The Architecture of FindBugs

David Hovemeyer

Contents

Introduction

This document presents a medium-level overview of the architecture of FindBugs. It should be helpful for anyone trying to understand how the tool works and how to extend its capabilities.

This information is accurate as of FindBugs version 0.9.4. The document revision is

```
$Revision: 1.7 $
```

Note that FindBugs has fairly complete javadoc API documentation. The `apiJavadoc` target in the Ant build.xml file will generate this documentation in the `apiJavaDoc` subdirectory.

1 Walkthrough

This section will walk through an analysis session, such as an invocation of the command line interface or the analysis started by the "Find Bugs!" button in the Swing GUI.

1.1 The FindBugs class

To start, we create an instance of the FindBugs class. The FindBugs class serves as the main engine that sets up and executes the analysis. The FindBugs constructor takes two parameters: a BugReporter and a Project.

The BugReporter receives all warnings generated by the analysis. Several bug reporter implementations exist in order to support various output formats. The XMLBugReporter class is perhaps the most useful; it can save generated warnings in a format that can be reloaded without loss of information.

A Project represents the software artifact to be analyzed. The main information in the Project is a list of files and directories containing classes to analyze. These classes are the "application classes". The Project also contains a list of files and directories containing classes that are *referenced* by the application classes, but which should not themselves be analyzed. These are the "auxiliary classpath" classes. Finally, the Project also contains a list of source directories and archives: this list can be consulted to find the source code for a range of instructions reported in a warning.

1.2 Configuring the FindBugs object

There are lots of ways for the user to configure the analysis parameters. Some examples:

- Which detectors are enabled and disabled
- Default reporting threshold (low, medium, high)
- Which bug categories to report (correctness, multithreaded correctness, malicious code vulnerability, etc.)
- Rule-based filtering of warnings
- The analysis effort level: greater effort means more bugs found and longer running time

These parameters are generally configured using accessor methods which modify fields in the FindBugs object. The UserPreferences class serves to organize many of these parameters into a single object.

1.3 FindBugs.execute()

Once the FindBugs object has been created and configured, the execute() method starts the analysis.

1.3.1 The Analysis Context

First, an AnalysisContext object is created. This object is the repository for all global information pertaining to the current analysis session. Examples of information stored in the analysis context:

- The Subtypes object representing the class hierarchy
- Databases of interprocedural analysis facts

In addition, a number of methods supporting access to the classpath and the repository of application classes are available in the AnalysisContext class. Although these tasks can be done using static methods in the BCEL Repository class, eventually we would like to decouple FindBugs from BCEL and its deeply flawed Repository implementation. Therefore, any classpath or class hierarchy queries should be done via the AnalysisContext.

Another way to look at AnalysisContext is that it allows FindBugs to do whole-program analysis. Most FindBugs analysis is local: but as deeper analysis is implemented in FindBugs (e.g., interprocedural analysis, whole-program call graph construction, etc.), this functionality should be available from the AnalysisContext.

Originally, the reference to the AnalysisContext object was explicitly passed to all bug detector objects using a setAnalysisContext() method. More recently, the AnalysisContext is stored in an InheritableThreadLocal, and is accessible from a static method, AnalysisContext.currentAnalysisContext().

1.3.2 Analysis Features and Training Databases

Once the AnalysisContext has been created, the "analysis features" are configured. Analysis features are configuration properties stored in the AnalysisContext that affect how the analysis is performed.

Around the same time as configuration of analysis features, training databases are created and configured. A training database records analysis facts pertaining to a particular method or field, and allows information collected in one analysis run to be re-used in a later run. For example, methods that unconditionally dereference method parameters can be recorded in one run, and loaded in a later run. Any time a null value is passed for an unconditionally dereferenced parameter, FindBugs can issue a warning about a possible null pointer dereference. Training databases are thus a simple way to do limited interprocedural analysis.

