }
```

A `NullPointerException` might occur because `o` itself is null, in which case the call to `f()` never occurs, or because some pointer referenced within `f()` is null. Since locals are accessible only from the method that declares them, in either case `l0` still necessarily has the value `0` at line 6. But in the absence of any information about the possible side-effects of `o.f()`, our CFG needs to indicate that some part of the method call may occur before control is transferred to the handler. It is not impossible, for example, that `o.f()` assigns

[12]At least it is impossible so long as we want to build control flow graphs for methods in isolation. An "all or nothing" representation would be conceivable, though probably impractical, if one built a CFG for the entire program. Such an interprocedural CFG would contain edges from each call, `o.f()` or `f()`, to the bodies of all the possible target methods implementing `f()`. Then if the call were within the scope of some handler catching an exception that `f()` might throw, there would be separate edges to the handler from each statement in the bodies that might throw the exception, instead of a catch-all edge from the call site.

a new value to `System.err` before referencing a null pointer, changing the effect of the `println()` in the `catch` clause. So a CFG for this example requires edges to the handler both from line 1, the predecessor of the call, and from the call itself, line 3.

In Java, the situation is aggravated by the fact that references to static fields may implicitly require method calls, so static gets and puts may have side effects. It is worth illustrating this with a contrived program which produces incorrect results if the wrong assumptions are made during optimisation:

(12)

```
 1  public class SideEffector {
 2      public static void main(String[] arg) {
 3          saveProduct(Integer.parseInt(arg[0]), Integer.parseInt(arg[1]));
 4      }
 5      private static void saveProduct(int i, int j) {
 6          int result = i * j;
 7          PrintStream savedStream = StreamHolder.out;
 8          try {
 9              ArrayHolder.array[i] = result;
10          } catch (ArrayIndexOutOfBoundsException e) {
11              PrintStream usedStream = savedStream;
12              usedStream.println(result);
13          }
14      }
15  }
16
17  class StreamHolder {
18      static PrintStream out = System.out;
19  }
20
21  class ArrayHolder {
22      static {
23          StreamHolder.out = System.err;
24      }
25      static int[] array = new int[10];
26  }
```

If `i` is less than `0` or greater than `10` so that statement 9 throws an `ArrayIndexOutOfBoundsException`, no value is assigned to the array element. So it would seem that the edge to the `catch` block should be rooted at the excepting statement's predecessor, on line 7.

But the existence of static initialisers means that resolving the left hand side of line 9 has a side effect, even when the assignment does not occur. Line 7's access of `StreamHolder.out` results in `StreamHolder`'s static initialiser being executed, assigning `System.out` to `StreamHolder.out`. Then line 9's access of `ArrayHolder.array` forces execution of `ArrayHolder`'s static initialiser, which has the side effect of changing `StreamHolder.out`'s value to `System.err`.

If we were to include CFG edges only from the excepting statement's predecessors to the handler, there would be a control flow edge from line 7 to 11, but none from 9 to 11. As a result an optimiser might try to save an apparently pointless copy by turning `saveProduct` into

(13)

```
 1      private static void saveProduct(int i, int j) {
    int result = i * j;
    try {
        ArrayHolder.array[i] = result;
    } catch (ArrayIndexOutOfBoundsException e) {
        PrintStream usedStream = StreamHolder.out;
        usedStream.println(result);
    }
}
```

with the result that the product would incorrectly be written to `System.err` instead of `System.out`.[13]

Adding control flow edges from the predecessors of excepting statements introduces a number of complications. First, it increases the number of edges in the graph along which flow facts must be propagated, both because the excepting instructions may have multiple predecessors, and because instructions with side effects must have an edge from the excepting instruction to the handler as well as from every predecessor. Second, it is not clear what to do when the first instruction is in the scope of a handler which catches some exception it may throw, and the first instruction has no side effects, as illustrated by:

```
 1   int m(int[] a, int [] b, int j) {                              (14)
 2       try {
 3           return a[j];
 4       } catch (ArrayIndexOutOfBoundsException e0) {
 5           try {
 6               return b[j];
 7           } catch (ArrayIndexOutOfBoundsException e1) {
 8               return 0;
 9           }
10       }
11   }
```

The initial instruction, line 3, has neither predecessors nor side effects, so what should be the origin of the edges to the handlers on lines 6 and 8? Are lines 3, 6 and 8 all potentially the "first" instruction in the method? Careful design of the intermediate representation can ameliorate this problem. Soot's Jimple, in particular, has identity statements to specify the correspondence between local variables and parameters, and their existence means that the excepting statements in this particular example do have predecessors after all (fig 3 is the pruned CFG of the Jimple for example 14; the symbols are explained in section 3.2). There are pathological cases, though, that require designating multiple "initial" instructions.

Another difficulty with adding edges from the predecessors of an excepting instruction to a handler is that now the predecessors have outgoing edges which are attributable not to the identity of the originating instruction, but to that of the instruction that it happens to precede. This is fine so long as the relationship between the instructions remains fixed, but compilers build CFGs precisely in order to transform code to improve it somehow, and the transformations frequently involve reordering instructions. To illustrate the difficulty, imagine that we are optimising a piece of code represented by the CFG in figure 4(a), where the labels `b0` and `b1` stand for two copies of code which have an identical effect. If there were no dependencies

[13]Soot's `Aggregator` performs similar optimisations, but it does not rely solely on the CFG for information about exceptional control flow. The `Aggregator` will not aggregate copies when the set of catch clauses in scope is different at the points of use and definition, so it would not mistakenly aggregate `savedStream` and `usedStream` in the example.

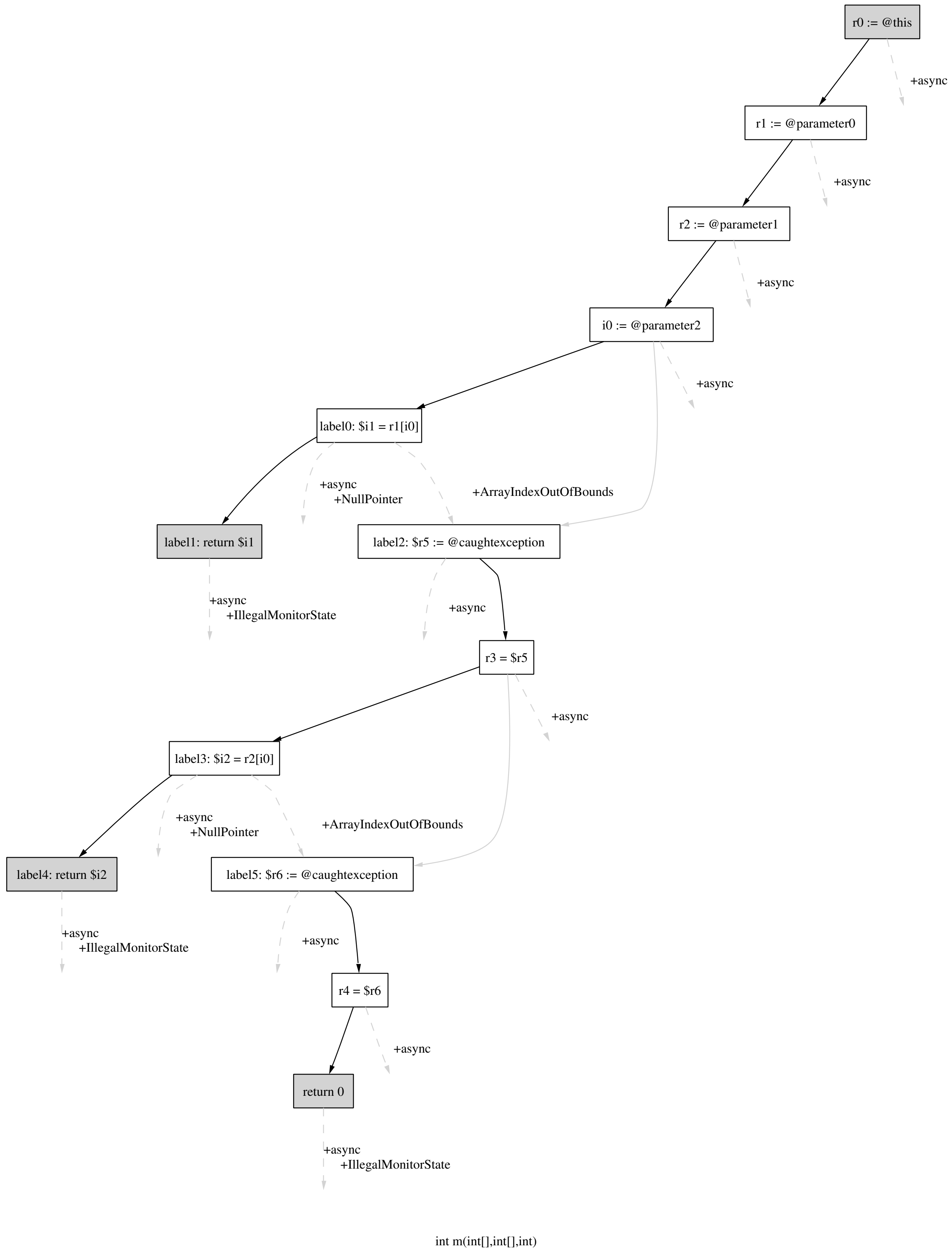

Figure 3: `CompleteUnitGraph` for example 14

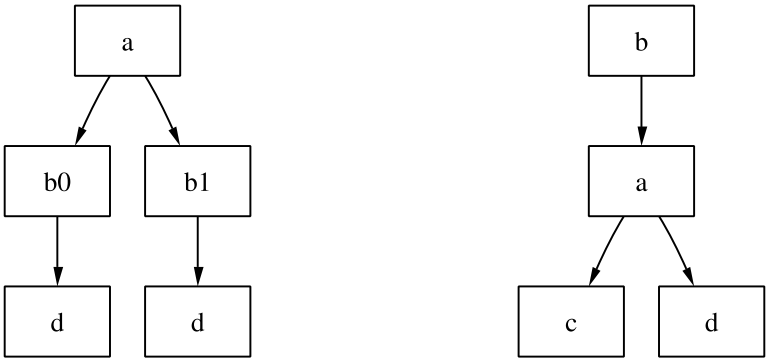

Figure 4: Is the transformation from (a) to (b) legitimate?

between the nodes, an optimiser might think it could replace `b0` and `b1` with a single copy, `b`, moved before node `a`, as in figure 4(b).

The transformation is probably legitimate if `a` is an `if` statement and `b0` and `b1` occur at the beginning of its two alternatives. It is most certainly not legitimate if `b0` is a potentially excepting instruction which is covered by a handler that begins with `b1`, that is, if `b0` is `a`'s regular successor, and `b1` is `a`'s successor in the case where `b0` throws an exception. In that case, control can flow from `a` to `b1` and thence to `d` only so long as `b0` follows `a`; move `b0` before `a`, and there is no longer any path from `a` to `d`.

Admittedly the interpretation of figure 4 as exceptional control flow is not very realistic: the reason the nodes are labelled with letters rather than example instructions is that we cannot think of any reasonable cases where the first instruction in a handler would be the same as the first instruction in the protected area. The example serves only to illustrate how, even if we try to make exceptional control flow edges look like regular control flow edges by adding them to the predecessors of excepting instructions, the exceptional edges remain different from regular edges. The resulting CFG does accurately reflect possible control flow in the input method, and thus can be used to perform dataflow analyses, but it cannot be used blindly as the basis for altering control flow.

2.4.1 Aside on intermediate representations

Earlier we noted that it is impossible, given the existence of method calls with potential side effects, to create an intermediate representation in which an instruction either throws an exception or does nothing. It is worth observing, though, that we could come closer to an "all or nothing" IR, that is one in which when an exception occurs, either all the work of the statement has been performed, or none of it has. The first step toward such an IR would be to separate function calls into two steps: execution of the called method and assignment of its result. Then the previous example illustrating the difficulties of choosing where to add control flow edges would become:

```
int l0 = 0;
try {
  @returnVal = o.f() ;
  l0 = @returnVal ;
  return l0;
```
(15)

```
6        } catch (NullPointerException e) {
7          System.err.println("NullPointerException");
8          if (l0 == 0) return 0; else return 1;
9        }
```

with `@returnval` a magic place-holder akin to the `@caughtexception` which is already found in Jimple. Then there could be an edge to line 7 from line 3 but not line 4, and constant propagation could still show that `l0` has the value `0` in the `catch` block.

In our example, though, we still have not succeeded in producing an "all or nothing" representation, since it does not distinguish `NullPointerException`s caused by `o` itself being `null`—which are thrown before executing `f()` and thus have no side-effects—from `NullPointerException`s that occur during the execution of `f()`. We could imagine an IR that followed the model of the Jikes RVM and represented exception tests explicitly in the code, making the example

```
 1      int l0 = 6;
 2      try {
 3        int l1 = l0 * 4;
 4        nullcheck o;
 5        @returnval = o.f();
 6        l0 = @returnVal;
 7        return l0 + l1;
 8      } catch (NullPointerException e) {
 9        System.err.println(l0 + 6);
10        return 0;
11      }
```

In a VM, where these exception check instructions correspond to work that the VM does need to perform, there are other advantages to representing implicit exception checks with explicit instructions. Doing so exposes opportunities to optimise the checks themselves (for example, moving a loop invariant `nullcheck` outside of the loop) and simplifies bookkeeping when a check can be proven unnecessary: simply remove the explicit check command.

Representing exception checks explicitly in the intermediate representation of a bytecode-to-bytecode transformer like Soot, though, has no advantages beyond distinguishing whether an exception occurs before or after a method call. When the target language is Java bytecode, the intermediate representation must maintain an association between the implicit check instruction and some other instruction which, in the emitted bytecode, actually performs the implicit check. Given that the exception check would always be tied to another instruction anyway, the bookkeeping required to make the separation is probably not worthwhile, especially considering that doing so cannot increase the precision of intraprocedural dataflow analyses which pessimistically assume that any method call has the potential to throw any exception anyway (in our example, there would have to be edges to the `NullPointerException` handler from both the `nullcheck` on line 4 and the call on line 5).

2.5 The strictures of precise exceptions

The Java Language Specification and *The Java Virtual Machine Specification* both require that Java exceptions be "precise":

> All exceptions in the Java programming language are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimised code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program.
>
> [7, Section 2.16.2]

Precise exceptions reduce the scope for reordering instructions which may throw exceptions. Soot makes no explicit provision for these ordering constraints, something which this project has not remedied. A discussion of the implications of precise exceptions is included here, nevertheless, to ensure that future Soot maintainers are aware of this loose end. This discussion follows [1] very closely; readers are referred there for more detail.

Here is an example to illustrate the constraints imposed by the combination of precise exceptions and Java's specification of the order of expression evaluation:

```
try {     // x, a[], and c have type int
  q = riskyBusiness(x=a[4*3], b[0]=new Object(), c=4*3);
} catch(ArrayIndexOutOfBoundsException e) {
  ... // b, and c definitely unchanged, x could have a new value
      // if the exception was raised by the assignment to b[0].
} catch(ArrayStoreException e) {
  ... // x == a[4*3], but b[0], c unchanged.
} catch(Exception e) {
  ... // x, b[0], c might all have new values.
}
```

A compiler could save one multiplication by introducing a temporary variable:

```
try {
  temp = 4 * 3;
  q = riskyBusiness(x=a[temp], b[0]=new Object(), c=temp);
} catch(ArrayIndexOutOfBoundsException e) {
  ... // b and c unchanged, x possibly changed
} catch(ArrayStoreException e) {
  .... // x == a[4*3], but b[0], c unchanged.
} catch(Throwable e) {
  ... // x, b[0], c might all have new values.
}
```

But if the variable `c` is used within the handlers for `ArrayIndexOutOfBoundsException` or `Array-StoreException`, a compiler could not try to save a register by assigning `4 * 3` to `c` instead of a temporary variable, like this:

```
try {
  c = 4 * 3;
```

```
    q = riskyBusiness(x=a[c], b[0]=new Object(), c);
  } catch(ArrayIndexOutOfBoundsException e) {
    ... // Oops! c has a new value here.
  } catch(ArrayStoreException e) {
    ... // And here!
  } catch(Throwable e) {
    ...
  }
```

since that would mean that the optimised code would see a different value for `c` than the unoptimised code.

Chambers et. al. [1] distill the constraints of precise exceptions into three rules:

1. An exception may be thrown by optimised code only if it would be thrown by the original code.
2. Exceptions must be thrown in the same order in optimised and unoptimised code.
3. Observable program state when an exception is thrown must be the same in optimised and unoptimised code.

The first constraint means that an instruction cannot be moved if doing so makes a new exception possible. Imagine that the Jimple instructions below represent machine code corresponding to the Java source, and that a virtual machine is trying to schedule the code to minimise pipeline bubbles due to load latencies.

Java **Jimple**

```
if (o1.field < 0)                 $i0 = r1.field;
  o1.field += o2.field;           if $i0 >= 0 goto label0;
                                  $i1 = r1.field;
                                  $i2 = r2.field;
                                  $i3 = $i1 + $i2;
                                  r1.field = $i3;
```

(16)

A compiler could reuse the value read from `r1` to avoid an extra load:

```
$i0 = r1.field;
if $i0 >= 0 goto label0;
$i2 = r2.field;
$i3 = $i0 + $i2;
r1.field = $i3;
```

(18)

It would not be permissible, though, to move the read of `r2.field` before the `if` statement in an attempt to avoid a pipeline bubble, like this:

```
$i0 = r1.field;
$i2 = r2.field;
if $i0 >= 0 goto label0;
$i3 = $i0 + $i2;
r1.field = $i3;
```

(19)

because doing so makes possible the raising of a `NullPointerException` in the case where `r1.field` $<$ 0 and `r2` is null, which would not have occurred in the unoptimised original.

The second constraint prevents an optimiser from interchanging instructions which may throw differing exceptions. In the following example,

Java	**Jimple**	
`int result = o1.field/d;`	`$i2 = r1.field;`	(20)
`result = result + o2.field;`	`i1 = $i2 / i0;`	
	`$i3 = r2.field;`	
	`i4 = i1 + $i3;`	

this constraint forbids moving the read of `r2.field` before the division, like this:

```
$i2 = r1.field;                                                    (22)
$i3 = r2.field;
i1 = $i2 / i0;
i4 = i1 + $i3;
```

since if `i0` $=$ 0 and `r2` $=$ `null`, such an optimised program will throw `NullPointerException`, while the unoptimised program would throw `ArithmeticException`.

The third constraint rules out optimisations which would change the observed state when an exception is thrown. Imagine that the following instructions occur in the scope of a handler for `ArithmeticException` which reads `$i3`:

Java	**Jimple**	
`o1.field = o1.field/d;`	`$i1 = r1.field;`	(23)
`o2.field = o2.field/d;`	`$i2 = $i1/i0;`	
	`r1.field = $i2;`	
	`$i3 = r2.field;`	
	`$i4 = $i3/i0;`	
	`r2.field = $i4;`	

Then a compiler could not move the read of `r1.field` ahead of the first division like this:

```
$i1 = r1.field;                                                    (25)
r1.field = $i2;
$i2 = $i1/i0;
$i3 = r2.field;
$i4 = $i3/i0;
r2.field = $i4;
```

since the value of `$i3` seen by such a handler could differ in the optimised code from what it would be in unoptimised code.

Chambers et. al. show how to model the dependences produced by precise exceptions by treating every potentially excepting instruction as if it wrote a single abstract location with its exception type (so the order of exceptions thrown cannot change), and as if it wrote all the variables which are live in its exception handler (to preserve the state observed by handlers). Gupta et. al. [5] suggest techniques for avoiding conservative overestimates of the state visible to exception handlers.

2.6 Exceptions and the Java bytecode verifier

Before it executes methods in a newly loaded class, a Java virtual machine is supposed to verify that the methods contain legitimate bytecode, following a procedure prescribed in *The Java Virtual Machine Specification*. Some of the properties verified involve purely structural characteristics of the class file which are unaffected by control flow, such as whether type descriptors in the class pool are syntactically correct or whether all branches are to the beginning of legitimate instructions. Such checks have no impact on how compilers deal with exceptions.

Other properties checked by the verifier, such as ensuring that no local variables are read before being initialised, can only be confirmed by performing a dataflow analysis of each method's code array. So any compiler producing Java bytecode needs to be aware of the assumptions underlying that dataflow analysis if it is to produce verifiable code. In particular, the specification says that when this analysis propagates values, it includes among the successors of each instruction "[a]ny exception handlers for this instruction." Judging from existing implementations, "any" exception handlers includes those handlers which cover the instruction, but catch exceptions that it cannot throw.

Consider the following bytecode:

(26)

```
static int local1Undefined(int[],int);
  Code:
   0:   aload_0
   1:   iload_1
   2:   iaload
   3:   istore_2
   4:   iload_2
   5:   iload_1
   6:   idiv    ; Only instruction that can throw an exception to handler at 8.
   7:   ireturn
   8:   pop     ; Catch and pop the exception.
   9:   iload_2 ; We know local 2 must be defined, but the verifier doesn't.
   10:  ireturn
  Exception table:
   from   to  target type
     0     8     8   Class java/lang/ArithmeticException
```

To a human reader—or to a compiler using a CFG which contains edges only for exceptions which can actually be thrown—it is clear that the `idiv` at index 6 is the only protected instruction which can throw an `ArithmeticException`, so the handler at index 8 cannot be reached until after instructions 0 through 5 have been executed. But the verifier thinks that any instruction from 0 to 7, inclusive, can throw an exception to the handler, so attempts to run this method will fail, with the message:

```
Exception in thread "main" java.lang.VerifyError: (class: LocalVerification1,
method: local1Undefined
signature: ([II)I) Accessing value from uninitialised register 2
```

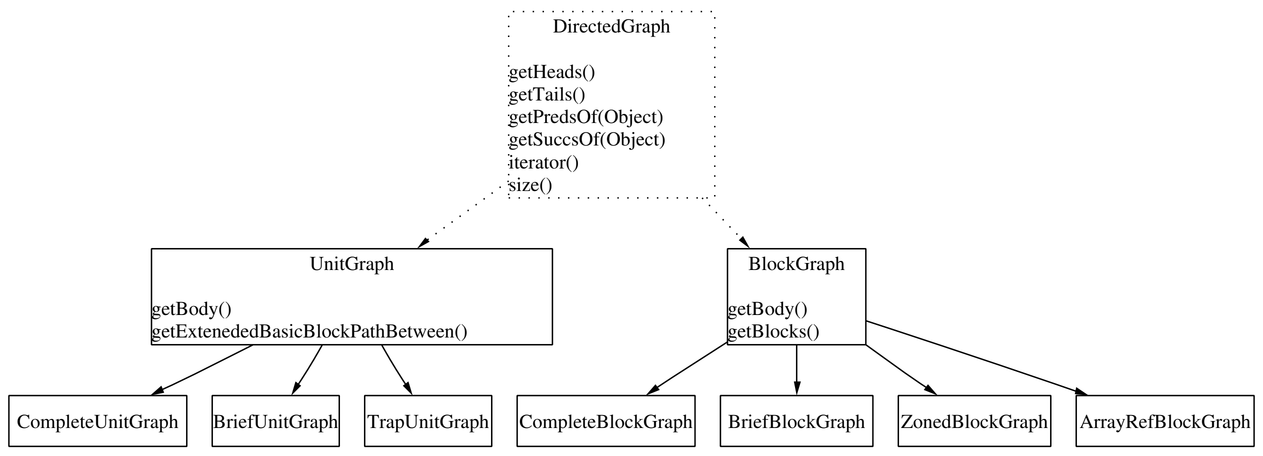

Figure 5: CFG classes in Soot 2.1.0

The modifications to Soot described in Section 3 prune unrealizable exceptional paths out of control flow graphs, so avoiding the production of unverifiable code is discussed in laborious detail in subsection 3.4.

3 Pruning Soot's control flow graphs

This section is an after action report, describing a project to prune from Soot's control flow graphs any edges to exception handlers which originate from instructions that cannot actually throw an exception matching the handler's `catch` parameter. It begins by sketching the family of classes which implement control flow graphs in Soot's 2.1.0 release. It proceeds to describe the modified interfaces provided by the replacement control flow graphs, then describes the algorithms used to produce them. Finally the section recounts in tedious detail the circumstances under which Soot might produce unverifiable code from pruned CFGs, and how to avoid those circumstances.

3.1 The old graph family tree

Figure 5 depicts the inheritance hierarchy among the classes implementing control flow graphs in Soot 2.1.0 and provides highlights of the interface they provide. All classes are members of the package `soot.toolkits.graph`.

`DirectedGraph` is an interface which declares access routines common to all implementations of directed graphs (Soot includes other subclasses of `DirectedGraph` in addition to the CFG classes shown in the diagram). The methods `getHeads()`, `getTails()`, and `iterator()` provide access to the nodes in the graph, while `getPredsOf()` and `getSuccsOf()` let you navigate from one node to its predecessors or successors.

A `UnitGraph` is a control flow graph where each graph node represents a unit, "units" being Soot's abstraction for individual statements or instructions in any of its intermediate representations for program code. A `BlockGraph` is a control flow graph where each node represents a basic block. The bulk of the implementation of these two classes is accounted for by their constructors, which take a method body as an argument and produce the corresponding control flow graph.

The subclasses of `UnitGraph` and `BlockGraph` are little more than wrappers around a call to the constructor of their parent classes. They differ only in that they pass to the constructors various flags which modify the algorithm used to construct the CFG.

`CompleteUnitGraph` is a CFG including a node for each unit, and incorporating edges representing exceptional control flow. Soot represents exception table entries by "traps", which comprise a pair of unit references delimiting the protected area, a reference to the initial unit in the exception handler, and the type of the `catch` parameter. A `CompleteUnitGraph` contains edges to each trap handler from every unit protected by the trap, as well as from the the predecessors of the first trapped unit.[14]

`CompleteUnitGraph` is very heavily used. The Soot 2.1.0 source contains at least 25 invocations of the `CompleteUnitGraph` constructor, spread among 23 different classes. When Soot is called without any arguments to disable default transformations or enable optional ones, the process of "jimplifying" the input—producing its representation in the Jimple intermediate representation—involves constructing at least five `CompleteUnitGraph`s for each method. Subsequently generating a class file from the Jimple representation requires at least two more `CompleteUnitGraph`s for each method.

Figure 9 shows the Soot 2.1.0 `CompleteUnitGraph` corresponding to the Jimple code in figure 6. The shaded nodes are those which are returned by either `getHeads()` or `getTails()`.

`BriefUnitGraph` is a CFG including a node for each unit, but containing no edges for exceptional control flow. Soot 2.1.0 contains 7 invocations of the `BriefUnitGraph` constructor in 7 classes. None of the invocations occur in the default jimplification process, but two are required to produce a class file via the Baf intermediate representation.

Figure 7 show the `BriefUnitGraph` corresponding to the example in figure 6.

`TrapUnitGraph` is a variant of `CompleteUnitGraph` which omits exceptional edges from the predecessors of a trap's first protected unit. `TrapUnitGraph`s are used exclusively in the construction of `DavaBody`s for decompiling class files to Java source. Presumably the graphs serve to record which units are covered by which traps.

`CompleteBlockGraph` is a CFG including a node for each basic block, and incorporating edges representing control flow to exception handlers from excepting units, which terminate basic blocks. `CompleteBlockGraph`s are constructed during the production of the Shimple SSA representation.

Figure 11 shows the Soot 2.1.0 `CompleteBlockGraph` corresponding to the example in figure 6.

`BriefBlockGraph` is a CFG including a node for each basic block, but with no edges representing exceptional control flow. `BriefBlockGraph`s are constructed by the `LoopConditionUnroller` associated with Soot's partial redundancy elimination, and by `JasminClass`, during the production of class files via Baf.

Figure 8 shows the `BriefBlockGraph` corresponding to the example in figure 6.

[14]Subsection 2.4 provides the rationale for including edges from the predecessors of trapped units. The Soot 2.1.0 implementation is flawed in that it contains edges only from the predecessors of the first protected unit, presumably because the implementors assumed that the predecessors of all subsequent units would themselves be trapped. But in arbitrary Java bytecode—though not in class files generated from Java source—it is possible to branch into the middle of a protected area, so the Soot 2.1.0 `CompleteUnitGraph` may lack some predecessor edges.

ZonedBlockGraph is a variant of BriefBlockGraph in which code protected by different sets of exception handlers is separated into different basic blocks (that is, the boundaries between protected areas create block leaders). ZonedBlockGraphs are used exclusively by the Baf LoadStoreOptimizer, which needs to ensure that it does not amalgamate operations which are in the scope of different exception handlers.

ArrayRefBlockGraph is a variant of BriefBlockGraph in which each array reference starts a new basic block. It is used exclusively by the ArrayBoundsCheckerAnalysis class, which finds array references for which runtime ArrayIndexOutOfBoundsException checks may be omitted.

An important feature of the CFG classes is that they are independent of the underlying intermediate representation, relying only on the facilities provided by the Unit interface common to all IRs. The LoopConditionUnroller, for example, builds BriefBlockGraphs from representations in Jimple, Soot's three-address representation, while JasminClass builds BriefBlockGraphs from methods represented in Baf, a stack-based representation. At different stages in Soot, CompleteUnitGraphs are built from methods represented by Jimple, Baf, Shimple (a variant of Jimple with ϕ nodes for SSA representations) and Grimp (Jimple with aggregated expressions).

3.2 The new graph family tree

Figure 13 shows the CFG hierarchy as modified to prune CFGs. The ExceptionalUnitGraph and ExceptionalBlockGraph classes are reimplementations of the original CompleteUnitGraph and CompleteBlockGraph classes which add machinery to trim unrealizable exceptional control flow. The new CompleteUnitGraph and CompleteBlockGraph classes are wrappers aound these Exceptional classes, included to allow old client code to compile without modification.[15]

Since the immediate aim of this project is to experiment with pruning unrealizable exceptional paths while minimising changes to the rest of Soot, it is important that the old interfaces to the graph classes remain usable. On the other hand, compiler analyses are probably better served by graphs which distinguish exceptional control flow from regular control flow, as suggested in section 2.4. So the CompleteUnitGraph interface has been extended in ExceptionalUnitGraph to include new methods for exception-savvy analyses. getExceptionalPredsOf(), getExceptionalSuccsOf(), getUnexceptionalPredsOf(), and getUnexceptionalSuccsOf() are similar to the original getPredsOf() and getSuccsOf, except that they distinguish between exceptional and regular control flow. getExceptionDests() returns a set of structures which indicate which exceptions a unit might throw, and which traps, if any, catch those exceptions. For details, see the javadoc comments for ExceptionalUnitGraph.

Figure 10 shows the pruned ExceptionalUnitGraph corresponding to the example in figure 6, while figure 12 shows the pruned ExceptionalBlockGraph. Regular control flow is shown by black edges and exceptional control flow by light grey edges. The dashed grey edges represent the structures returned by getExceptionDests(); we will call them "exception destination edges", as distinguished from "exceptional control flow edges". They are labelled with the types of exceptions thrown and either connect the thrower with a catching handler, or with nothing at all, should the exception escape the method. When exception class names are surrounded by parentheses, they stand for the named class, plus all of its subclasses.

[15]While the Exceptional graphs were written to remove unrealizable exceptional edges from CFGs, it is not necessarily the case that an Exceptional graph is a subset of the Complete graph which earlier releases of Soot would have generated for the same method: under rare circumstances Exceptional graphs may include some edges that were missing from the old Complete graph classes, because the old graphs could omit edges from some predecessors of excepting units.

Java

```
int m(int[] a, int i, int j) {
    int sum = 0;
    for (int k = i; k < j; k++) {
        try {
            sum += a[k];
        } catch (NullPointerException e) {
            return 0;
        } catch (ArrayIndexOutOfBoundsException e) {
            sum += 0;
        }
    }
    return sum;
}
```

Jimple

```
int m(int[], int, int) {
    BasicCFGExample r0;
    int[] r1;
    int i0, i1, i2, i3, $i4;
    java.lang.NullPointerException r2, $r3;
    java.lang.ArrayIndexOutOfBoundsException $r4, r5;
    r0 := @this;
    r1 := @parameter0;
    i0 := @parameter1;
    i1 := @parameter2;
    i2 = 0;
    i3 = i0;
 label0:
    if i3 >= i1 goto label6;
 label1:
    $i4 = r1[i3];
    i2 = i2 + $i4;
 label2:
    goto label5;
 label3:
    $r3 := @caughtexception;
    r2 = $r3;
    return 0;
 label4:
    $r4 := @caughtexception;
    r5 = $r4;
    i2 = i2 + 0;
 label5:
    i3 = i3 + 1;
    goto label0;
 label6:
    return i2;