1.3.3 Execution Plan

The next main task of the execute() method is to create an execution plan. FindBugs has recently acquired the ability to define ordering constraints among bug detectors, including the ability to specify that a detector must run in an earlier or later pass than another detector. The execution plan explicitly organizes bug detectors into passes, and orders the execution of detectors within passes. Analysis passes can be used in much the same way as training databases: information about methods and fields is collected in an earlier pass and then used in a later pass. The advantage over training databases is that both passes run in the context of a single invocation of FindBugs. It is also sometimes useful to specify that within a single analysis pass, one detector runs before another. With a single pass, each analyzed class is considered (in no particular order). The detectors in the pass are applied to the class in order. Therefore, by specifying an ordering constraint between two detectors, the earlier detector can collect information on the class that can then be used by a later detector (or detectors).

Note: The implementation of ExecutionPlan and related classes is needlessly complicated, and should probably be rewritten at some point.

1.3.4 Scanning for Application Classes

After it creates the execution plan, the next task of execute() is to determine which classes will be analyzed. This is done by scanning all class files, archive files, and directories specified to be part of the analyzed application by the Project. As each class is scanned, it is parsed into a BCEL JavaClass object and placed in the BCEL Repository (via the AnalysisContext). Thus, all application classes are in memory during the entire analysis. (This is one reason why FindBugs requires a lot of memory.)

Note: The BCEL Repository is implemented using static fields and static methods. This makes FindBugs non-reentrant, so it is not possible to create two FindBugs objects and perform two separate analysis sessions at the same time. Unfortunately, BCEL uses the static Repository extensively in its internal implementation, meaning that the Repository is very tightly coupled with the rest of of BCEL. The only long-term solution to this problem is to drop BCEL. (Which we should do anyway.)

```
for each analysis pass in the execution plan do
    for each application class do
        for each detector in the analysis pass do
            request ClassContext for the class from the AnalysisContext
            apply the detector to the ClassContext
        end for
    end for
end for
```

Figure 1: Performing the analysis using the execution plan.

One subtlety of the class scanning process is that we look for Class-Path attributes in Jar manifests in order to discover referenced libraries. When analyzing Java bytecode, it is often necessary to inspect classes that are referenced by analyzed code in order to make the analysis as accurate as possible. For example, when analyzing a method call, you can't tell what exceptions can be thrown unless you can examine the classfile of the class specified as the receiver of the method. Another reason that referenced classes are necessary is that any analysis of the inheritance hierarchy (including the analysis needed to perform type inference on class files) requires access to referenced classes. Thus, FindBugs wants the auxiliary classpath to be as complete as possible. The additional codebases discovered from class scanning are themselves examined for additional classpath references: in the end, we should have the transitive closure of codebases referenced in this way. Hopefully, the user has also made an effort to explicitly specify referenced classes.

1.3.5 Performing the Analysis

Once the execution plan is in place and all of the application classes have been collected, we're ready to perform the analysis. This is done according to the pseudo-code shown in Figure **??**.

Each bug detector is an object implementing the Detector interface, which specifies a visitClassContext() method. This method takes an instance of ClassContext created from the analyzed class. The ClassContext serves as a cache for analysis results from a single class, because many kinds of analysis (such as type inference) are used by many detectors. By performing an analysis only once and then sharing it among multiple detectors, we decrease CPU time at the expense of increasing memory use. (ClassContext is described in more detail in Section **??**.)

1.3.6 Finishing Up

After all analysis passes have executed, we call finish() and reportQueuedErrors() on the BugReporter, and then clear the BCEL Repository (since we no longer need to access classes contained therein).

2 Bug Detectors

This section presents an overview of bug detectors and how they work. The bug detectors implemented in FindBugs can be divided into two basic categories: visitor-based and CFG-based. Ultimately, FindBugs does not place any constraints on how bug detectors are implemented: any

analysis technique that works can and should be used. However, the visitor-based detectors tend to be used for simpler "peephole" analyses, and the CFG-based detectors tend to be used for more complex analyses, especially ones that use the dataflow analysis framework.

Ultimately, the task of a bug detector is very straightforward: look at a single classfile and find potential bugs in it. The detector reports potential bugs by creating a BugInstance object, decorating it with information about the context of the warning and the reasons for reporting it, and then reporting it via the BugReporter.

All of the "core" bug detectors—those that we include with FindBugs and which are enabled by default—live in the edu.umd.cs.findbugs.detect package. However, FindBugs has a plugin architecture allowing users to create their own detectors and use them without modifying FindBugs. By dropping a FindBugs plugin (which is just a jar file with an XML descriptor, message translation files, and detector classes) in the `plugin` directory of a FindBugs installation, new detectors can easily be added. For example, FindBugs core team member Dave Brosius maintains the fb-contrib project (`http://fb-contrib.sourceforge.net`), which provides additional bug detectors.