    catch java.lang.NullPointerException from label1 to label2 with label3;
    catch java.lang.ArrayIndexOutOfBoundsException from label1 to label2 with label4;
}
```

Figure 6: `BasicCFGExample.m()` example

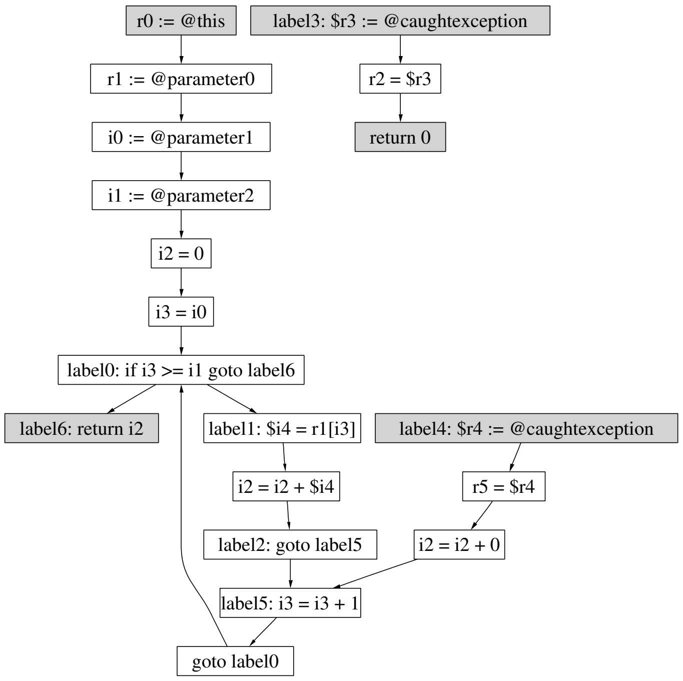

Figure 7: `BriefUnitGraph` for `BasicCFGExample.m()` (unchanged from 2.1.0)

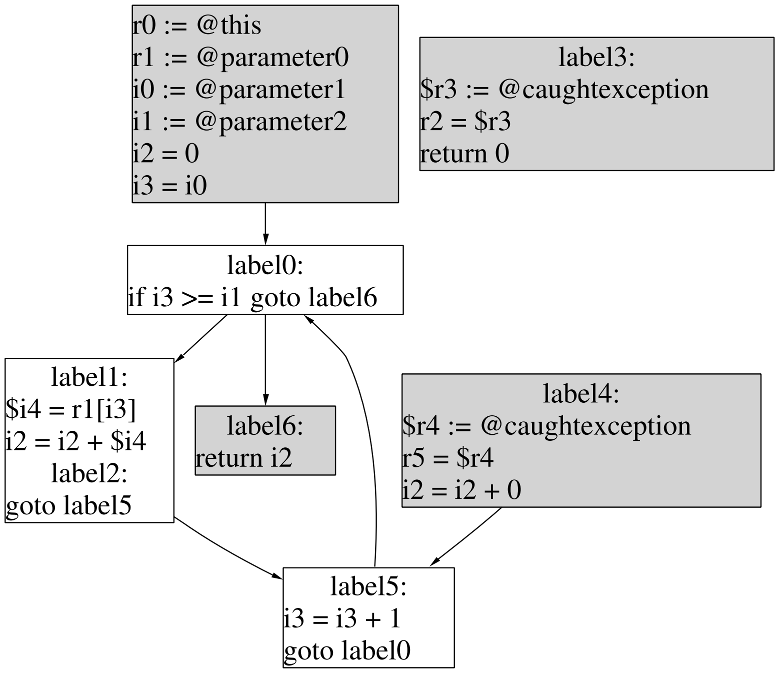

Figure 8: `BriefBlockGraph` for `BasicCFGExample.m()` (unchanged from 2.1.0)

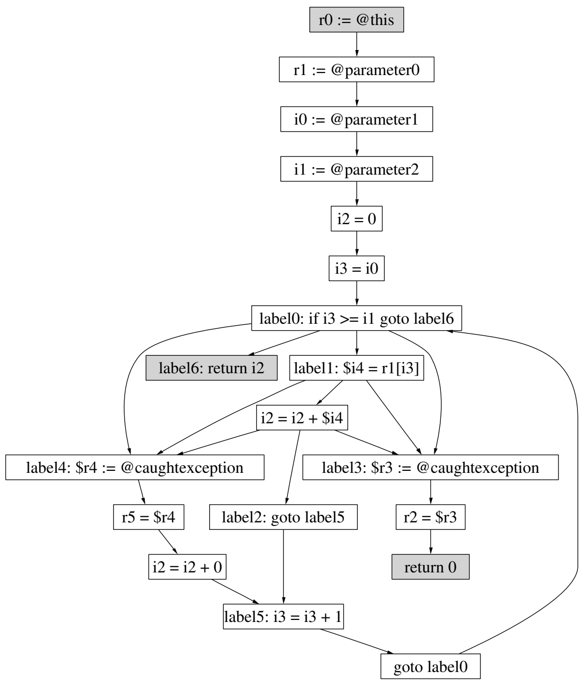

Figure 9: Soot 2.1.0 `CompleteUnitGraph` for `BasicCFGExample.m()`

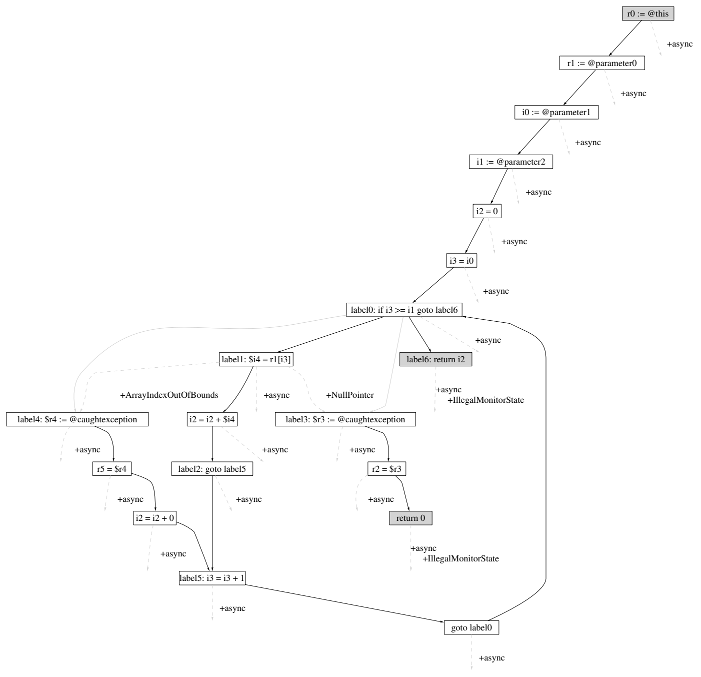

Figure 10: Pruned `ExceptionalUnitGraph` for `BasicCFGExample.m()`

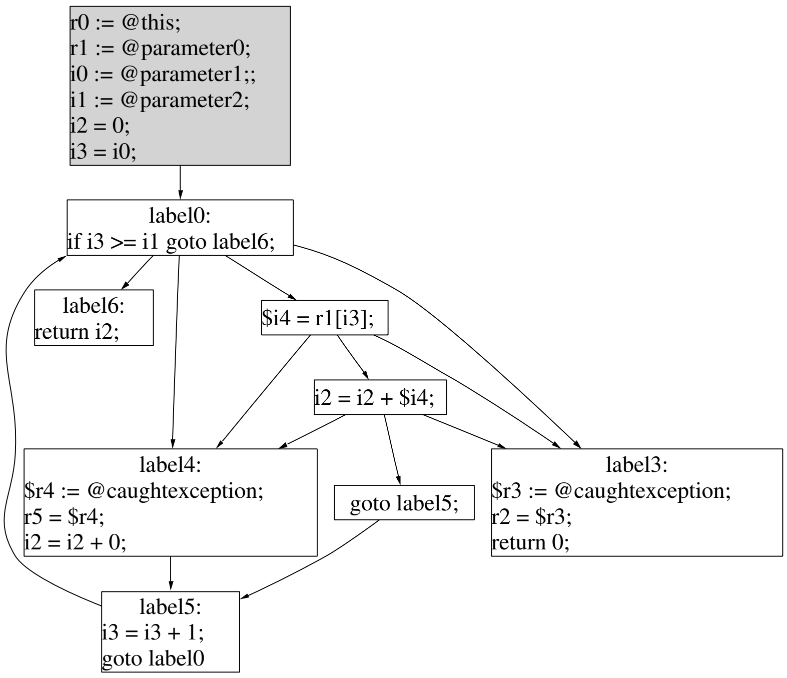

Figure 11: Soot 2.1.0 `CompleteBlockGraph` for `BasicCFGExample.m()`

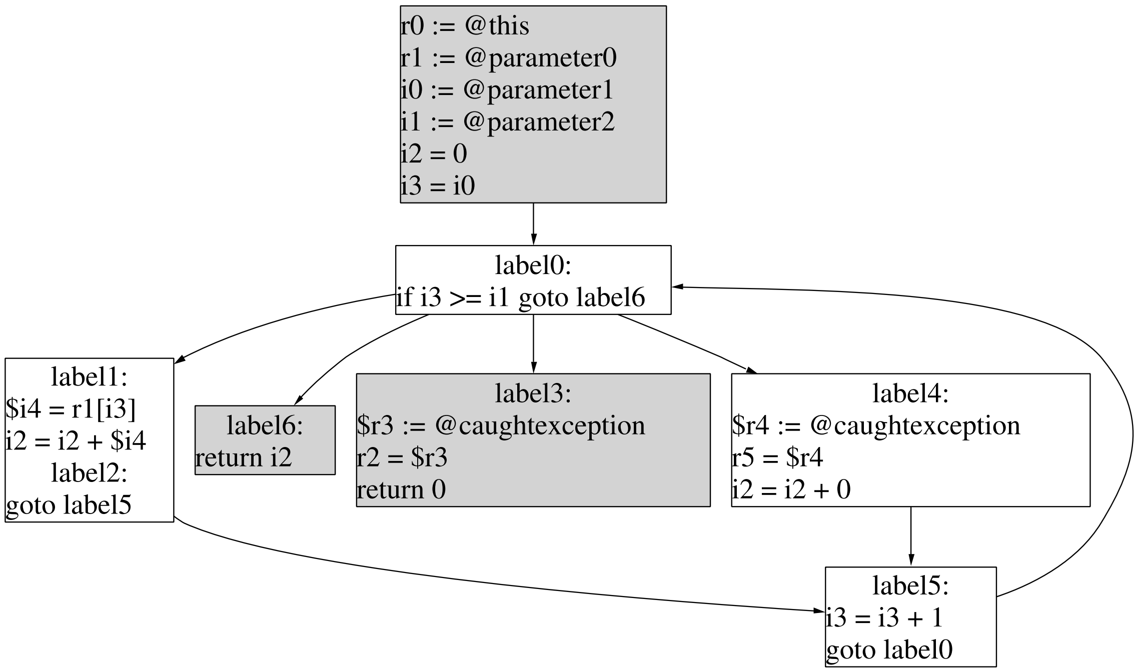

Figure 12: Pruned `ExceptionalBlockGraph` for `BasicCFGExample.m()`

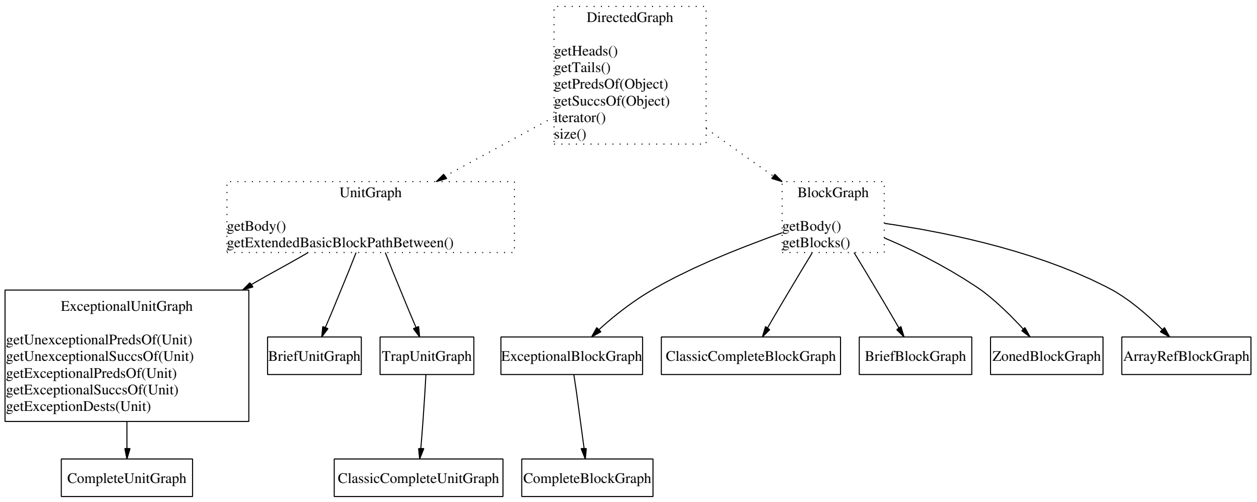

Figure 13: CFG classes with pruned exceptions

The visual clutter created by the exception destination edges in figure 10 obscures the fact that there are fewer control flow edges in the pruned graph. This is more evident in a comparison of figures 11 and 12, where the three statements forming the body of the loop remain a single basic block in the pruned graph, and where the exception handlers have only one predecessor each.[16]

Figure 13 shows two other additions: `ClassicCompleteUnitGraph` and `ClassicCompleteBlock-Graph`. They exist only for the purposes of validating the reimplementation of CFGs and are discussed in section 3.5.

The other apparent difference between the hierarchies depicted in figures 5 and 13 is the dotted borders around `UnitGraph` and `BlockGraph`, indicating that these classes are now abstract. In a fit of object-oriented orthodoxy, we discarded the original design where the distinctions between the various subclasses of `UnitGraph` and `BlockGraph` were actually implemented within the `UnitGraph` and `BlockGraph` constructors, and the subclasses were simply wrappers which supplied the superclass constructor with a flag indicating which subclass to build. Instead, `BlockGraph` and `UnitGraph` now provide some infrastructure for constructing graphs, and the subclasses mix and match pieces from the infrastructure. For example, the different subclasses of `BlockGraph` override a method called `computeLeaders()` to redefine which units begin a new basic block.

Making `UnitGraph` and `BlockGraph` abstract renders their constructors inaccessible, eliminating one of the drawbacks of the old design, that it allows client code to bypass the subclass constructors and thus build instances of `UnitGraph` or `BlockGraph` whose type does not indicate which variant of the constructor was called. The redesign also allows the addition of new subclasses without requiring modifications to existing code. On the other hand, the new design may be harder to comprehend, since figuring out how a graph is constructed now requires understanding the relationships between a number of classes, rather than examining a single method in a single file.

[16]Figure 12 illustrates a transformation that could be implemented in Soot, though it probably would not constitute an optimisation since Soot does not generate native machine code. The block starting at `label5` could be eliminated if its code were duplicated at the end of the blocks starting with `label1` and `label4`, providing longer basic blocks for instruction scheduling (a superscaler processor could perform the additions assigning to `i2` and `i3` simultaneously).

3.3 Building Pruned Graphs

Three families of classes are involved in the pruning of Soot control flow graphs. `ExceptionalUnitGraph`'s constructor queries a `ThrowAnalysis` to determine which exceptions each unit might throw. The `ThrowAnalysis` responds by returning a `ThrowableSet`, which represents a set of subclasses of `java.lang.Throwable`.

Ostensibly, this section proceeds by describing in turn `ThrowAnalysis`, `ThrowableSet`, and then the various `Graph` constructors. In practice, the exposition will not be strictly linear because the design and implementation of the first two classes evolved in response to lessons learned while building the graphs, while the graph building algorithms were in turn influenced by limitations in the representation of `ThrowableSet`.

3.3.1 Who throws what: `ThrowAnalysis`

`ThrowAnalysis` is an interface consisting of a single method:

```
public ThrowableSet mightThrow(Unit u);                         (27)
```

There are currently two implementations:

`PedanticThrowAnalysis.mightThrow(Unit)` always returns a `ThrowableSet` which represents `java.lang.Throwable` and all its subtypes. Strictly speaking, this is the correct response for all units in any Java program, since the deprecated library call `java.lang.Thread.Stop(Throwable)` allows a Java thread to raise an arbitrary `Throwable` in another thread at any time, asynchronously from the perspective of the victim thread.

Control flow graphs built using `PedanticThrowAnalysis` [17] are usually indistinguishable from graphs built by Soot 2.1.0, since they include edges to exception handlers from all protected units. They differ only in the case where there is a branch into the middle of the protected area, since the new graph construction algorithms include edges from such branches to the handler (the branches being predecessors of a protected unit).

`UnitThrowAnalysis.mightThrow(Unit)` is essentially a pair of big switch statements, one mapping from the different subtypes of `soot.Unit` to the exceptions such a unit may throw, and one mapping from the different subtypes of `soot.Value` to the exceptions that the evaluation of such a value might throw. The switches encode the information in Appendix A.

The only significant complication is the requirement to deal with multiple intermediate representations, including the aggregated expressions of the Grimp representation. It is this complication that forces the use of two switches, since `mightThrow(Unit)` needs to make recursive calls to `mightThrow(Value)` in order to accumulate the exceptions that could be thrown by subexpressions.

A second, minor, complication is that `UnitThrowAnalysis` needs to be capable of dealing with untyped representations of methods (since type resolution cannot occur until after an initial CFG is constructed) but should use type information when it is present, so that `ExceptionalUnitGraphs` built after type resolution occurs can take advantage of the extra information.

[17]And with `ExceptionalUnitGraph`'s `omitExceptingUnitEdges` parameter set to `false`, see section 3.3.3

While it looks at each unit in isolation, `UnitThrowAnalysis` is still capable of a few simple deductions to rule out unrealizable exceptions. Most of these are examples where estimates of possible exceptions can be improved after typing:

- Assignments to primitive array elements cannot raise `ArrayStoreException`.
- Division by non-zero integer constants cannot raise `ArithmeticException`.
- Casts of some class to itself or a superclass cannot raise `ClassCastException`.
- New arrays whose dimensions are compile-time constants cannot raise `NegativeArraySize-Exception` unless those constants are negative.

In retrospect it is clear that a mistake was made in the design of `UnitThrowAnalysis`, despite its simplicity. Currently the switches map from Baf and Jimple/Grimp/Shimple entities to the corresponding exceptions (in terms of Appendix A, from values in columns 3 and 4 of the table to the corresponding values in column 2). A better design would consist of a core switch from the individual Java bytecode instructions to the exceptions they can throw (mapping from column 1 to column 2 of Appendix A), supplemented with maps from the entities in the various IRs to the bytecode instructions they represent. Such a design would more clearly distinguish information fixed externally by the JVM specification from information decided internally by the implementors of Soot. It might also reduce the work required to add new intermediate representations.

There are two routes to incorporating more sophisticated analyses which could rule out more exceptions by examining more than one unit at a time (such as the `arraycheck` and `nullcheck` packages in `soot.jimple.toolkits.annotation`).

One is to provide a new implementation of `ThrowAnalysis` which incorporates the deductions to rule out more exceptions. A simple example would be to create a variant of `UnitThrowAnalysis` which takes a `LocalDefs` object as an argument. This variant could be used when an initial CFG is already available, and would allow tighter typing of the arguments of `throw` units. Currently, when the type of the argument to a throw is known to be t, `UnitThrowAnalysis` says that the `throw` may throw t or any of t's subtypes, unless the argument happens to be an instance of Grimp's `NewInvokeExpr`, in which case the exception class can be restricted to t itself. With a `LocalDefs` parameter, the exception class could also be restricted to t itself in the case where the `throw`'s argument has a single definition, and that definition is a `NewExpr`.

The second route to incorporating more sophisticated analyses to rule out impossible exceptions is to record the `ThrowableSet` objects returned by `UnitThrowAnalysis` for the units of a method, and then remove from those sets any exceptions which subsequent analysis proves impossible. This route, though, requires improvements to the current `ThrowableSet` class.

3.3.2 Representing who throws what: `ThrowableSet`

The Java libraries define several hundred subclasses of `Throwable` and programmers are free to define new `Throwable`s of their own. The large and open-ended number of exception classes rules out any representation of `ThrowableSet` which depends on a complete inventory of the universe of possible members, as would a bitset. It also implies that for cases where a unit is known to throw an object which is an instance of some class c or any of c's subclasses (see section 2.3.1), the representation of the exception type actually needs to stand for "any subtype of c", rather than consisting of the collection of c's subclasses that happen to be known at the time.

Since the addition of the Spark pointer analysis framework to Soot, it has been possible to represent the exceptions which a unit may throw with instances of `soot.RefLikeType`, where each instance is either a `soot.RefType`—if the exception is known to have a particular run-time class—or a `soot.AnySubType`—if the exception might be any subclass of a particular class.

While the exceptions each unit might throw are, at bottom, represented by a a `java.util.Set` of `RefLikeType`s, that `Set` is hidden within the class `ThrowableSet` to provide encapsulation, to improve efficiency, and to allow for the desire, as yet unrealized, to add a "remove" facility to `ThrowableSet`.

First, encapsulation. If the exceptions thrown by a unit were exposed to clients of `ThrowableSet` as a `Set` of `RefLikeType`s, those clients would have to know how to interpret the collection and its contents. In particular, they would have to know how to deal with `AnySubType` elements, and how to normalise the set when the addition of such elements subsumes existing elements (for example, if one adds `AnySubtype(RuntimeException)` to the set {`ArrayIndexOutOfBoundsException,ArithmeticException`}, the result should be {`AnySubtype(RuntimeException)`}). `ThrowableSet` spares its callers the burdens and the temptations implied by knowledge of its implementation by providing typed `add()` methods and a `catchableAs(RefType)` method for checking if a particular `catch` parameter might match an element in the `ThrowableSet`.

Second, efficiency. A program being analysed might contain millions of ALU operations which throw only asynchronous exceptions. It seems worthwhile to have `ThrowAnalysis` return references to a single copy of the same `ThrowableSet` for each of those instructions, instead of constructing millions of duplicate `Set`s for the garbage collector to track. So `ThrowableSet`s are immutable objects, created by a `ThrowableSet.Manager` which ensures there is only one copy of each set. New `ThrowableSet`s can be created only by adding a `RefType`, an `AnySubType` or a `ThrowableSet` to an already existing `ThrowableSet` (the `Manager` creates the initial empty set). The need to check whether added elements are already accounted for by `AnySubType`s already in the set or, conversely, whether added elements subsume any existing elements, can make additions an expensive process, so each `ThrowableSet` "memoizes" the add operation, recording which sets have been produced from it by previous additions.

Finally[18], defining `ThrowableSet` left open the possibility that it could have a "remove" operation. Of course any implementation of `java.lang.util.Set` has a `remove()` method that lets you delete an element, but `Set.remove()` does not suffice to permit the removal of arbitrary `RefLikeType`s. That would require not only the ability to recognise that removing `AnySubType(`c`)` from a set implies the removal of any `RefType`s which are subclasses of c. It also requires the ability to record "holes" in the type hierarchy, so that you can represent the set of `AnySubType(`c_0`)` minus `AnySubType(`c_1`)`, where c_1 is a subclass of c_0.

To allow for the removal of arbitrary `RefLikeType`s and collections of `RefLikeType`s, `ThrowableSet`'s implementation needs to consist of a collection of "positive" and "negative" terms. We have many scribbled attempts to prove the soundness of such a design to ourselves, but no satisfactory conclusion as yet. Since one only needs to add exceptions to a set in order to construct pruned CFGs, we have proceeded without a remove operation. This is unsatisfactory in three respects:

1. Without the ability to remove elements, the algorithm for producing pruned CFGs is messier than it needs to be, and requires exposing the individual elements of `ThrowableSet` to the graph constructor, undermining `ThrowableSet`'s encapsulation.
2. Without a remove operation, static analyses which prove that some exceptions cannot occur must

[18]And the real reason this section of the report even exists.

re-implement the ThrowAnalysis interface in order to supply that information to the rest of Soot, rather than just removing elements from the ThrowableSets returned by UnitThrowAnalysis.

3. Without a remove operation, ExceptionalUnitGraph is an unsatisfactory tool for program understanding. This is best illustrated with an example. Figure 14 depicts the ExceptionalUnitGraph derived from the following Java source:

```
 1      int m(int[]a, int i) {
 2          try {                                               (28)
 3              return subscript(a,i);
 4          } catch (NullPointerException e) {
 5              return -1;
 6          } catch (IndexOutOfBoundsException e) {
 7              return -2;
 8          } catch (RuntimeException e) {
 9              return -3;
10          }
11      }
```

   The exception edge from label0 (line 3) to label4 (line 8) is labelled "+(Runtime)" when it should be labelled "+(Runtime)-(IndexOutOfBounds)-(NullPointer)", since IndexOutOfBounds-Exception and NullPointerException are caught by earlier handlers. Similarly, the exception edge that escapes from label0 should be labelled "+(Throwable)-(Runtime)".

3.3.3 Building UnitGraphs

The abstract UnitGraph class provides a method to create the edges that represent a method's unexceptional control flow. The BriefUnitGraph constructor simply calls that method to create CFGs with no exceptional control flow. The ExceptionalUnitGraph constructor uses the same routine to model unexceptional control flow. It then creates exception destination edges which link excepting statements to the handlers which catch their exceptions and uses those edges to guide the placement of exceptional control flow edges to handlers from excepting statements and their predecessors.

Static statistics computed from benchmarks (see Table 1) confirm a prediction most Java programmers could make from their own experience: throwers are many, but handlers are few. In the Java 1.4 libraries, for example, 60% of units can throw some Exception and 67% can throw an Exception or some synchronous Error, but less than 10% of units are in the scope of any handler[19] and 90% of methods define no handlers at all. So the algorithms which construct ExceptionalUnitGraphs should try to minimise the time and space they spend tracking the many potential exceptions which escape their method. That observation led to an initial idea for exceptional analysis which proved unworkable, mentioned here partly to prevent others from following the same misguided path, and partly to motivate the algorithm actually adopted.

The original design of ThrowAnalysis did not have a mightThrow(Unit) method to return the set of exceptions that its argument could potentially throw. Instead, it had a mightThrow(Unit, RefType) method which returned true if the passed unit could throw the passed exception and false otherwise. The idea was to check only for exceptions that could actually be caught, using the following algorithm:

[19]That is, are statically within the protected area of a handler defined by their own method. We have not performed the dynamic instrumentation that would be required to determine the percentage of instructions which are executed while their exceptions could be caught by a handler in some method further down the call stack.

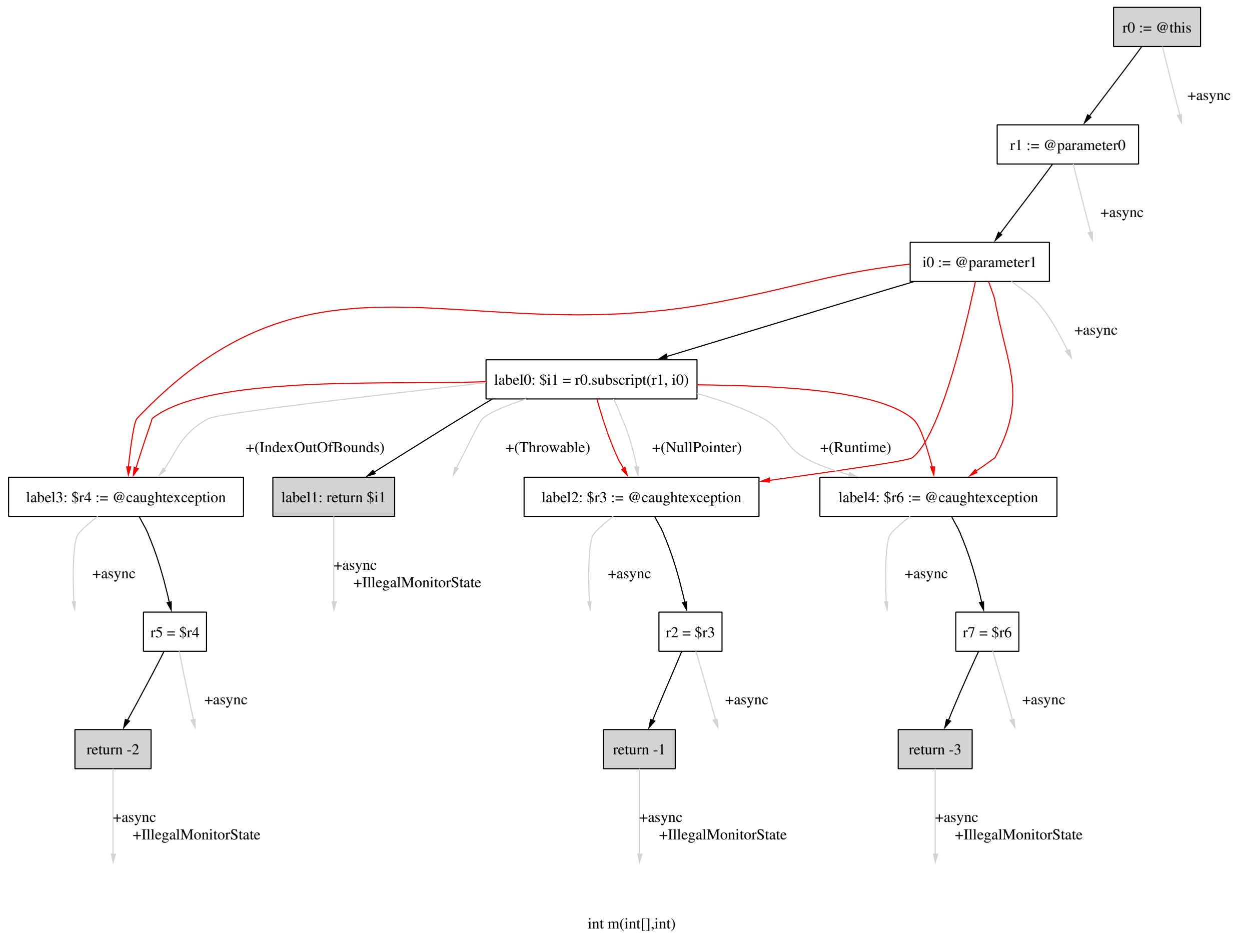

Figure 14: Example of misleading exception labels caused by `ThrowableSet`'s lack of remove

```
for each Trap, $t$,
    let $e$ be $t$'s catch parameter type
    for each Unit, $u$, trapped by $t$
        if mightThrow($u$, $e$)
            add an exception destination edge from $u$ to $t$
```

Unfortunately, one cannot iterate through the Traps in isolation like this, since when multiple Traps cover the same Unit, determining which ones may catch an exception requires considering the types caught by each trap, the types thrown by the unit, and the order of the traps in the exception table. For example, in the code

(29)

```
public void m(char[] a, String s, int j) {
  try {
    char c = s.charAt(j);
    a[j] = c;
  } catch (ArrayIndexOutOfBoundsException e) {
    // ...
```

```
 7        } catch (IndexOutOfBoundsException e) {
 8          // ...
 9        } catch (RuntimeException e) {
10          // ...
11        }
12      }
```

line 4 does have the potential to throw an exception whose type matches the `catch` parameter of line 7, but the CFG should include no exception edge from line 4 to line 7, since the exceptions that line 4 throws which are catchable as `IndexOutOfBoundsException`s will be intercepted by line 5 instead. On the other hand, we cannot let the fact that line 3 can throw to line 7 and that line 4 can throw to line 5 prevent us from seeing that both can throw `NullPointerException`s to line 9 as well.

To implement the idea of only checking for exceptions that could actually be caught, then, would require `ThrowAnalysis`'s interface to provide `mightThrowExceptFor(Unit, RefType, Set)`, where the `Set` contains `RefType`s which have already been caught. This routine would return true if the passed `Unit` could throw some exception that is catchable as the first `RefType`, but not as any of the `RefType`s in the passed set, allowing us, in the example above, to ask if line 4 can throw an `IndexOutOfBoundsException` which is not also an `ArrayIndexOutOfBoundsException`.

Such a design is unsatisfactory because it confounds the separate concerns of determining which exceptions a given unit can throw and tracking where they might be caught. Since it is possible to imagine other motivations for representing the entire set of exceptions a unit can throw (such as displaying them to programmers trying to understand exceptional control flow in some software), we adopted the designs of `ThrowAnalysis` and `ThrowableSet` described in previous sections. In an attempt to avoid most of the overhead of tracking uncaught exceptions, `ExceptionalUnitGraph`'s constructor does not compute `ThrowableSets` for methods which have no traps (for such methods, `getExceptionDests()` computes the set of possible exceptions for a unit on-the-fly, secure in the knowledge that all the exceptions necessarily escape the method).

If `ThrowableSet` provided the ability to remove exceptions, the algorithm to find exception destination edges would be obvious, elegant, and require no access to `ThrowableSet`'s internals:[20]

```
for each Unit, $u$, protected by at least one Trap
    let $s$ be the set of exceptions $u$ might throw
    for each Trap, $h$, protecting $u$, in order
        let $c$ be $h$'s catch parameter type
            if $s$.catchableAs($c$)
                add an exception destination edge, from $u$ to $h$
                remove (c) and all its subclasses from $s$
    any exceptions remaining in $s$ at this point escape the method.
```

Ugly compromises are necessary in the absence of `ThrowableSet.remove()`. The result is perhaps still obvious, but not elegant:

```
for each Unit, $u$, protected by at least one Trap
```

[20]This description fudges one inelegant detail: in order to label the exception destination edge with the exceptions actually thrown, rather than with "any subclass of c", the `remove()` operation would have to return the set of elements actually removed. But the returned representation could itself be a `ThrowableSet`, preserving the encapsulation.

```
let $s$ be the set of exceptions $u$ might throw
for each element, $e$, in $s$
    for each Trap, $t$, protecting $u$, in order
        let $c$ be $t$'s catch parameter type
        if $e$ is assignable to $c$
            add an exception destination edge, from $u$ to $t$
            continue to the next element in $s$, skipping remaining Traps
        if $e$ is AnySubType($e_0$) and $c$ is assignable to $e_0$
            add an exception destination edge, from $u$ to $t$
            continue to the next Trap, which may catch other subtypes of $e_0$
    if this point is reached, $e$ escapes the method
```

Once the method's exception destination edges have been established, creating the corresponding control flow edges is relatively simple. For every exception destination edge encoding that unit u may throw an exception to handler h, a control flow edge is added from every predecessor of u to h. If u has side-effects, or if it is an explicit `throw`, or if the `ExceptionalUnitGraph` constructor was passed `false` as the value for its `omitExceptingUnitEdges` parameter, an edge is also added from u itself to h.[21]

The new `CompleteUnitGraph` class is just a wrapper which calls the `ExceptionalUnitGraph` constructor specifying the use of a `PedanticThrowAnalysis` and `omitExceptingUnitEdges == false`, since these settings produce an `ExceptionalUnitGraph` which is identical to the Soot 2.1.0 `CompleteUnit-Graph` for the same method, except for the rare cases where there is a branch into the middle of a protected area (in such cases, the 2.1.0 `CompleteUnitGraph` lacks exceptional control flow edges from the branch to any handlers that catch exceptions thrown by the branch's target). All the `CompleteUnitGraph` constructor calls within Soot 2.1.0 have been replaced by `ExceptionalUnitGraph` constructor calls using default values for the `ThrowAnalysis` and `omitExceptingUnitEdges` settings. For now, the default values are `PedanticThrowAnalysis` and `false` to preserve compatibility, but the defaults can be changed from the Soot command line.

There are two complications when deriving control flow edges from exception destination edges. The first is that if the first unit in the method is protected by a handler, and that method has no side effects, it isn't clear where the edge leading to the handler should start from: the throwing unit has no predecessor. The correct answer to this problem is probably to have a distinguished *begin* node for the graph, with edges from it to the first unit in the method, and to any handlers reachable from the first unit. That is not a course to be taken lightly, though, as the assumption that each node in a `UnitGraph` corresponds to some instruction in the underlying `Body` seems deeply rooted in Soot. Since this project aimed to make minimal changes to CFG clients, we opted to include as heads of the graph any exception handler units that may catch an exception thrown by the method's initial unit.[22]

The second complication when adding edges from the predecessors of an excepting unit to its handlers is the possibility that the first unit in the handler might itself throw an exception. In most of Soot's intermediate

[21]It is probably not necessary to include edges from the predecessors of an explicit `throw` to a handler that might catch its exceptions. It is not the case that all control flow to an exception handler from a `throw` is the result of the `throw`'s completing normally—any throw can implicitly raise `NullPointerException` or `IllegalMonitorStateException`, and in the Grimp IR, a `throw` may also raise arbitrary exceptions in the course of evaluating its argument. But because `throw`s have no effect on the computation other than raising their exception (and possibly evaluating it, in Grimp), no information is lost by connecting the `throw` itself to the handler, rather than its predecessors, even in the case where an implicit error has prevented the `throw` from raising its intended argument.

[22]Of course there is no fundamental difference between having multiple heads and having a distinguished *begin* node whose edges lead to the same set of heads. In practice, though, it is cleaner for clients to to deal with a single CFG entry point.

representations, this possibility could be ignored, because the first unit in each exception handler is a special `IdentityUnit` associating the caught exception with a local, for example:

```
  $r2 := @caughtexception
```

Such `IdentityUnit`s do not correspond to any real bytecode, so they could be considered immune even from asynchronous exceptions. The Baf intermediate representation, though, does not begin handlers with such `IdentityUnit`s. As a result, `ExceptionalUnitGraph` needs to perform a transitive closure when it creates exceptional control flow edges.

Both complications are illustrated by the following example, whose `ExceptionalUnitGraph` is depicted in figure 15:[23]

(30)