2.1 Visitor-Based Detectors

The visitor-based detectors are generally subclasses of BytecodeScanningVisitor, which itself is a subclass of DismantleBytecode. The base classes perform a top-down traversal of the features of the classfile, decoding all of the symbolic information. When a feature of the classfile is encountered, such as a field, method, instruction, and so forth, DismantleBytecode invokes a callback method. By overriding these callback methods, a visitor-based detector can inspect the classfile looking for suspicious features that may indicate bugs. For example, the FindUnconditionalWait detector looks for methods containing a `monitorenter` instruction followed by an `invokevirtual` instruction that invokes `Object.wait()`.

One important idiom used in visitor-based detectors is a state machine recognizer over the sequence of instructions in a method. Each invocation of `sawOpcode()`, which is the callback method that handles individual instructions, is a single input symbol to the state machine. The overall state machine is essentially a finite-state automaton that accepts some regular language, where the language is a pattern that, if it appears in the bytecode for a method, indicates a probable bug. In other words, if the string of input symbols (instructions) causes the state machine to accept the string, then the detector issues a warning. Even though this kind of analysis is very simple, especially in its treatment of control flow, detectors implemented this way can be surprisingly effective for many kinds of bugs.

In addition to being simple and concise, visitor-based detectors tend to be *much* faster than the CFG-based detectors. Back when FindBugs used visitor-based detectors exclusively, it was possible to analyze a large jar file such as `rt.jar` in several seconds.

A recent improvement to the visitor-based detectors is the OpcodeStack class, which attempts to maintain information about the operand stack as each instruction in a method is visited. However, it does this in a fairly ad-hoc way. In the future we may implement something more sophisticated to take the place of OpcodeStack. (For example, we could use the existing dataflow analyses to populate the operand stack contents, providing a bridge between the visitor-based detectors and the more sophisticated CFG-based analyses.)

```
for each method in the class do
    request a CFG for the method from the ClassContext
    request one or more analysis objects on the method from the ClassContext
    for each location in the method do
        get the the dataflow facts at the location
        inspect the dataflow facts
        if a dataflow fact indicates an error then
            report a warning
        end if
    end for
end for
```

Figure 2: Pseudo-code for a CFG-based detector.

2.2 CFG-Based Detectors

CFG stands for "Control Flow Graph". The CFG-based detectors use a CFG representation of Java methods to perform somewhat more sophisticated analysis than the visitor-based detectors. In particular, FindBugs implements many kinds of dataflow analysis for use by CFG-based bug detectors.

Unlike the visitor-based detectors, which generally inherit from BytecodeScanningVisitor, CFG-based detectors usually implement the Detector interface directly.

Note: Some common functionality could be factored out of the CFG-based detectors in order to simplify the code.

Figure **??** shows the pseudo-code for the visitClassContext() method of a CFG-based detector. The basic idea is to visit each method of an analyzed class in turn, requesting some number of analysis objects. An analysis object is simply the end product of some analysis: it records facts (or probable facts) about the method based on some form of program analysis, usually dataflow analysis. After getting the required analyses, the detector iterates through each location in the control flow graph. A location is the point in execution just before a particular instruction is executed (or after the instruction, for backwards analyses). Locations are not quite synonymous with instructions because of `jsr` subroutines—more on those in Section **??**. At each location, the detector checks the dataflow facts to see if anything suspicious is going on. For example, the null dereference detector uses a null value dataflow analysis to see if any instructions that have a runtime null check are used on a value that is either definitely null, or is null on a path that through the CFG that is likely to be feasible. If suspicious facts are detected at a location, then the detector issues a warning.

CFG-based detectors are useful for any kind of analysis that can be defined as a forward or backward dataflow problem. Unfortunately, they tend to be far slower than the visitor-based detectors, but hey, computers are fast.[1]

[1] I am not sure if we know exactly why the CFG-based detectors are so much slower. They do require BCEL to create heavier-weight representations of the class, methods, and instructions, which takes up time and memory. Constructing the CFG and performing the dataflow analyses takes additional time. I suspect that at least a 2x or 3x speedup is possible.

3 The Bytecode Analysis (BA) Package

The edu.umd.cs.findbugs.ba package is the "bytecode analysis" package (hereafter referred to as the "BA package"), and is intended to be a generic library of classes for doing static analysis of Java classes. As such, the classes defined in this package (and subpackages) should not refer to any of the "bug-finding" classes, such as the FindBugs warning object model or detector classes. References to BCEL and the FindBugs generic graph library are necessary and permitted.

Note: Although parts of the bytecode analysis package, such as the CFG builder, might be useful in other contexts, most of the analyses implemented in the BA package are probably not that useful in other contexts. In particular, no analysis used in FindBugs is designed to be *sound*, meaning that it deduces only facts that are true. Many, many liberties are taken by the analyses implemented in the BA package: while useful for finding probable bugs, you wouldn't want to use these analyses in a compiler or in a program verifier for safety critical systems.

Note: Until recently the BA package was a single flat Java package. Due to the large number of classes, we have started to organize it into subpackages. This work is still ongoing. Also, you may find that there are some unused or experimental subpackages, such as the "type2" subpackage.

3.1 AnalysisContext and ClassContext

The AnalysisContext and ClassContext classes described earlier are part of the BA package. An AnalysisContext object is first created, and its repository populated with classes to analyze (as well as the codebases of referenced classes). Analyses may be performed on a particular class by submitting the JavaClass representing the class to the AnalysisContext, getting back the ClassContext for the class, and then requesting particular analyses directly from the ClassContext.

Initially, a ClassContext has only a reference to the JavaClass object. Specific analyses are performed on demand and then cached. A few of the analyses, such as the AssignedFieldMap, are global to the class as a whole, and are obtained via a no-argument method. Most of the analyses, however, are performed on methods individually, and are obtained via methods that take a BCEL Method object as a parameter. In any case, whenever an analysis is requested from the ClassContext a cache is first checked. If the analysis has already been performed, the cached version is returned. Otherwise, the analysis is performed, added to the cache, and returned.

Almost all methods that request an analysis can throw both CFGBuilderException and DataflowAnalysisException. Therefore, detectors using the BA package will generally need to be able to handle these exceptions. A typical strategy is to surround the handling of each method in a detector with a try/catch that catches these exceptions and, if they occur, reports them to the BugReporter via one of the logError() methods. Even if an exception occurs while checking one of the methods in the analyzed class, the detector should generally continue analyzing the remaining methods.

3.2 Control Flow Graphs

The Control Flow Graph implementation consists of the CFG, BasicBlock, Edge, and BetterCFGBuilder2[2] classes.

CFG, BasicBlock, and Edge are a very this layer on top of the BCEL MethodGen, InstructionHandle, and Instruction classes. FindBugs uses BCEL InstructionHandle objects to represent single instructions. Thus, if you ask a BasicBlock for the instructions it contains, it will give you back InstructionHandle objects.

The edges of the CFG have an *edge type* that describes the kind of control flow represented by the edge. The constants describing edge types are defined in the EdgeTypes interface. *Exception edges* represent exceptions that lead to an exception handler, or to the exit block of the CFG: these edges have type `HANDLED_EXCEPTION_EDGE` and `UNHANDLED_EXCEPTION_EDGE` respectively. Edges with type `IFCMP_EDGE` represent branches taken by the `ifcmp`, `ifacmp`, `ifnull`, or `ifnonnull` instructions when the condition tested by the instruction evaluates to true. Edges with type `SWITCH_EDGE` or `SWITCH_DEFAULT_EDGE` represent the cases of a switch instruction. Edges with type `GOTO_EDGE` represent unconditional branches. The `JSR_EDGE` and `RET_EDGE` edge types are used to transfer control to and from inlined `jsr` subroutines: for all practical purposes, these can be considered to be equivalent to `GOTO_EDGE`. An edge with type `FALL_THROUGH_EDGE` represents control falling through to the next instruction. Edges with type `RETURN_EDGE` transfer control to the exit block of the CFG, representing either an explicit `return` statement or reaching the end of a void method without an explicit `return` statement.