```
 1      static Object[][] array;
 2      static int m() {
 3          try {
 4              try {
 5                  array = new FirstUnitThrows[3][5];
 6              } catch (Error e1) {
 7                  try {
 8                      return -1;
 9                  } catch (InternalError e2) {
10                      return -2;
11                  }
12              }
13          } catch (VirtualMachineError e3) {
14              return -3;
15          }
16          return 0;
17      }
```

Line 5 (`label0` in the graph) does not throw any exceptions to the handler at line 13 (`label6`), since all `VirtualMachineError`s are `Error`s, and thus caught at line 6 (`label2`). But the handler at line 6 might itself throw a `VirtualMachineError` before completing any instruction, producing a transitive exceptional control flow edge from line `label0` to `label6`. Because `label2` would be executed before `label0` assigns any value to `$r2` and `label6` before `label2` assigns any value to `$r3`, they must be considered heads of the graph (hence their nodes are shaded in the figure).

The `TrapUnitGraph` constructor uses the same infrastructure as `ExceptionalUnitGraph`. To add edges from every trapped unit to the protecting handler, regardless of which exceptions the unit may throw, `Trap-UnitGraph` just builds exception destination edges using `PedanticThrowAnalysis` instead of `Unit-ThrowAnalysis`. Since it adds no edges from the predecessors of excepting units, it substitutes a much simpler algorithm for deriving control flow edges from exception destination edges. `ClassicComplete-UnitGraph` is a subclass of `TrapUnitGraph` because it needs only to add a few predecessor edges in order to duplicate the Soot 2.1.0 `CompleteUnitGraph`.

[23]Given the last paragraph, readers may be surprised to see that figure 15 is a Jimple graph rather than a Baf graph. Since we need to perform a transitive closure of exception control flow edges anyway, the current `UnitThrowAnalysis` does report that `Iden-tityUnits` can throw asynchronous exceptions, just because its implementation is simpler if everything can throw asynchronous exceptions. A Jimple graph trapping asynchronous exceptions is smaller and easier to understand than a Baf graph trapping more commonly caught exceptions, so it is used for the example.

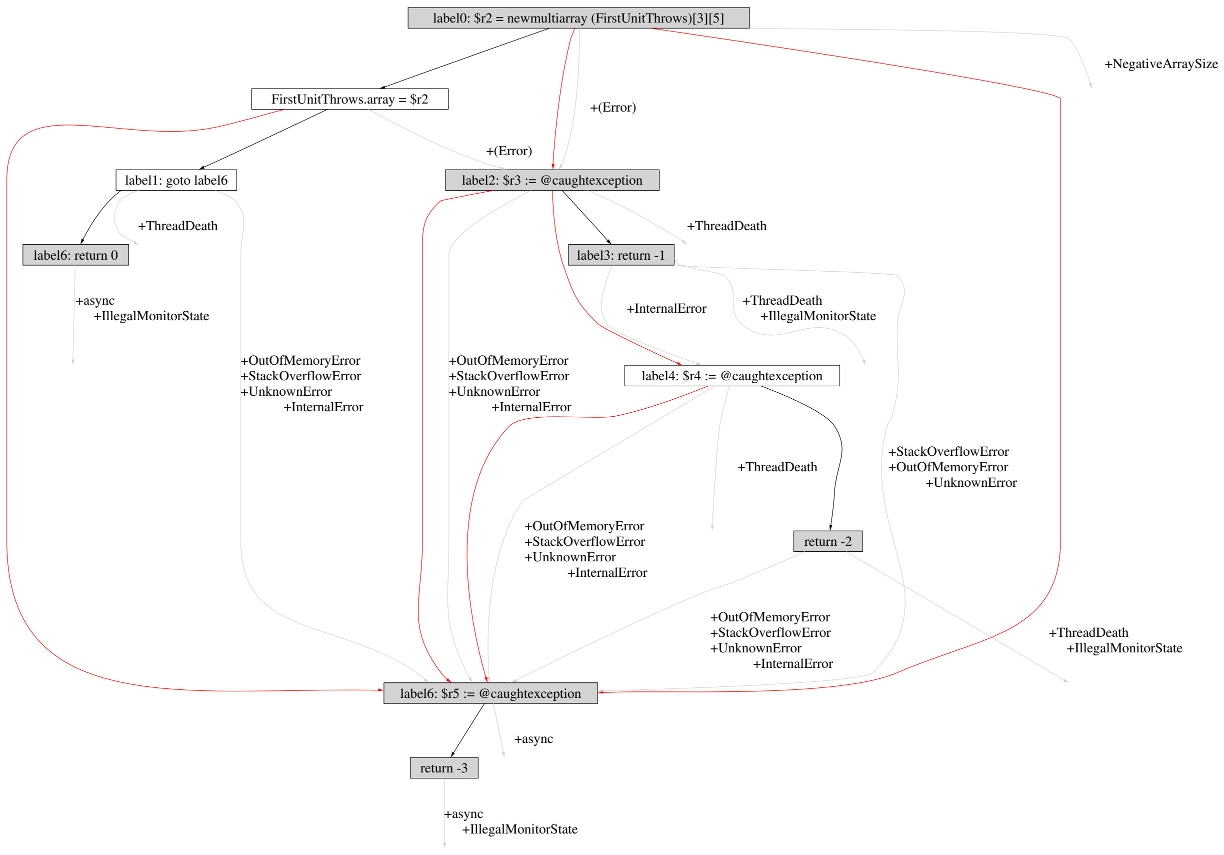

Figure 15: Example requiring a transitive closure of exceptional predecessors

3.3.4 Turning `UnitGraphs` into `BlockGraphs`

Soot 2.1.0's `CompleteBlockGraph` includes its own logic for analyzing exceptional control flow, separate from and not entirely consistent with that in the 2.1.0 `CompleteUnitGraph`. In the revised CFG classes, `BlockGraphs` are derived from `UnitGraphs`, guaranteeing consistency and reducing the costs of maintenance.

Implementing the derivation revealed that the notion of basic blocks is not so basic after all. A definition based purely in graph theory would say that every unit which has more than one predecessor, or which has a predecessor with more than one successor, must start a new basic block. If the idea of a basic block is "if you execute the first instruction in the block, you execute them all", then such a definition is correct. But for at least some uses of basic blocks—scheduling machine code, for example—what is important is not that each unit has only one successor and predecessor, but that the units are packed consecutively in the code array, so that executing them requires no branches. By the pure graph theory approach, the bold face instructions in the Jimple

```
    i2 = i0 + i1;                                                    (31)
    goto label1;
  label0:
    i3 = i0 * i1;
    return i3;
  label1:
    i4 = i2 * i1;
    return i4;
```

and in

```
    i2 = i0 + i1;                                                    (32)
    if i2 >= 0 goto label1;
  label1:
    i2 = -1 * i2;
    return i2;
```

would constitute basic blocks. One can even manufacture situations where exception handlers could be included in the end of a basic block, in cases where the exception can be proved to always happen, or if the excepting unit would fall through to the exception handler anyway (a situation which is difficult, but not impossible, to contrive).

Another sticking point is where to divide basic blocks when a unit with side effects may throw an exception that is caught within the method. If, for example, the following jimple statements occur within the scope of a handler which catches `ArithmeticException`, the only edge to the handler in the `Exceptional-UnitGraph` will have its origin at the statement preceding the assignment to `i4`, implying that the three statements form one basic block:

```
    i4 = r1[i0];                                                     (33)
    $i5 = i4 * i3;
    return $i5;
```

From one perspective, this is correct: the three statements form a basic block in the sense that if the effects of *any* of them occur, then the effects of *all* of them occur. They are not a basic block, though, if basic blockness means that if you *start* executing the first instruction, you will necessarily finish executing the last. The former conception is probably more relevant to dataflow analyses performed over basic blocks, while the latter would be more relevant to instruction scheduling.

The correct answer to the question of how to define basic blocks depends on the uses to which the blocks will be put. Since Soot does not produce native code, the important consideration is that when the effects of one instruction in a block occur, then the effects of all of them occur. So our block graphs consider example 33 to be a single basic block. On the other hand, we are not so radical as to allow basic blocks to span branches when the branching and target instruction have only one successor and predecessor, respectively, as in examples 31 and 32. So the `computeLeaders()` method supplied by the abstract `BlockGraph` class defines the following as block leaders:

- Any unit which has zero predecessors (for example, dead code following a `return` or `goto`) or more than one predecessor.
- Any unit which is the target of any branch (even if it has no other predecessors and the branch has no other successors.

- All successors of any unit which has more than one successor.
- The first unit in any exception handler.

The different subclasses of `BlockGraph` are distinguished mainly by deriving their blocks from different varieties of `UnitGraph`: `ExceptionalBlockGraph`s derive blocks from `ExceptionalUnitGraph`s while `BriefBlockGraph`s use `BriefUnitGraph`s. `ArrayRefBlockGraph` and `ZonedBlockGraph` derive their control flow from `BriefUnitGraph`s, but override the `computerLeaders()` method to add new conditions for demarcating basic blocks.

3.3.5 Designating heads and tails

A difficulty with the 2.1.0 CFG classes is that the meaning of `getHeads()` and `getTails()` is not quite nailed down. The documentation for `UnitGraph` and `BlockGraph` says that `getHeads()` returns the entry points for the graph while `getTails()` returns the exit points. One would expect the main significance of entry and exit points to be that the former are the starting points for forward dataflow analyses and the latter for backward dataflow analyses. In fact, many parts of Soot seem to use their own ad hoc methods to determine starting points in the method; the abstract classes `ForwardFlowAnalysis`, `BackwardFlow-Analysis`, and `ForwardBranchedFlowAnalysis` did not include code to initialize entry points until the release of Soot 1.2.3 in 2002.

The 2.1.0 implementation of `UnitGraph` actually defines `getHeads()` to return all nodes with no predecessors and `getTails()` to return all nodes with no successors. `BriefBlockGraph` defines `getHeads()` to return the first block in the method and all blocks which begin exception handlers, while `Complete-BlockGraph`'s `getHeads()` returns only the first block. The 2.1.0 `BlockGraph` classes do not implement `getTails()` at all. Since Soot 2.1.0 usually removes unreachable code and it adds edges from all trapped units to their handler, in practice the definitions of heads are usually consistent between the different types of graph, unless the method's first unit is at the beginning of a loop or exception handler, in which case the `UnitGraph`s have no heads at all.

This project has started an attempt to rationalize the definitions, but has yet to study all the uses of `get-Heads()` and `getTails()`—or the places where they should be used but are not—with the care needed to make a definitive decision. The new abstract `UnitGraph` class supplies a default `buildHeadsAndTails()` method which preserves the old behaviour where predecessor-less units are heads and successor-less units are tails. This method is used by all subclasses of `UnitGraph` except `ExceptionalUnitGraph`, which includes as a head the initial unit in the method, and the first unit in every handler that might catch an exception thrown by the initial unit in the method (see the discussion of transitive exceptional control flow at the end of section 3.3.3). `ExceptionalUnitGraph` defines as a tail all `return` instructions and all explicit `throw`s whose exception may escape the method.

Including escaping `throw`s leaves us open to a charge of inconsistency: the vast majority of units in every method might throw an implicit exception that escapes the method: why not designate all of them as tails as well? An initial response to this accusation is to recall the motivation for distinguishing exit points: as starting points for backwards flow analyses. Unless it is a `return` instruction, a unit that throws an escaping implicit exception will have unexceptional successors, ensuring that it is reached by backward dataflow analyses. We need to include escaping `throw`s because they may lack successors. But this response only changes the basis of the inconsistency: why then include as tails those `return` instructions which may throw exceptions caught within the method? They have successors from whom to receive backwards dataflow values. Our response to this rebuttal is a weak plea that we do not understand the universe of backward flow analyses well enough to know if some might need to distinguish normal exits from exceptional exits.

In one respect, though, the new family of control flow graphs does impose a new consistency: the various `BlockGraph`s define as heads all those blocks that begin with a unit which is a head according to the `UnitGraph` used to build the `BlockGraph`. Similarly, blocks are defined as tails if their last unit is a tail in the underlying `UnitGraph`.[24]

3.4 Mollifying the bytecode verifier

As described in section 2.6, the Java bytecode verifier performs a dataflow analysis which assumes that every instruction protected by an exception handler has the potential to throw an exception to that handler. Identifying situations where pruning unrealizable exceptions from Soot's CFGs might cause it to produce unverifiable code requires examining in turn each property checked by the verifier's dataflow analysis and determining whether it would be possible for a method to satisfy that property from the perspective of its pruned CFG, but not from the perspective of an unpruned CFG. [25]

3.4.1 Verifier checks affected by control flow

The verifier's dataflow analysis confirms that:

1. At the beginning of each instruction, the stack height and the types of values stored in each stack location are consistent, regardless of the path taken to reach the instruction.

2. The stack does not underflow or overflow.

3. At the beginning of each instruction, the possible types of values stored in any stack locations or local variables that the instruction reads are appropriate for the instruction. Similarly, the types of arguments to method invocations and field writes, the types of return instructions, and the types of values obtained from method invocations and field reads, must all match the declarations of the invoked methods and accessed fields.

4. No local can be used before being defined. This is really equivalent to the previous condition, if you think of "undefined" as a possible type for a local, but one which is illegal for any operation that reads the local.

   When the dataflow analysis merges the types of locals and stack locations flowing into each instruction, it effectively undefines locals which have inconsistent types flowing into them, giving them the the special type "unusable".

5. `protected` fields or methods are only accessed when the object containing them is an instance of the current class or one of its subclasses.[26]

[24]An assertion check triggers a fatal exception should any head unit not appear as the initial unit in its block, or any tail unit not appear at the end of its block.

[25]It would be gratifying to report we anticipated that pruning unrealizable exceptions from Soot's CFGs would cause it to generate unverifiable code, but we did not. When we were surprised to discover output classes which failed verification, we realized that to minimize future surprises we needed to examine the verifier specification in detail, rather than just deal with failed tests on a case-by-case basis. We include the examination of the specification here since it serves as an argument that Soot does in fact deal with all the verification issues associated with pruning exceptional control flow.

[26]This enforces a somewhat abstruse aspect of `protected` access: a subclass's method does not have blanket access to `protected` members of any instance of its superclass, but only of instances of itself and its own subclasses.

6. Some `<init>()` method is called for each new object instance before it is accessed, and no instance has `<init>()` executed more than once.

   To ensure the verifier can enforce this restriction, the specification imposes two subsidiary conditions:

   - no reference to an uninitialized object may be on the stack or stored in a local when any backward branch is made (that is, during a potential loop).
   - no reference to an uninitialized object may be stored in a local in code that is protected by an exception handler (since the stack is cleared on exceptions, it does not matter if uninitialized objects are on the stack in the scope of a handler).

   In practice, implementations of the verifier are more liberal than the specification: they do not complain about uninitialized objects stored in locals within the scope of a handler if the local is never accessed by the handler. This is fortunate for Soot, since the translation from bytecode to Jimple routinely stores uninitialized objects in local variables (even before the addition of pruned CFGs) as a consequence of simulating stack locations with locals. Most, but not all, of these stores are subsequently optimised away by the Baf `LoadStoreOptimizer`.

7. `<init>()` methods (except `java.lang.Object`'s) do nothing with the new object before calling the superclass's `<init>()`, except writing to fields defined by the current class (that is, the one defining the `<init>()` being verified). The superclass `<init>()` may be called indirectly, by calling another `<init>()` in the current class.

8. No `jsr` return address may be loaded from a local, nor may any of a list of arcane restrictions on the order of `jsr` and `ret` instructions be violated.

9. Execution cannot fall off the end of the code array.

There is also one check that is unaffected by control flow in principle, but actually affected in practice, because the JVM specification permits verifier implementations to delay checking the validity of references to other classes until the first execution of the accessing instruction:

10. All named fields, methods, and classes exist and the class being verified has access to all the methods and fields that it accesses.

Verifiers in the wild will actually accept a class that includes instructions which reference nonexistent or private members, so long as those instructions are not actually executed. If you compile the following classes, for example:

(34)

```
package package1;
import package2.SecondClass;

public class SometimesAccessPrivate {
    private int i;

    public SometimesAccessPrivate(int i, int j, SecondClass s) {
        try {
            this.i = i/j;
        } catch (ArithmeticException e) {
            this.i = (int) s.f;
        }
    }
```

```
package package2;

public class SecondClass {
   public float f;       // Leave this uncommented while compiling
                         // SometimesAccessPrivate
   // private float f;   // Leave this uncommented instead while
                         // compiling SecondClass itself.
   public SecondClass(float f) {
       this.f = f;
   }
}
```

then change the declaration of `SecondClass.f` to `private` and recompile it, the VMs that we have tested do not reject `SometimesAccessPrivate` unless and until its constructor is passed a 0 as its second parameter, provoking an `ArithmeticException`. This means that illegal accesses which appear in unreachable code do not in practice make the code unverifiable, though the JVM specification allows verifiers to reject such code.

The verifier may also delay some type checks until runtime. For example, in a program that includes the statement:

```
ClassB b = new ClassA();
```
(35)

The VM may delay checking whether `ClassA` is assignable to `ClassB` until the first time the statement is executed, forcing it to load `ClassA` and `ClassB`. So again, a method which should in principle fail static verification may in fact run correctly.

We will discuss these checks in two stages, first examining which checks admit situations where the check succeeds using a pruned CFG but fails with the verifier's unpruned CFG. Then we will investigate how Soot's optimisations, using a pruned CFG, might actually create those situations, producing unverifiable code.

3.4.2 When might pruned CFGs make unverifiable code look verifiable

Assuming that it begins with verifiable code as input, for a bytecode-to-bytecode transformer to output unverifiable code as a result of pruning unrealizable exceptions from its CFGs, it must be the case that one of the checks listed in section 3.4.1 fails when values are flowed into an exception handler from every instruction it protects, but passes if they are flowed into the handler only from instructions that might actually throw the exception caught.

Stack values and height We can immediately rule out verification problems involving the state of the operand stack (checks 1 and 2, as well as 3 as regards stack locations), since when an exception is thrown, the operand stack is cleared before the thrown exception is pushed onto it. Thus, the stack always contains a single `Throwable` at the beginning of any exception handler, regardless of how many paths an analysis thinks there might be to that handler.

Conflicting definitions for locals Example 26 in Section 2.6 shows a method which fails verification because to the verifier it appears that a local might be read in the handler without ever having been defined, while with a pruned CFG it is clear that the local always receives a value before any exception caught by the handler can be raised. The same basic problem—that conflicting definitions for locals reach the handler with unpruned CFGs but not with pruned ones—can also manifest itself in verification failures reported as type mismatches, as inappropriate `protected` access, or as inappropriate access to an uninitialized object.

Schematically, the situation is this:

```
beginTryBlock:
          ⋮
     instruction i, the first which might throw a caught exception
          ⋮
     instruction j, the last which might throw a caught exception
          ⋮
endTryBlock:
     ⋮
handler:
     ⋮
     use of local l
     ⋮
```

If local l is undefined at the beginning of the `try` block but there is a definition of l which precedes instruction i, then verification fails because the verifier mistakenly thinks the handler might refer to an undefined local (check 4), as in example 26.

If l is defined before instruction i, but there are other definitions of l within the `try` block which precede instruction i or follow instruction j and whose types conflict with the definitions of l which are in effect between i and j, then verification will fail because of conflicting types (check 3), even if all the definitions of l which can really reach the handler are consistent.

If the handler accesses a `protected` member of an object stored in l and if, between i and j, l always contains an instance of a subclass of the current class (so the `protected` access within the handler is actually acceptable), but at some point within the `try` block before i or after j l is assigned an instance of a superclass of the current class, the verifier will complain about "`Bad access to protected data`" (check 5).

If a reference to an uninitialized object is assigned to l at some point within the `try` block after instruction j, or if such a reference is assigned to l and then the object is initialized at some point within the `try` block before instruction i, then the verifier will complain about access to an uninitialized object, even though the uninitialized object cannot really reach the handler (check 6; in fact, such a situation is extremely unlikely, as the following aside explains).

Similarly, if the schema represents an `<init>()` method and a reference to the object being initialized (`this`) is assigned to l at some point in the `try` block after instruction j but before the superclass's `<init>()` is invoked, or if such a reference is assigned to l and then the superclass 's `<init>()` invoked at a point within the `try` block but before instruction i, then the verifier will complain about inappropriate access to an uninitialized object even though the uninitialized object cannot really reach the handler (check 7).[27]

Aside about uninitialized objects So long as exception analysis remains strictly intraprocedural, it is very difficult to cook up a situation where the verifier would complain about an uninitialized object being accessed in the handler (check 6) without that complaint being justified.

A strictly intraprocedural analysis must assume that any method invocation, including `<init>()`, might throw any exception. So regardless of the type of exceptions it catches, if a handler's protected region includes a call to `<init>()`, there is a path to the handler from that call's predecessor, that is, at a point where the object is uninitialized. That means we cannot contrive a protected region where a local contains an uninitialized object at some point, but where the only possible exceptions occur after the object is initialized. Thus the only way to create a situation where the verifier is mistaken when it says a handler may access an uninitialized object is to exclude the `<init>()` from the protected area, and ensure that the last instruction which may throw an exception precedes the `new`.

This is not code anybody is likely to write. Nor can it ever result from compiling Java source: the `new` and the `<init>()` get generated from a single object instantiation in Java, which is either entirely within or entirely without any given `try` block, so all the instructions generated between the `new` and `<init>()` would also be either entirely within or entirely without any areas of protection.

At first blush, the converse situation seems more plausible, where unrealizable exceptions cause the verifier to think that an already initialized object might be initialized a second time by a handler even though, when considering only realizable exceptions, it is clear that only uninitialized objects can reach the handler. But this situation doesn't actually require separate consideration, since if only uninitialized objects can reach the handler and the handler accesses a local containing a reference to these uninitialized objects, then the code fails verification regardless of whether the handler might initialize the object a second time.

`jsr` restrictions The verifier includes a number of complicated restrictions on the possible uses of `jsr` return addresses and the permissible order in which subroutines may be called, as well as elaborate mechanisms to avoid merging the types of locals at the different points where the subroutine is called. Thankfully, Soot replaces `jsr` instructions with inlined copies of the called subroutine, so we can ignore the effect of pruning CFGs on verifying `jsr`s (check 8).

Falling off the code array Theoretically, the distinction between unrealizable and realizable exceptional paths could affect the verifier's check that execution does not fall off the end of the code array (check 9). If the code array ended with an improperly terminated handler (one whose last instruction was not a `return`, `throw`, or `goto`), but the method could not throw any exception caught by that method, then one could consider the method to be acceptable when only realizable execution paths are considered, even though it would fail verification. If Soot were to produce such improperly terminated handlers, though, we would prefer that they fail verification so we discovered the questionable code, so this issue will be ignored in the rest of this discussion.

[27]There is a wrinkle when it comes to the handling of uninitialized objects within an `<init>()` method. In principle, the situation should exactly parallel that of uninitialized objects in any other method: no objects can be stored into a local in protected code before the superclass's `<init>()` is invoked if the handler might access the local. In practice, Sun's verifier seems to have become more lenient as of Java 1.4.0: it allows a handler within an `<init>()` to actually access an uninitialized object, so long as the handler doesn't do anything with the object that would be illegal outside of the handler (that is, so long as it calls super's `<init>()` before doing anything other than assigning to fields declared in the current class).

Delayed linkage failures If a handler is completely unreachable because no exceptions it catches are thrown, then, from the perspective of a pruned CFG, an illegal reference to a method or instance (check 10) within that handler need not cause the method to fail verification, even though it would fail a purely static test. Since existing implementations of the verifier do not, in fact, check access restrictions on instructions before they are executed, verifiers in practice already use pruned CFGs for this test. In principle, verifiers are allowed to reject such methods, so we really should address this issue, but if anything in Soot is adding references to inaccessible members in unreachable code, then that part of Soot is probably broken, and we should welcome verification failures which uncover it.

In summary, then, there is a single mechanism whereby pruning unrealizable exceptions from control flow graphs might cause Soot to produce unverifiable, but otherwise correct, code: if more precise knowledge of exceptional control flow leads Soot to use local variables in such a way that the verifier thinks that the locals might contain inappropriate values when they are accessed within some handler.

3.4.3 How Soot can generate conflicting local definitions

Soot's `DeadAssignmentEliminator` provides the most straightforward example of how removing unrealizable paths from CFGs can cause Soot to generate unverifiable code. When `CompleteUnitGraph` includes only realizable exceptions, the `DeadAssignmentEliminator` might remove some local initializations which the verifier requires in order to ensure that all locals are defined before use, since the verifier takes unrealizable paths into consideration. If you compile the following Java

(36)

```
static int m(Object o) {
    int l1 = 0;
    try {
        l1 = 20;
        l1 = o.hashCode();
    } catch (NullPointerException e) {
        l1 = -1 * l1;
    }
    return l1;
}
```

and run "`soot -p jb.dae only-stack-locals:false`" using pruned `CompleteUnitGraph`s, Soot removes the first assignment to `l1` and the verifier complains "`Accessing value from uninitialized register 1`".

Without the "`only-stack-locals:false`" option, the `DeadAssignmentEliminator` only removes assignments to local variables which stand for stack locations in the input bytecode, in which case it cannot generate such verification failures on its own. The stack is cleared when an exception occurs, so in a handler, the only reads from locals that originally corresponded to stack locations are going to be reads of values that were pushed after the exception occurred. Thus there is no risk that pruning unrealizable paths to the handler can cause `DeadAssignmentEliminator` to remove an assignment to a stack local that is read within the handler. Since by default the `DeadAssignmentEliminator` option `only-stack-locals` is true, one might think all we need do is to document that setting the option to `false` can produce verification problems. One would be forgetting about the Jimple `LocalSplitter`.

Let's say our code follows this pattern, with no use of l between the first definition and the redefinition:

(37)

```
          ⋮
     definition of local l to some dummy value
          ⋮
startTry:
          ⋮
     redefinition of l
          ⋮
     instruction i, the first which might throw a caught exception
          ⋮
endTry:
          ⋮
handler:
          ⋮
     use of l
          ⋮
```

(The pattern is typical of situations where a local which communicates values from a `try` block to its `catch` handler has to be declared and initialized outside the `try` and `catch` in order to be in scope within both of them, even though it is not used outside of them.)

When unrealizable paths are eliminated from the CFG, the LocalSplitter sees that the first definition of l is not used anywhere, so it splits the redefinition and use of l into a separate local, l`#2`. If Soot's LocalPacker subsequently leaves l`#2` in a separate local (this could happen if the Packer finds some other variable, k, to combine l with, but k and l`#2` interfere so they cannot be packed), then the verifier will complain that l`#2` might be undefined when accessed in the handler, since it is too dumb to see that an exception cannot be thrown to the handler before the first assignment to l`#2`.

The LocalSplitter can combine with the Baf LocalPacker (which is not type sensitive) to cause verification failures due to type clashes. Add a use of l to example 37, so that the first definition may no longer be eliminated:

```
          ⋮                                                        (38)
     definition of local l to some dummy value
          ⋮
startTry:
          ⋮
     use of l
          ⋮
     redefinition of l
          ⋮
     instruction i, the first which might throw a caught exception
          ⋮
endTry:
        ⋮
handler:
        ⋮
    use of l
        ⋮
```

A pruned CFG will still cause the Local Splitter to split l into l and l`#2`, since it will still see that only the second definition of l reaches the handler. Now if the Baf `LocalPacker` finds some other variable which is dead in the handler to combine with l`#2`, say k, but k's type differs from l`#2`'s, then the verifier will complain that the type of l`#2`'s local could be incorrect.

Precisely this sort of interaction causes example 34 to fail verification even when `only-stack-local` is left `true`. Give soot no arguments at all, and the verifier's complaint is about register 2 instead of 1. Use "`-p bb.lp off`" to take the `LocalPacker` out of the picture and the `LocalSplitter` and `DeadAssignmentEliminator` can do the job alone, though the verifier complains about register 3 in that case.

3.4.4 Repairing unverifiable code

The interactions between the `DeadAssignmentEliminator`, `LocalSplitter`, and `LocalPacker`s which make the bytecode verifier believe that inappropriate values might be stored in locals accessed by a handler are too intricate to address the problem by adding simple checks to catch specific instances where an optimisation should not be performed. On the other hand, having the `DeadAssignmentEliminator`, `LocalSplitter`, and `LocalPacker`s use unpruned control flow graphs, so their worldview matches that of the bytecode verifier, is most unattractive; the very fact that pruned CFGs change local use sufficiently to produce unverifiable code is evidence that pruning the exceptions before assigning locals may actually accomplish something!

So we have adopted a different approach: we make the code verifiable by shrinking the protected areas to reduce the possibility that conflicting local definitions will appear to reach the handlers, even from the perspective of the verifier's unpruned CFGs. We have added a new initial phase, the `TrapTightener`, to the "`JimpleBody` pack" which Soot uses to turn input bytecode into Jimple. The `TrapTightener` simply adjusts the area covered by each exception handler to begin with the first unit that might actually throw a trapped exception, and to end just after the last such unit. So long as our exception analysis is correct, this does not change the semantics of the program. It does, though, ensure that the verifier will not consider any local definitions to flow into the handler which occur before the first definition that Soot considers to flow to the handler, or after the last definition that Soot considers to flow to the handler.

This does not guarantee that the method is verifiable. There could be a control flow path that enters the trapped region after the first excepting instruction, and leaves it before the last one, as a result of which the verifier might mistakenly believe that some local accessed within the handler might be undefined (though such a method could not be the result of compiling Java source). More plausible would be a situation like the following, where i and j are the only instructions which can throw an exception caught by the handler:

```
        ⋮                                                    (39)
startTry:
        ⋮
    l defined as type t
```