BetterCFGBuilder2 takes a MethodGen and uses it to construct a CFG object representing the control flow graph for the method. This process is complicated by the fact that Java methods may use the `jsr` instruction, usually because of a try/finally construct. Code reachable from a `jsr` instruction is known as a "jsr subroutine". FindBugs handles jsr subroutines by simply inlining them into the control flow graph. Because they are guaranteed not to be recursive, this process is guaranteed to terminate for any valid Java method, and produces a CFG that is equivalent to the original method.

Because of jsr subroutines, an individual bytecode instruction (InstructionHandle) may appear multiple times in the CFG, once for each expansion of the subroutine it appears in. This is the reason for the Location class: a Location is a BasicBlock, InstructionHandle pair. All dataflow analyses use Location objects as a key to retrieve computed analysis facts because different dataflow facts will be true in different expansions of jsr subroutines. The CFG class has a locationIterator() method that returns an Iterator over all defined locations in a CFG.

Another subtlety of FindBugs CFGs the representation of exception control flow. Generally, any instruction that can throw an exception is preceded by an Exception Throwing Block (ETB). The exception control edge coming from an ETB represents what will happen if the exception occurs: either transferring control to an exception handler, or throwing it out of the method. The fall-through edge coming from the ETB represents what will happen if no exception occurs. One surprising feature of ETB blocks is that they appear to be empty: that is, they contain no actual instruction. The instruction responsible for throwing the exception will be the first instruction in the block reachable from the outgoing fall through edge from the ETB; however, the occurrence of the instruction in the fall through block represents the case where the exception does

[2]This class was preceded by the SimpleCFGBuilder and BetterCFGBuilder classes. Constructing a CFG from Java bytecode is surprisingly difficult. BetterCFGBuilder2 has proved to be very robust, so further evolution will probably not be needed.

not occur. Although this may seem confusing, it means that code that models the semantics of bytecode instructions in a dataflow analysis can completely ignore exceptions, because whenever an instruction appears in a basic block, no exception occurs within the basic block. Instead, they represent an implicit "conditionally throw an exception" instruction: for example, a null check. ETBs may be identified by calling the isExceptionThrower() method on a BasicBlock.

Initially, a FindBugs CFG will assume that any ETB can transfer control to any reachable exception handler in the method, and unless there is a reachable handler that handles *all* exception types (subtypes of Throwable), will assume that the exception can be thrown out of the method. This conservative assumption results from the fact that, during initial CFG construction, we don't really know what methods are being called, and what exceptions those methods can throw. A CFG with conservative exception edges is a "raw" CFG—you can see some evidence of raw CFGs in ClassContext, and a few other places. To the extent possible, ClassContext tries to convert raw CFGs into "pruned" CFGs by removing obviously infeasible exception edges. The more accurate the exception edges, the more accurate analyses that use CFGs can be.

3.3 Dataflow Analysis

FindBugs implements a very generic dataflow analysis framework. It is generic in the sense that any kind of object can be used as a dataflow fact, and also in the sense that type parameters are used wherever appropriate. Thus, adding a new kind of dataflow analysis to FindBugs tends to be a fairly simple exercise.

This section will briefly discuss the generic parts of the dataflow framework, and also discuss some of the most important concrete dataflow analysis classes.

3.3.1 Dataflow and DataflowAnalysis

The DataflowAnalysis interface is the supertype for all concrete dataflow analysis classes. It defines methods for creating and manipulating dataflow facts. The Dataflow class implements the actual dataflow analysis algorithm using a CFG and an instance of DataflowAnalysis.

AbstractDataflowAnalysis is an abstract class implementing DataflowAnalysis, and providing default implementations of some of its methods. In addition, AbstractDataflowAnalysis defines the notion of implementing the dataflow transfer function at the level of individual instructions rather than complete basic blocks, and automatically implements a cache recording analysis facts at each CFG location. Most dataflow analyses where dataflow facts are modeled at the instruction level will want to inherit from AbstractDataflowAnalysis. However, there are some exceptions. For example, DominatorsAnalysis is a dataflow analysis where the transfer function makes sense only at the level of complete basic blocks: individual instructions have no effect on dataflow facts.

ForwardDataflowAnalysis and BackwardDataflowAnalysis, as their names suggest, are generic base classes for forward and backward dataflow analyses. Both of these classes inherit from AbstractDataflowAnalysis. So, when defining a new dataflow analysis class, inheriting from one of these classes is a reasonable choice. The StackDepthAnalysis class is a very good example of how to implement a forward dataflow analysis in FindBugs. It uses the StackDepth class to represent dataflow facts, and computes the depth of the Java operand stack at each Location in the analyzed CFG.

3.3.2 Frame, FrameDataflowAnalysis, and AbstractFrameModelingVisitor

In many dataflow analyses, we want to know something about the values of local variables and stack operands. The Frame class represents the Java stack frame at a single CFG location, and models stack operands and local variables. Note that both stack operands and locals are considered to be "slots" that contain a single symbolic value: there is really nothing fundamentally different about them, other than the way that they are accessed by instructions. Most Java instructions operate exclusively on stack operands.

Note: A peculiarity of Frame-based dataflow analysis in FindBugs is that long and double values are considered to occupy two stack slots, or two local variable slots. This quirk is inherited from BCEL, which inherits it from the original JVM specification from Sun.

The Frame class is parametrized with the type (class) representing the symbolic values stored in the locals and stack locations. By convention, you should inherit from Frame to create a Frame subtype that is specific to your analysis. For example, IsNullValueFrame is a frame subtype where the locals and stack operands are of type IsNullValue, and is used (no surprise) by IsNullValueAnalysis.

FrameDataflowAnalysis is a convenient abstract base class for dataflow analysis classes that will use Frames as dataflow facts. Its specific purpose is to allow the dataflow meet function to be specified on individual symbolic values rather than entire frames.

AbstractFrameModelingVisitor is a useful base class for defining a visitor that models the effect of bytecode instructions on a Frame. Using the notion of a "default" dataflow value, it provides generic implementations for all Java bytecode instructions. You can then simply override the instruction-visiting methods that are actually important for the particular analysis.

3.3.3 Using the Results of a Dataflow Analysis

The most convenient way to access and use the results of applying a particular dataflow analysis to a method is to get a reference to the Dataflow object that was used to execute the analysis. For example, to get the results of applying IsNullValueAnalysis, get a reference to the IsNullValue-Dataflow object that executed the analysis. (You can get this object directly from the ClassContext using the getIsNullValueDataflow() method, passing a reference to the Method object you want to analyze.)

All Dataflow subclasses support the methods getStartFact() and getResultFact(), which get the dataflow fact before or after a particular basic block, respectively. Most Dataflow subclasses also support methods called getFactAtLocation() and getFactAfterLocation(), which return the dataflow fact immediately before or after a particular Location in the CFG.

3.3.4 TypeAnalysis and TypeDataflow

One of the most important dataflow analyses in FindBugs is TypeAnalysis/TypeDataflow. This frame-based analysis performs type inference on a Java method, computing types for all local variables and stack operands.

The dataflow meet operation for types in this analysis computes the least upper bound of the merged values in the *class* inheritance hierarchy. Note that the least upper bound for superinterfaces of two types is not guaranteed to yield a unique result in Java, due to multiple inheritance of

interfaces. Therefore, TypeAnalysis in FindBugs uses the same recourse as the JVM verifier, which is to consider only superclasses. One consequence of this limitation is that the type of a receiver object in an `invokeinterface` instruction is often only known as java.lang.Object, because dataflow merges tend to lose information about implemented interfaces.

Note: In the long term, it would be nice to have type information that was based on *sets* of types.

3.3.5 ValueNumberAnalysis and ValueNumberDataflow

A common problem in analysis of Java bytecode is determining when two slots in a stack frame contain the same value. This problem is especially important when you consider that Java bytecode is stack-based: most instructions can only manipulate stack values, not values in local variables. Information gained when modeling an instruction that affects values on the stack should also be applied to identical values elsewhere on the stack, or in local variables.

ValueNumberAnalysis and ValueNumberDataflow attempt to assign value numbers to each slot in each frame. If two frame slots have the same value number, then at runtime they definitely contain the same value.