```
        ⋮
    instruction $i$, which might throw a caught exception
        ⋮
    redefinition of $l$ as conflicting type $u$
        ⋮
    redefinition of $l$ as type $t$
        ⋮
    instruction $j$, which might throw a caught exception
        ⋮
endTry:
handler:
        ⋮
    use of $l$ as type $t$
        ⋮
```

Even after the `TrapTightener` moved `startTry:` to just before instruction i and `endTry:` to just after instruction j, this code would not pass verification, since the verifier would think that the handler could try to use a type u value stored in l as a type t value, even though that could not really happen.[28]

To deal with such a situation, we would need a `TrapSplitter`, rather than a `TrapTightener`. The `TrapSplitter` would replace an existing exception table entry with multiple entries, each catching the same `Throwable` type with the same handler, but protecting only a portion of the original trapped region over which the local definitions are consistent. The simplest way to accomplish this would be by creating a new trap for every unit in the original protected area which can actually throw the caught exception, with each new protected area covering only the single unit.

Inflating the exception table in such a manner, though, would certainly cost memory and disk space, and it could also cost execution time, since "stack cutting" implementations of exception handling perform more housekeeping when there are more entries in the exception table [9]. In our benchmarks the `TrapTightener` suffices to avoid producing unverifiable code, so we have yet to implement a `TrapSplitter`.

In the future, we would like to create a dataflow analysis that would mimic the bytecode verifier, so that we could use the `TrapTightener` only for classes which are not verifiable, and resort to a `TrapSplitter` if the class remains unverifiable even after trap tightening. The ersatz verifier should be a relatively simple variation on `SimpleLocalDefs`, Soot's analysis for finding local varible definitions. The detector would differ from `SimpleLocalDefs` in that it would use a `CompleteUnitGraph` built with `PedanticThrowAnalysis`, that is, an unpruned CFG. It would consider all locals to be initially defined to a special *uninitialized* value at the beginning of the method, and it would not kill the *uninitialized* definition of a local assigned the result of a `new` instruction until after the new object is initialized.

3.5 Validation

Since handlers are comparatively rare in benchmark programs, and thrown exceptions even rarer, we cannot assume that our implementation of pruned control flow graphs is correct just because it successfully

[28]The Baf `LocalPacker` has the potential to create such code, though we have yet to figure out how to coax it into doing so.

transforms a set of benchmarks into class files which continue to produce the same results that they used to. Errors in the generated graphs could easily go undetected.[29]

This section outlines some steps which have been taken to ensure the correctness of the new CFG implementation. As a consequence of looming deadlines, it also outlines steps which should be taken to ensure that the new implementation is correct, but have not as yet.

The `ThrowableSet` class is the most thoroughly tested component in our modifications, as a result of a set of test cases implemented using the `JUnit` framework, which exercises every member function and includes what amounts to a re-implementation of `ThrowableSet`'s memoization code.

`UnitThrowAnalysis`'s handling of the Jimple IR and its derivatives Shimple and Grimp has been tested against a set of unit tests which include every Jimple and Grimp statement, though of course not every conceivable combination of values (since expressions are defined recursively, the set of possible combinations is infinite). Testing of the Baf representation has been less extensive, consisting of an unautomated check of the `CompleteUnitGraph` produced for a set of classes originally written to test VM instrumentation.[30] It is easy to conceive of regression tests for `UnitThrowAnalysis` which can be checked automatically, but tedious to write them by hand. We hope to modify the XML source used to produce Appendix A to generate the tests automatically and guarantee that they are consistent with the appendix.

The least tested aspect of the new CFGs is the algorithm used to link possible exceptions to their catchers, and to build control flow edges from the exception destination edges. Ideally, to automate such tests we would like to write a tool which could parse annotations added as comments to a Jimple file which indicated where each unit's exceptions should be thrown and what its predecessors and successors should be. Then we could annotate some test files with the graph edges we expect to find, and use these files in automated regression tests against the `CompleteUnitGraph` actually produced.

An indirect test of our modified graphs, though, is provided by our tests of the unmodifed ones. As a result of our reorganization of the `UnitGraph` and `BlockGraph` hierarchies, we re-implemented all of the CFG classes, including those whose results should be identical to those in Soot 2.1.0. To test that the unpruned CFG classes continue to produce the same results as they did before being modified, we wrote a class to compare graphs, and a rudimentary user-defined class loader, which allows loading the old and new graph classes at the same time.

Classes loaded with the user-defined loader continue to implement the `DirectedGraph` interface, which is sufficient to determine whether two graphs are identical. The restrictions on intermixing classes defined by different loaders, though, hampers the detailed examination of differences between two graphs which are not identical. `ClassicCompleteUnitGraph` and `ClassicCompleteBlockGraph` are supposed to have exactly the same set of nodes and edges as would be produced by the `CompleteUnitGraph` and `CompleteBlockGraph` classes of Soot 2.1.0. To validate the new graph classes, then, requires ensuring that the `Classic` graphs are identical to corresponding graphs built with the old classes loaded through the user-defined class loader, and then comparing the `Classic` graphs with the corresponding new `CompleteUnitGraph` or `CompleteBlockGraph`, to confirm that they differ only in ways that they are supposed to differ.

All comparisons performed to date show no changes between the unpruned graphs produced by the old and

[29]Indeed, in the process of this project we have found some pre-existing errors in the handling of exceptions which have long gone undetected, or at least unrepaired: for instance, Soot does not deal properly with a protected area which extends to the last instruction in a method's code array.

[30]The classes contain one short method for each JVM opcode, which contains the corresponding instruction and as little else as possible.

	check	db	javac	jess	mpegaudio	sablecc-w	soot-j	j2sdk1.4.2
Classes	41	30	206	175	78	326	621	8100
Methods	387	314	1478	953	579	2193	3091	71307
Units	8726	7293	31069	17488	19585	26911	42107	1252582
Trapped Units	928 (10.6%)	876 (12.0%)	2549 (8.2%)	1158 (6.6%)	851 (4.3%)	2404 (8.9%)	621 (1.5%)	93314 (7.4%)
Uncaught Trapped Units	354 (4.1%)	315 (4.3%)	1305 (4.2%)	397 (2.3%)	306 (1.6%)	1454 (5.4%)	383 (0.9%)	22372 (1.8%)
Methods by number of exception handlers:								
0	335 (86.6%)	273 (86.9%)	1380 (93.4%)	894 (93.8%)	541 (93.4%)	2159 (98.4%)	3078 (99.6%)	64714 (90.8%)
1	40 (10.3%)	36 (11.5%)	72 (4.9%)	48 (5.0%)	33 (5.7%)	12 (0.5%)	8 (0.3%)	3019 (4.2%)
2	8 (2.1%)	3 (1.0%)	19 (1.3%)	7 (0.7%)	4 (0.7%)	2 (0.1%)	2 (0.1%)	2188 (3.1%)
3	1 (0.3%)	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	18 (0.8%)	0 (0.0%)	558 (0.8%)
4–9	2 (0.5%)	1 (0.3%)	5 (0.3%)	3 (0.3%)	0 (0.0%)	2 (0.1%)	3 (0.1%)	711 (1.0%)
10+	1 (0.3%)	1 (0.3%)	1 (0.1%)	1 (0.1%)	1 (0.2%)	0 (0.0%)	0 (0.0%)	117 (0.2%)
Total handlers	109	84	174	112	79	79	26	14526
Methods by number of explicit `athrows`:								
0	368 (95.1%)	304 (96.8%)	1384 (93.6%)	881 (92.4%)	554 (95.7%)	2089 (95.3%)	2911 (94.2%)	60936 (85.5%)
1	16 (4.1%)	8 (2.5%)	79 (5.3%)	55 (5.8%)	22 (3.8%)	69 (3.1%)	158 (5.1%)	7839 (11.0%)
2	2 (0.5%)	1 (0.3%)	9 (0.6%)	12 (1.3%)	2 (0.3%)	33 (1.5%)	18 (0.6%)	1389 (1.9%)
3	0 (0.0%)	0 (0.0%)	1 (0.1%)	2 (0.2%)	0 (0.0%)	1 (0.0%)	2 (0.1%)	518 (0.7%)
4–9	1 (0.3%)	1 (0.3%)	4 (0.3%)	3 (0.3%)	1 (0.2%)	1 (0.0%)	2 (0.1%)	590 (0.8%)
10+	0 (0.0%)	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	35 (0.0%)
Total `athrows`	24	14	127	97	30	143	208	15624
Partition of units by exceptions thrown (sums to 100 %):								
any `Throwable`	3239 (37.1%)	2885 (39.6%)	8929 (28.7%)	6269 (35.8%)	3622 (18.5%)	7268 (27.0%)	14439 (34.3%)	291033 (23.2%)
Only *async*	2371 (27.2%)	1760 (24.1%)	9629 (31.0%)	4913 (28.1%)	4069 (20.8%)	7045 (26.2%)	12434 (29.5%)	408224 (32.6%)
Only *linkage*	1069 (12.3%)	800 (11.0%)	2189 (7.0%)	1478 (8.5%)	1295 (6.6%)	1502 (5.6%)	3231 (7.7%)	92043 (7.3%)
async + some `Exception`	1033 (11.8%)	783 (10.7%)	3987 (12.8%)	2018 (11.5%)	8339 (42.6%)	6352 (23.6%)	4601 (10.9%)	252815 (20.2%)
linkage + some `Exception`	1014 (11.6%)	1065 (14.6%)	6335 (20.4%)	2810 (16.1%)	2260 (11.5%)	4744 (17.6%)	7402 (17.6%)	208467 (16.6%)
Number of units which might throw various `RuntimeException` and `Errors` (rows are not exclusive; units throwing “any `Throwable`” do not contribute to totals for individual exceptions):								
async	8726 (100.0%)	7293 (100.0%)	31069 (100.0%)	17488 (100.0%)	19585 (100.0%)	26911 (100.0%)	42107 (100.0%)	1252582 (100.0%)
linkage	2083 (23.9%)	1865 (25.6%)	8524 (27.4%)	4288 (24.5%)	3555 (18.2%)	6246 (23.2%)	10633 (25.3%)	300510 (24.0%)
Arithmetic	18 (0.2%)	4 (0.1%)	9 (0.0%)	6 (0.0%)	4 (0.0%)	0 (0.0%)	6 (0.0%)	442 (0.0%)
ArrayIndexOutOfBounds	306 (3.5%)	284 (3.9%)	1215 (3.9%)	621 (3.6%)	6599 (33.7%)	3129 (11.6%)	751 (1.8%)	138468 (11.1%)
ArrayStore	190 (2.2%)	185 (2.5%)	604 (1.9%)	262 (1.5%)	217 (1.1%)	249 (0.9%)	115 (0.3%)	73104 (5.8%)
ClassCast	55 (0.6%)	65 (0.9%)	378 (1.2%)	127 (0.7%)	56 (0.3%)	551 (2.0%)	1303 (3.1%)	14913 (1.2%)
IllegalMonitorState	648 (7.4%)	437 (6.0%)	2539 (8.2%)	1287 (7.4%)	775 (4.0%)	2433 (9.0%)	3524 (8.4%)	102028 (8.1%)
NegativeArraySize	49 (0.6%)	30 (0.4%)	109 (0.4%)	55 (0.3%)	946 (4.8%)	741 (2.8%)	104 (0.2%)	34456 (2.8%)
NullPointer	1295 (14.8%)	1325 (18.2%)	7415 (23.9%)	3437 (19.7%)	8847 (45.2%)	7363 (27.4%)	7055 (16.8%)	324473 (25.9%)
Number of handlers, by classes caught								
`Throwable`	39 (35.8%)	38 (45.2%)	42 (24.1%)	49 (43.8%)	38 (48.1%)	2 (2.5%)	0 (0.0%)	7102 (48.9%)
`Exception`	16 (14.7%)	8 (9.5%)	13 (7.5%)	8 (7.1%)	11 (13.9%)	9 (11.4%)	4 (15.4%)	1166 (8.0%)
`RuntimeException`	2 (1.8%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	56 (0.4%)
some `Error`	4 (3.7%)	3 (3.6%)	5 (2.9%)	3 (2.7%)	3 (3.8%)	0 (0.0%)	0 (0.0%)	73 (0.5%)
some implicit `RuntimeException`	22 (20.2%)	0 (0.0%)	3 (1.7%)	3 (2.7%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	207 (1.4%)
some explicit `Exception`	26 (23.9%)	35 (41.7%)	111 (63.8%)	49 (43.8%)	27 (34.2%)	68 (86.1%)	22 (84.6%)	5922 (40.8%)

Table 1: Benchmark characteristics

new classes, and only expected changes between the pruned classes. Since the mechanisms used to turn `UnitGraph`s into the corresponding `BlockGraph`s are identical in the pruned and unpruned classes, this provides some reassurance that the pruned classes are implemented correctly.

3.6 Experimental Results

To date, this project has focused on the correctness of the pruned control flow graphs, rather than on the performance of the analysis or of the resulting code. This section nevertheless provides a rudimentary comparison of the times required to analyze a set of benchmarks with pruned and unpruned CFGs and of the execution times of the output code.

Table 1 characterizes the benchmarks used.

Check, db, jess, mpegaudio, and javac are all components of SPEC JVM98 as packaged for the Ashes benchmark suite. Check is of little interest from the perspective of measuring performance of output code, but it does test the correctness of some aspects of JVM implementations, including their exception handling. db queries a memory-resident database, jess is an expert system, and mpegaudio decompresses audio files. They have no particular distinguishing characteristics, other than mpegaudio's very large proportion of array accesses. javac is included because it often seems to be an outlier in reports that use the SPEC JVM98

Soot version and options	check	db	javac	jess	mpegaudio	sablecc-w	soot-j	j2sdk1.4.2
soot-1143	21.33	19.968	40.876	29.794	42.094	46.455	55.684	47.236
soot-x	22.11 (103.7%)	20.971 (105.0%)	43.899 (107.4%)	30.684 (103.0%)	42.09 (100.0%)	47.451 (102.1%)	56.158 (100.9%)	50.925 (107.8%)
soot-1143 -O	22.33	20.711	45.013	32.009	45.725	50.846	61.916	50.695
soot-x -O	23.356 (104.6%)	22.136 (106.9%)	48.703 (108.2%)	33.892 (105.9%)	44.883 (98.2%)	51.505 (101.3%)	64.299 (103.8%)	54.356 (107.2%)
soot-1143 -O -p jop.cse on	22.808	21.318	47.064	33.721	47.997	53.745	65.461	52.426
soot-x -O -p jop.cse on	—	22.62 (106.1%)	50.336 (107.0%)	35.153 (104.2%)	47.916 (99.8%)	54.535 (101.5%)	81.211 (124.1%)	56.477 (107.7%)
soot-1143 –via-grimp	20.768	19.479	40.042	29.138	39.646	43.193	67.084	45.594
soot-x –via-grimp	21.489 (103.5%)	20.658 (106.1%)	41.843 (104.5%)	30.031 (103.1%)	39.185 (98.8%)	44.043 (102.0%)	69.93 (104.2%)	48.874 (107.2%)
soot-1143 –via-grimp -O	21.953	20.389	43.984	31.782	41.266	48.315	75.724	49.225
soot-x –via-grimp -O	22.909 (104.4%)	21.653 (106.2%)	45.977 (104.5%)	32.49 (102.2%)	41.853 (101.4%)	49.114 (101.7%)	80.214 (105.9%)	52.836 (107.3%)
soot-1143 –via-grimp -O -p jop.cse on	22.59	20.97	45.954	32.461	44.907	50.216	80.063	50.875
soot-x –via-grimp -O -p jop.cse on	—	22.309 (106.4%)	48.432 (105.4%)	33.592 (103.5%)	46.026 (102.5%)	51.353 (102.3%)	80.489 (100.5%)	55.056 (108.2%)

Table 2: Soot's execution time for analyzing each benchmark, with and without pruned CFGs, in seconds. Results are from a single run only.

	check	db	javac	jess	mpegaudio	sablecc-w	soot-j	j2sdk1.4.2
`ThrowableSet`s thrown	17	15	26	17	17	22	37	372
`ThrowableSet`s reg'd	65	51	91	66	55	49	69	1398
Additions	280633	239419	806141	368663	343711	521105	492993	29087379
of `RefType`	134024 (47.8%)	106515 (44.5%)	361068 (44.8%)	148768 (40.4%)	127792 (37.2%)	214322 (41.1%)	137795 (28.0%)	13780453 (47.4%)
of `AnySubType`	17192 (6.1%)	17250 (7.2%)	50991 (6.3%)	22028 (6.0%)	16302 (4.7%)	32658 (6.3%)	18283 (3.7%)	1557810 (5.4%)
of `ThrowableSet`	129417 (46.1%)	115654 (48.3%)	394082 (48.9%)	197867 (53.7%)	199617 (58.1%)	274125 (52.6%)	336915 (68.3%)	13749116 (47.3%)
Adds from map	24 (0.0%)	0 (0.0%)	0 (0.0%)	8 (0.0%)	18 (0.0%)	16 (0.0%)	0 (0.0%)	124 (0.0%)
Adds from memo	280506 (100.0%)	239333 (100.0%)	806005 (100.0%)	368549 (100.0%)	343600 (100.0%)	521006 (100.0%)	492888 (100.0%)	29085602 (100.0%)
Adds needing search	103 (0.0%)	86 (0.0%)	136 (0.0%)	106 (0.0%)	93 (0.0%)	83 (0.0%)	105 (0.0%)	1653 (0.0%)
`catchableAs` queries	210	180	345	210	210	285	510	5530
from map	170 (81.0%)	140 (77.8%)	302 (87.5%)	170 (81.0%)	167 (79.5%)	245 (86.0%)	470 (92.2%)	5471 (98.9%)
needing search	40 (19.0%)	40 (22.2%)	43 (12.5%)	40 (19.0%)	43 (20.5%)	40 (14.0%)	40 (7.8%)	59 (1.1%)

Table 3: Results from `ThrowableSet` instrumentation

benchmarks.

Sablecc-w and soot-j are components of the Ashes benchmark suite. Sablecc-w is a parser generator which is itself very object-oriented and which produces object-oriented code. Soot-j is an early version of soot and is also very object-oriented.

j2sdk1.4.2 is the `rt.jar` file included in Sun's 1.4.2 release of Java. It is not, strictly speaking, a benchmark since it cannot be run on its own. The library is, nevertheless, included for two reasons. First, since the Java libraries are such an important part of most Java applications, Soot's treatment of the library is a crucial factor in its overall performance. Second, we expected that exception handling would play a more prominent role in library code than in application code, since the exception mechanism exists precisely to help library functions communicate problems to their callers. The proportion of explicit `athrow` instructions is noticeably higher in the Java library than in any of the benchmarks, with roughly 10% more methods including an `athrow` instruction. The library also contains a higher proportion of exception handlers and

Soot version and options	check	db	javac	jess	mpegaudio	sablecc-w	soot-j
soot-1143	.23	25.11	8.88	3.95	6.34	4.60	10.82
soot-x	.28 (121.7%)	24.74 (98.5%)	8.87 (99.9%)	3.95 (100.0%)	6.30 (99.4%)	4.66 (101.3%)	10.84 (100.2%)
soot-1143 -O	.22	25.29	8.82	3.96	6.22	4.64	10.81
soot-x -O	.28 (127.3%)	25.79 (102.0%)	8.84 (100.2%)	3.98 (100.5%)	6.23 (100.2%)	4.60 (99.1%)	10.86 (100.5%)
soot-1143 -O -p jop.cse on	.28	25.29	8.82	4.00	6.28	4.60	10.87
soot-x -O -p jop.cse on	—	24.83 (98.2%)	8.84 (100.2%)	4.00 (100.0%)	6.28 (100.0%)	4.61 (100.2%)	10.84 (99.7%)
soot-1143 –via-grimp	.28	25.32	8.84	3.95	6.16	4.66	10.88
soot-x –via-grimp	.28 (100.0%)	25.94 (102.4%)	8.87 (100.3%)	4.00 (101.3%)	6.16 (100.0%)	4.65 (99.8%)	10.90 (100.2%)
soot-1143 –via-grimp -O	.28	25.09	8.87	3.96	6.24	4.66	10.76
soot-x –via-grimp -O	.28 (100.0%)	24.98 (99.6%)	8.89 (100.2%)	4.00 (101.0%)	6.21 (99.5%)	4.63 (99.4%)	10.80 (100.4%)
soot-1143 –via-grimp -O -p jop.cse on	.28	24.65	8.91	4.01	6.29	4.60	10.84
soot-x –via-grimp -O -p jop.cse on	—	24.72 (100.3%)	8.84 (99.2%)	4.00 (99.8%)	6.28 (99.8%)	4.60 (100.0%)	10.88 (100.4%)

Table 4: Execution time, in seconds, for benchmarks after transformation by Soot. Results are the average of 10 runs.

trapped code. For example, j2sdk1.4.2 contains roughly 30 times as many instructions as soot-j, but 150 times as many handlers.

The numbers reported in table 1 were gathered using Soot's new pruned `CompleteUnitGraph`. This is unsatisfactory since it introduces a Heisenberg effect: what we measure is being influenced by the means used to measure it. We hope to reimplement a portion of the exceptional control flow analysis at the level of bytecode instructions, instead of Soot units, so that we can characterize the benchmarks independently of the new CFG. That would allow comparisons of the exceptional control flow in class files before and after transformation by soot.

The most notable feature revealed by the benchmark table is the paucity of exception handlers, and thus the limited scope for improving precision by pruning CFGs. The "Trapped Units" row shows how many units in each benchmark lie statically within the scope of an exception handler. The "Uncaught Trapped Units"row shows how many of the trapped units throw no exception that their handlers can catch. Thus "Uncaught Trapped Units" indicates roughly how many fewer edges will be in the benchmark's CFGs as a result of pruning unrealizable exceptions. Uncaught trapped units account for only 1–6% of the benchmarks' units.

The scope for improving the precision of analyses is also limited by the large number of units which may throw "Any `Throwable`" and the large proportion of exception handlers which catch `Throwable`. The bulk of the units throwing "Any `Throwable`" are method invocations (recall that our analysis is strictly intraprocedural and must assume that any method may throw any exception), with the remainder consisting of rethrown exceptions which were caught in a `catch` parameter of type `Throwable`. The large proportion of `catch` parameters of type `Throwable` is almost certainly due to the fact that Java's `finally` clauses and `synchronized` blocks are implemented using handlers which catch and rethrow all exceptions. Note that sablecc-w and soot-j, which are single-threaded, declare almost no `Throwable catch` parameters. By far the second greatest proportion of `catch` parameters catch some explicit exception which signals a library error. There are relatively few handlers for specific implicit exceptions.

Table 2 shows the time required for Soot to analyze the benchmarks with a few different command line options. All measurements in tables 2 and 4 were performed with Sun's Java 1.4.2 for Linux, on a PC running Linux 2.4.20 on two 1.6 gigahertz Athlon processors and 2 gigabytes of memory. "`soot-1143`" refers to revision 1143 of the trunk of Soot's `subversion` repository; while "`soot-x`" refers to the "`soot-2-exceptional`" branch in the repository, which differs from revision 1143 by the addition of pruned control flow graphs. The blank entries where pruned CFGs are combined with "`-p jop.cse on`" result from the inability of Soot's `FastAvailableExpressions` analysis to deal with an unreachable handler whose exception is never raised. This is an example of Soot's inconsistent treatment of CFG entry points.

A priori, it is not clear whether to expect longer or shorter analysis times with pruned CFGs. On the one hand, constructing the new pruned `CompleteUnitGraph` for a method which includes exception handlers requires more computation than the old `CompleteUnitGraph`, and the resulting data structure is larger (the addition of exception destination edges more than offsets the lower number of exceptional control flow edges). On the other hand, pruned graphs provide follow-on analyses with fewer edges to analyze. The table indicates that analysis times are slightly higher with the pruned graphs.[31] The difference is small, but since only 10% of methods include handlers, a small difference is telling. On the other hand, a number of simple improvements to the `CompleteUnitGraph` constructor, described in Section 5, promise to reduce the time required to prune graphs.

[31]The wide discrepancy in analysis times for soot-j with "`-O -p jop.cse on`" is likely the result of background activity during the single analysis run; note that there is no similar discrepancy for "`--via-grimp -O -p jop.cse on`", which differs only in the stage which generates bytecode from the Jimple intermediate representation.

Table 3 documents the one unequivocal success of this project: the implementation of `ThrowableSet`. The table reports the operations on `ThrowableSets` required to produce table 1. Even j2sdk1.4.2's 1,252,582 units throw only 372 different sets of exceptions. It is disappointing that we generate and retain 1398 different `ThrowableSet` objects to represent these 372 sets (a result of memoizing each intermediate result in the construction of `ThrowableSets`), but doing so saves the garbage collector from having to collect over 1.2 million sets. More importantly, memoization is a clear win: virtually all add operations reuse cached results. Since the current algorithm for generating `CompleteUnitGraph` does not call `catchableAs()`, on the other hand, the high proportion of `catchableAs` queries which are satisfied from the `Throwable-Set`'s map of `RefLikeType` is meaningless, an artifact of the algorithm used to produce the "Partition of units by exception thrown".

Table 4 shows the execution times of the class files produced by the analyses reported in table 2. The execution times of classes produced with pruned control flow graphs do not differ appreciably from those produced with unpruned graphs (the superficially large percentage discrepancy for two rows of the check benchmark is due to the briefness of the periods being compared). This is slightly disappointing, given that there is no reason for pruned graphs to lead Soot to produce slower code, but may reflect the scarcity of exception handlers in the benchmarks being measured. We need to perform detailed comparisons of the class files produced with and without pruned CFGs, but have yet to create the tools necessary to do so.

4 Background and Related Work

The Jikes RVM project has produced particularly useful studies of Java exception handling. Choi et. al. [2] describe a representation of control flow which bundles together the implicitly thrown exceptions which might prematurely exit what would otherwise be a basic block, together with modifications to standard optimisations which accommodate these "factored control flow graphs". Chambers et. al. [1] provide a straightforward way to represent the restrictions imposed by precise exceptions as data dependences. Gupta et. al. [5] describe static and dynamic analyses which track the variables remaining live within exception handlers. This permits dependences to be relaxed between potentially excepting instructions and other instructions which do not affect any values used within the exception handlers, though the complex analyses enabling these optimisations have not been incorporated into any public releases of the Jikes RVM.

Like the Jikes RVM, Marmot, an ahead-of-time Java compiler developed at Microsoft Research [3], also factors together all exceptions which might occur during the execution of a basic block. This provides longer basic blocks for analysis than would be available if excepting instructions terminated the blocks, but requires that flow analyses—especially backwards analyses—treat exception edges specially.

Sinha and Harrold [10] provide algorithms for interprocedural analyses that link explicit `throw` statements to the `catch` clauses to which they may pass control. These algorithms are unlikely to be directly applicable to production systems, though, where implicit exceptions predominate and where compile-time analyses frequently lack access to all classes that might execute at run-time.

Stevens [11] provides an empirical study of the frequency of different implicit exceptions, showing the prevalence of potential `NullPointerExceptions` and potential errors due to linking and loading new classes (which we will call "linkage errors"). He provides methods to statically determine when such exceptions cannot occur, but the techniques for linkage errors require a closed world assumption that may only be justified in an embedded system.[32]

[32]And not always in embedded systems, if unattributable gossip is allowed as evidence. At PLDI 2002 I spoke with a developer working on a JVM intended for use within automobiles, who told me that dynamic class loading was a requirement in his firm's

5 Conclusions and Loose Ends

This project suggests that one cannot justify pruning unrealizable exceptions from control flow graphs on the basis of producing better code from the more precise graphs. Static statistics from benchmarks show that exception handlers are relatively rare and a large proportion of the handlers that exist catch all `Throwable`s, meaning that there will be edges to them from every unit they cover regardless of whether one tries to prune unrealizable exceptions. Our experimental data on execution times is scanty, but shows no benefit from pruned CFGs. Performance benefits seem not to justify the complications of dealing with such issues as ensuring that output code is verifiable, or determining whether exception edges should originate from the excepting unit's predecessors instead of, or in addition to, the unit itself.

The ability to report on exceptional control flow remains valuable for aiding program understanding. Pruning exceptional control flow may also be worthwhile if it simplifies other analyses. The production of SSA form, for example, is complicated by the large number of control flow paths to exception handlers. Indeed, we observed that in some micro benchmarks pruned CFGs reduced the tendency of Soot's "–via-shimple" option to produce bloated code for `try` blocks with multiple `catch` clauses. Unfortunately, we could not include "–via-shimple" in our experimental results because Shimple currently cannot deal with graphs with multiple entry points, which occur in the pruned CFGs for several of our benchmarks.

Several improvements remain to be made to this implementation of pruned control flow graphs:

- Adding exception destination edges to `CompleteBlockGraph` as well as `CompleteUnitGraph`.
- Distinguishing between those instructions which may cause a new class to be loaded but will not cause it to be initialized and those instructions which may cause a new class to be initialized. This will allow some units which are now reported to throw any `Error` to be reported as throwing only `LinkageError`s instead.
- Adding control flow edges only from explicit `throw` units, and not from the predecessors of those units.
- Reducing graph construction time by performing exceptional control flow analysis only for trapped units, rather than all units in methods which contain one or more traps.
- Reducing graph construction time by modifying clients which require both a `CompleteUnitGraph` and some `BlockGraph` to pass their `CompleteUnitGraph` as the basis for constructing the `BlockGraph`, so the `BlockGraph` constructor will not waste time building a duplicate `UnitGraph`.
- Writing tools to perform basic exceptional control flow analysis at a bytecode level so that the results of running Soot with and without pruned CFGs may be compared.
- Adding a `TrapSplitter` and a test for methods which may be unverifiable, to ensure Soot does not produce unverifiable code, while minimizing modifications to exception tables.
- Adding the ability to remove elements from `ThrowableSet`s, to streamline the algorithm for producing `CompleteUnitGraph` and to provide more accurate information to human readers of exception destination edges.

system. He assured me that the JVM would not be involved in the safety-critical systems actually controlling the car, but only in ancillary facilities available for distracting the driver.

- Performing an inventory of all CFG client analyses, to confirm that they do not make assumptions violated by the pruned graphs.
- Automating the creation of unit tests from Appendix A.
- Producing regression tests for `CompleteUnitGraph` which can be verified automatically.

A Who Throws What

Table 5: Implicit exceptions which instructions may throw

Bytecode	**Exceptions**	**Baf**	**Jimple**
`aconst_null`	*async*	`push null`	`null`
`iconst_m1`	*async*	`push -1`	`-1`
`iconst_0`	*async*	`push 0`	`0`
`iconst_1`	*async*	`push 1`	`1`
`iconst_2`	*async*	`push 2`	`2`
`iconst_3`	*async*	`push 3`	`3`
`iconst_4`	*async*	`push 4`	`4`
`iconst_5`	*async*	`push 5`	`5`
`lconst_0`	*async*	`push 0L`	`0L`
`lconst_1`	*async*	`push 1L`	`1L`
`fconst_0`	*async*	`push 0.0F`	`0.0F`
`fconst_1`	*async*	`push 1.0F`	`1.0F`
`fconst_2`	*async*	`push 2.0F`	`2.0F`
`dconst_0`	*async*	`push 0.0`	`0.0`
`dconst_1`	*async*	`push 1.0`	`1.0`
`bipush`	*async*	`push` *int constant*	*int constant*
`sipush`	*async*	`push` *int constant*	*int constant*
`ldc`	*async*	`push` *int|float|string constant*	*int|float|string constant*
`ldc_w`	*async*	`push` *int|float|string constant*	*int|float|string constant*
`ldc2_w`	*async*	`push` *long|double constant*	*long|double constant*
`iload`	*async*	`load.[bcsi]` *word local*	*int32 local on rhs*
`lload`	*async*	`load.l` *dword local*	*long local on rhs*
`fload`	*async*	`load.f` *word local*	*float local on rhs*
`dload`	*async*	`load.d` *dword local*	*double local on rhs*
`aload`	*async*	`load.r` *word local*	*ref local on rhs*
`iload_0`	*async*	`load.[bcsi]` *word local*	*int32 local on rhs*
`iload_1`	*async*	`load.[bcsi]` *word local*	*int32 local on rhs*
`iload_2`	*async*	`load.[bcsi]` *word local*	*int32 local on rhs*
`iload_3`	*async*	`load.[bcsi]` *word local*	*int32 local on rhs*
`lload_0`	*async*	`load.l` *dword local*	*long local on rhs*
`lload_1`	*async*	`load.l` *dword local*	*long local on rhs*
`lload_2`	*async*	`load.l` *dword local*	*long local on rhs*
`lload_3`	*async*	`load.l` *dword local*	*long local on rhs*
`fload_0`	*async*	`load.f` *word local*	*float local on rhs*
`fload_1`	*async*	`load.f` *word local*	*float local on rhs*
`fload_2`	*async*	`load.f` *word local*	*float local on rhs*
`fload_3`	*async*	`load.f` *word local*	*float local on rhs*
`dload_0`	*async*	`load.d` *dword local*	*double local on rhs*
`dload_1`	*async*	`load.d` *dword local*	*double local on rhs*
`dload_2`	*async*	`load.d` *dword local*	*double local on rhs*
`dload_3`	*async*	`load.d` *dword local*	*double local on rhs*
`aload_0`	*async*	`load.r` *word local*	*ref local on rhs*
`aload_1`	*async*	`load.r` *word local*	*ref local on rhs*
`aload_2`	*async*	`load.r` *word local*	*ref local on rhs*
`aload_3`	*async*	`load.r` *word local*	*ref local on rhs*

iaload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.i	*int local*[*imm*] *on rhs*
laload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.l	*long local*[*imm*] *on rhs*
faload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.f	*float local*[*imm*] *on rhs*
daload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.d	*double local*[*imm*] *on rhs*
aaload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.r	*ref local*[*imm*] *on rhs*
baload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.b	*boolean*|*byte local*[*imm*] *on rhs*
caload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.c	*char local*[*imm*] *on rhs*
saload	*async*, NullPointer, ArrayIndexOutOfBounds	arrayread.s	*short local*[*imm*] *on rhs*
istore	*async*	store.[bcsi] *word local*	*int32 local on lhs*
lstore	*async*	store.l *dword local*	*long local on lhs*
fstore	*async*	store.f *word local*	*float local on lhs*
dstore	*async*	store.d *dword local*	*double local on lhs*
astore	*async*	store.r *word local*	*ref local on lhs*
istore_0	*async*	store.[bcsi] *word local*	*int32 local on lhs*
istore_1	*async*	store.[bcsi] *word local*	*int32 local on lhs*
istore_2	*async*	store.[bcsi] *word local*	*int32 local on lhs*
istore_3	*async*	store.[bcsi] *word local*	*int32 local on lhs*
lstore_0	*async*	store.l *dword local*	*long local on lhs*
lstore_1	*async*	store.l *dword local*	*long local on lhs*
lstore_2	*async*	store.l *dword local*	*long local on lhs*
lstore_3	*async*	store.l *dword local*	*long local on lhs*
fstore_0	*async*	store.f *word local*	*float local on lhs*
fstore_1	*async*	store.f *word local*	*float local on lhs*
fstore_2	*async*	store.f *word local*	*float local on lhs*
fstore_3	*async*	store.f *word local*	*float local on lhs*
dstore_0	*async*	store.d *dword local*	*double local on lhs*
dstore_1	*async*	store.d *dword local*	*double local on lhs*
dstore_2	*async*	store.d *dword local*	*double local on lhs*
dstore_3	*async*	store.d *dword local*	*double local on lhs*
astore_0	*async*	store.r *word local*	*ref local on lhs*
astore_1	*async*	store.r *word local*	*ref local on lhs*
astore_2	*async*	store.r *word local*	*ref local on lhs*
astore_3	*async*	store.r *word local*	*ref local on lhs*
iastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.i	*int local*[*imm*] *on lhs*
lastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.l	*long local*[*imm*] *on lhs*
fastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.f	*float local*[*imm*] *on lhs*
dastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.d	*double local*[*imm*] *on lhs*

aastore	*async*, NullPointer, ArrayIndexOutOfBounds, ArrayStore	arraywrite.r	*ref local*[*imm*] *on lhs*
bastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.b	*boolean|byte local*[*imm*] *on lhs*
castore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.c	*char local*[*imm*] *on lhs*
sastore	*async*, NullPointer, ArrayIndexOutOfBounds	arraywrite.s	*short local*[*imm*] *on lhs*
pop	*async*	pop.[bcsifr]	—
pop2	*async*	pop.[bcsifr] pop.[bcsifr], pop.[ld]	—
dup	*async*	dup1.[bcsifr], load.[bcsifr] *word local*	*cat1 local on rhs*
dup_x1	*async*	load.[bcsifr] *word local*	*cat1 local on rhs*
dup_x2	*async*	load.[bcsifr] *word local*	*cat1 local on rhs*
dup2	*async*	dup1.[ld], load.[ld] *dword local* , load.[bcsifr] *word local* load.[bcsifr] *word local* , dup1.[bcsif] ... store.[bcsifr] *word local* load.[bcsifr] *word local* ... store.[bcsifr] *word local* load.[bcsifr] *word local* load.[bcsifr] *word local*	*cat1 local on rhs* ... *cat1 local on rhs* , *double|long local on rhs*
dup2_x1	*async*	load.[ld] *dword local*	*double|long local on rhs*
dup2_x2	*async*	load.[ld] *dword local*	*double|long local on rhs*
swap	*async*	—	—
iadd	*async*	add.[bcsi]	*int imm+imm*
ladd	*async*	add.l	*long imm+imm*
fadd	*async*	add.f	*float imm+imm*
dadd	*async*	add.d	*double imm+imm*
isub	*async*	sub.[bcsi]	*int imm−imm*
lsub	*async*	sub.l	*long imm−imm*
fsub	*async*	sub.f	*float imm−imm*
dsub	*async*	sub.d	*double imm−imm*
imul	*async*	mul.[bcsi]	*int imm***imm*
lmul	*async*	mul.l	*long imm***imm*
fmul	*async*	mul.f	*float imm***imm*
dmul	*async*	mul.d	*double imm***imm*
idiv	*async*, Arithmetic	div.[bcsi]	*int imm*/*imm*
ldiv	*async*, Arithmetic	div.l	*long imm*/*imm*
fdiv	*async*	div.f	*float imm*/*imm*
ddiv	*async*	div.d	*double imm*/*imm*
irem	*async*, Arithmetic	rem.[bcsi]	*int imm*%*imm*
lrem	*async*, Arithmetic	rem.l	*long imm*%*imm*
frem	*async*	rem.f	*float imm*%*imm*

drem	*async*	rem.d	*double imm*%*imm*
ineg	*async*	neg.[bcsi]	*int* neg *imm*
lneg	*async*	neg.l	*long* neg *imm*
fneg	*async*	neg.f	*float* neg *imm*
dneg	*async*	neg.d	*double* neg *imm*
ishl	*async*	shl.[bcsi]	*int imm*<<*imm*
lshl	*async*	shl.l	*long imm*<<*imm*
ishr	*async*	shr.[bcsi]	*int imm*>>*imm*
lshr	*async*	shr.l	*long imm*>>*imm*
iushr	*async*	ushr.[bcsi]	*int imm*>>>*imm*
lushr	*async*	ushr.l	*long imm*>>>*imm*
iand	*async*	and.[bcsi]	*int imm*&&*imm*
land	*async*	and.l	*long imm*&&*imm*
ior	*async*	or.[bcsi]	*int imm*||*imm*
lor	*async*	or.l	*long imm*||*imm*
ixor	*async*	xor.[bcsi]	*int imm*^^*imm*
lxor	*async*	xor.l	*long imm*^^*imm*
iinc	*async*	push *int constant* add.[bcsi] , push *int constant* sub.[bcsi]	*int imm*+*imm* , *int imm*-*imm*
i2l	*async*	i2l	(long) *int32 imm*
i2f	*async*	i2f	(float) *int32 imm*
i2d	*async*	i2d	(double) *int32 imm*
l2i	*async*	l2i	(int) *long imm*
l2f	*async*	l2f	(float) *long imm*
l2d	*async*	l2d	(double) *long imm*
f2i	*async*	f2i	(int) *float imm*
f2l	*async*	f2l	(long) *float imm*
f2d	*async*	f2d	(double) *float imm*
d2i	*async*	d2i	(int) *double imm*
d2l	*async*	d2l	(long) *double imm*
d2f	*async*	d2f	(float) *double imm*
i2b	*async*	i2b	(boolean) *int32 imm* , (byte) *int32 imm*
i2c	*async*	i2c	(char) *int32 imm*
i2s	*async*	i2s	(short) *int32 imm*
lcmp	*async*	cmp.l	*long imm* cmp *long imm*
fcmpl	*async*	cmpl.f	*float imm* cmpl *float imm*
fcmpg	*async*	cmpg.f	*float imm* cmpg *float imm*
dcmpl	*async*	cmpl.d	*double imm* cmpl *double imm*
dcmpg	*async*	cmpg.d	*double imm* cmpg *double imm*
ifeq	*async*	ifeq *label*	if *int32 imm* == 0 goto *l*
ifne	*async*	ifne *label*	if *int32 imm* != 0 goto *l*
iflt	*async*	iflt *label*	if *int32 imm* < 0 goto *l*
ifge	*async*	ifge *label*	if *int32 imm* >= 0 goto *l*
ifgt	*async*	ifgt *label*	if *int32 imm* > 0 goto *l*
ifle	*async*	ifle *label*	if *int32 imm* <= 0 goto *l*

if_icmpeq	*async*	ifcmpeq.[bcsi]	if *int32 imm* == *int32 imm* goto *l*
if_icmpne	*async*	ifcmpne.[bcsi]	if *int32 imm* != *int32 imm* goto *l*
if_icmplt	*async*	ifcmplt.[bcsi]	if *int32 imm* < *int32 imm* goto *l*
if_icmpge	*async*	ifcmpge.[bcsi]	if *int32 imm* >= *int32 imm* goto *l*
if_icmpgt	*async*	ifcmpgt.[bcsi]	if *int32 imm* > *int32 imm* goto *l*
if_icmple	*async*	ifcmple.[bcsi]	if *int32 imm* <= *int32 imm* goto *l*
if_acmpeq	*async*	ifcmpeq.r	if *ref imm* == *ref imm* goto *l*
if_acmpne	*async*	ifcmpne.r	if *ref imm* != *ref imm* goto *l*
goto	*async*	goto *label*	goto *label*
jsr	*async*	—	—
ret	*async*	—	—
tableswitch	*async*	tableswitch	tableswitch(*int32 imm*)
lookupswitch	*async*	lookupswitch	lookupswitch(*int32 imm*)
ireturn	*async*, IllegalMonitorState	return.[bcsi]	return *int32 imm*
lreturn	*async*, IllegalMonitorState	return.l	return *long imm*
freturn	*async*, IllegalMonitorState	return.f	return *float imm*
dreturn	*async*, IllegalMonitorState	return.d	return *double imm*
areturn	*async*, IllegalMonitorState	return.r	return *ref imm*
return	*async*, IllegalMonitorState	return	return
getstatic	*async*, *linkage*	staticget *field*	*field on rhs*
putstatic	*async*, *linkage*	staticput *field*	*field on lhs*
getfield	*async*, *linkage*, NullPointer	fieldget *field*	*field on rhs*
putfield	*async*, *linkage*, NullPointer	fieldput *field*	*field on lhs*
invokevirtual	*async*, *linkage*, NullPointer	virtualinvoke *methodspec*	*ref imm* virtualinvoke *methodspec*
invokespecial	*async*, *linkage*, NullPointer	specialinvoke *methodspec*	*ref imm* specialinvoke *methodspec*
invokestatic	*async*, *linkage*	staticinvoke *methodspec*	staticinvoke *methodspec*

`invokeinterface`	*async*, *linkage*, `NullPointer`	`interfaceinvoke` *methodspec* int *constant*	*ref imm* `interfaceinvoke` *methodspec*
`new`	*async*, *linkage*	`new` *typespec*	`new` *typespec*
`newarray`	*async*, `NegativeArraySize`	`newarray.`[bcsildf]	`newarray` `(`*boolean*|*byte*|*short*|*char*|*int*|*float*|*long*|*doub*
`anewarray`	*async*, *linkage*, `NegativeArraySize`	`newarray` *typespec*	`newarray` `(`*typespec*`)[`*dim*`]`
`arraylength`	*async*, `NullPointer`	`arraylength`	`lengthof` *ref imm*
`athrow`	*async*, `NullPointer`, `IllegalMonitorState`, *argumentThrowable*	`athrow`	`lengthof` *ref imm*
`checkcast`	*async*, *linkage*, `ClassCast`	`checkcast` *typespec*	`(`*typespec*`)` *ref imm*
`instanceof`	*async*, *linkage*	`instanceof` *typespec*	*refimm* `instanceof` *typespec*
`monitorenter`	*async*, `NullPointer`	`entermonitor`	`entermonitor` *ref imm*
`monitorexit`	*async*, `NullPointer`, `IllegalMonitorState`	`exitmonitor`	`exitmonitor` *ref imm*
`multianewarray`	*async*, *linkage*, `NegativeArraySize`	`newmultiarray`	`newmultiarray` `(`*typespec*`)[`*dim*`]`...`[`*dim*`]`
`ifnull`	*async*	`ifnull` *label*	`if` *ref imm* `==` `null` `goto` *l*
`ifnonnull`	*async*	`ifnonnull` *label*	`if` *ref imm* `!=` `null` `goto` *l*
`goto_w`	*async*	`goto` *label*	`goto` *label*
`jsr_w`	*async*	—	—

Table 5: Exceptions which Instructions may throw

References

[1] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *Languages and Compilers for Parallel Computing (LCPC'99)*, pages 35–52, 1999.

[2] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 21–31, 1999.

[3] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30:199–232, 2000.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, first edition, 1996.

[5] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, pages 422–446, 2000. ECOOP'00.

[6] Tim Lindholm and Frank Yellin. *The Java virtual machine specification*. Addison-Wesley, first edition, 1997.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

[8] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[9] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in Java. In *Conference on Object-Oriented Programming, Systems, Languages, adn Applications*, pages 83–95, 2001.

[10] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[11] Andrew Stevens. *JeX: an implementation of a Java exception analysis framework to exploit potential optimisations*. PhD thesis, University of Sussex, 2001.

[12] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, 2000.