One application of value numbers is in the null value dataflow analysis, which makes use of the various `if` comparison instructions. When a value is compared against null, we learn something about the value: depending on which branch is taken, the value is either definitely null or definitely not null. The `if` comparison instructions—`ifnull`, `ifnonnull`, `ifacmp_eq`, and `ifacmp_ne`—operate on values that are on top of the operand stack. On the edges leading from these instructions, we can use the information gained about the value compared to null on the top of the operand stack to update other occurrences of the same value elsewhere on the operand stack or in local variables.

One feature of the value number analysis in FindBugs is that it performs redundant load elimination and forward substitution. This is required in blocks of code like the following:

```
synchronized (this.lock) {
    while (!someCondition()) {
        this.lock.wait();
    }
}
```

Without redundant load elimination, the value number analysis would not assume that the two references to `this.lock` refer to the same value. While the Java memory model does allow the two loads to return different values (e.g., another thread might update the field between accesses), by far the most likely behavior is that both loads will return the same value (and most JVMs will optimize the code to guarantee this outcome). Without redundant load elimination, FindBugs will assume that the `wait()` call is made to a different object than the one synchronized on.

3.3.6 IsNullValueAnalysis and IsNullValueDataflow

IsNullValueAnalysis determines which frame slots contain definitely-null values, definitely non-null values, and various kinds of conditionally-null or uncertain values. This information is used by the FindNullDeref detector to detect possible null pointer dereferences and redundant null comparisons.

3.3.7 LockAnalysis and LockDataflow

LockAnalysis computes LockSet values at each CFG location. A LockSet is the set of value numbers that are known to be locked. This analysis is used, for example, by the FindInconsistentSync2 detector, which looks for unsynchronized field accesses that might be reachable from multiple threads. Note that this analysis is strictly intra-procedural: it does not consider that some values might be locked in a calling method.

3.4 Class Hierarchy Queries and Missing Classes/Interfaces

A very common task in bug detectors and static analysis in general is to perform class hierarchy queries. Some examples:

1. Find immediate superclass and superinterfaces of a type
2. Find transitive superclasses and superinterfaces of a type
3. Find the superclass method that a particular method overrides or implements
4. Find all subtypes of a particular class or interface
5. Find all possible methods that might be called by a particular `invokevirtual` instruction

The first three examples are *upwards* queries: they only require superclasses and superinterfaces to answer. Since superclasses and superinterfaces must be present for a class to be loaded by the JVM, it is reasonable to assume that a static analysis can find them. Therefore, we make the assumption that upwards queries can always be answered definitively, and that if a superclass or superinterface cannot be found, it is a serious error. In fact, many methods in BCEL implicitly do upwards queries using the BCEL Repository class, and throw ClassNotFoundException if a required superclass or superinterface is missing. Any time you are in a context where you need to handle ClassNotFoundException, you should find a RepositoryLookupFailureCallback object and pass the exception to its reportMissingClass() method. All BugReporter objects implement RepositoryLookupFailureCallback, so you can use the BugReporter. You can also call the static reportMissingClass() method in AnalysisContext, which will also locate the current lookup failure callback (generally the bug reporter) and pass the exception along. (The latter approach is necessary within the BA package, since it should not have any dependencies on the bug-finding code.)

The last two examples are *downwards* queries: they attempt to answer questions about possible subtypes. This is a closed world assumption that is generally unsound because Java allows classes to be loaded dynamically. However, in a static analysis to find bugs we don't require soundness, and for most real applications it a reasonable assumption that the user will give the analysis access to the complete application. Therefore, FindBugs uses both upwards and downwards class hierarchy queries. (Upwards queries are used much more extensively.)

It would be nice to say that there is a single well-designed interface for class hierarchy queries in FindBugs. However, this is not quite the case. The Subtypes class provides a nice interface for downwards queries. The Hierarchy class provides a number of upwards query methods, although these are somewhat less well thought-out.

Note: Designing and implementing a better API for class hierarchy queries would be a good project.

4 The Warning Object Model

This section describes the object model used to represent FindBugs warnings.

4.1 BugInstance

When a detector finds code that is likely to be an error, it should report a warning. In FindBugs, the BugInstance class represents a single warning, including the type of warning and associated context information.

Note: The BugInstance class really should have been called "Warning".

A BugInstance is created with three parameters:

Detector: A reference to the Detector reporting the warning.

Bug type: This is a String that contains the warning's *bug code*.

Priority: One of `HIGH_PRIORITY`, `NORMAL_PRIORITY`, and `LOW_PRIORITY`. (These are integer constants that are defined in the Detector interface.) By convention, normal or high priority is only used when there is a high degree of confidence that the user will consider the warning to represent a serious problem.

Note: `NORMAL_PRIORITY` should have been called `MEDIUM_PRIORITY`.

Note: The current single-valued priority scheme conflates two distinct notions: the *confidence* that a warning is accurate, and the *severity* of the warning if it is accurate. We should change the object model so that these values are represented separately. In addition, having only three discrete values for priority is too coarse: sometimes it would be nice to make finer distinctions between warnings. Therefore, priority, confidence, and/or severity reports should be real-valued within the range 0.0 (least confident/severe) to 1.0 (most confident/severe).

4.2 BugPatterns

The bug type used to create a BugInstance is a unique key that defines a particular *bug pattern*. Behind the scenes, there is a BugPattern class that represents the bug patterns defined by a particular FindBugs plugin. Each bug pattern has some meta-information associated with it that can be accessed through the BugPattern object.

4.3 BugAnnotation

In its raw form, the BugInstance doesn't have any information about important details such as what class the warning is being reported in, what method, which source lines are affected, and so forth. BugAnnotation objects provide this information. Some kinds of BugAnnotations include:

ClassAnnotation: A class

MethodAnnotation: A method in a class

FieldAnnotation: A field of a class

SourceLineAnnotation: A range of source lines in a source file

IntAnnotation: An integer value

BugAnnotations should be added to a BugInstance in a well-defined order. The order and types of BugAnnotations vary by bug pattern, but there are some conventions. The most important convention is that the first BugAnnotation must be a ClassAnnotation: every warning is associated with a single primary class. If the warning pertains to code, then the second BugAnnotation should be a MethodAnnotation and the third should be a SourceLineAnnotation; together, these define the code affected by the warning. Otherwise, BugAnnotations can be added in more or less free-form. BugInstances sharing a common bug type may even have different numbers of BugAnnotations, depending on the exact circumstances in which the warnings were reported.

4.4 I18N

Formatting BugInstance and BugAnnotation objects in human readable form is done using format strings. The BugAnnotation format strings are stored directly in findbugs.jar, and accessed through resource bundles. The BugInstance format strings are loaded through XML files in the plugin containing the bug detector reporting the warning. The reason for this division is that BugAnnotations are part of the core object model, and are not defined by detectors. Bug patterns, on the other hand, are defined by individual plugins, so their messages are local to the plugin.

A single BugAnnotation class can have multiple format strings, depending on the *role* of the BugAnnotation. For example, a MethodAnnotation might be the method in which the warning is reported (this is the default role), or could be a method called at a call site reported by the method (the `METHOD_CALLED` role).

Formatting model objects as human-readable Strings is done by the I18N class. The default language is English, but partial or full translations contributed by FindBugs users also exist for French, German, and Japanese. Any missing message translations fall back to the default English translations.

4.5 BugCollection

The BugCollection class defines (no surprise) a collection of BugInstance objects: generally, the output of a single analysis session. In addition to storing BugInstance objects, the BugCollection also stores some meta-information about the analysis session, such as missing classes, reported analysis errors, etc. In recent versions of FindBugs, a single BugCollection can represent a history of analysis sessions on a single software artifact, where each BugInstance may be present in multiple sessions. The *workflow* tools can use this functionality to do data mining on analysis results over time.

There is only one concrete BugCollection class, called SortedBugCollection.

Note: BugCollection and SortedBugCollection are not a particularly elegant design.

4.6 Reading and Writing XML

The XML format supported by FindBugs is a very straightforward representation of the underlying object model. As such, it is great for being able to store a BugCollection and later load it without loss of information. However, because it does not contain human-readable messages, it is not very good for conversion to other formats, e.g. via an XML stylesheet.

In recent versions, FindBugs supports an `-xml:withMessages` option that adds formatted strings (reflecting the current language/locale settings) to the saved XML. If the saved XML is re-loaded, these messages will be discarded. However, they are useful for creating human-readable output (such as HTML reports) from the saved XML.