Evaluating Static Analysis Frameworks

Ciera Nicole Christopher
Carnegie Mellon University
Analysis of Software Artifacts
17-754
cchristo@cs.cmu.edu

May 10, 2006

Abstract

Many static analyses share a core set of common techniques. These techniques can be described in a static analysis framework so that new analyses can reuse a common infrastructure. This allows companies to create proprietary static analyses for specific domains. In this paper, we examine the ideal form for a static analysis framework and evaluate four existing frameworks against this ideal. We also consider where these frameworks need to improve to meet this ideal and satisfy the needs of a sample scenario.

1 Introduction

Static analysis has already shown to be a worthwhile endeavor for software companies since it allows them to automatically discover defects with minimal overhead. However, many of the static analysis tools that currently exist are "generic"; they find defect paradigms that are common across all software. While it is very useful t o find errors such as null pointers, buffer overflows, and unhandled exceptions, there are defects that are specific to domains, frameworks, and company policies. While using generic analyses would still help these software projects, they would like to be able to have customizations.

Consider the following scenario:
A web based software project would like to check that one of their security protocols in enforced. This protocol states that any instance of class X may not be stored as a member data. Additionally, any information retrieved from class X may not be stored. Instances of this class may be used to make local decisions only.

This company would like to statically analyze their code to check for violations of this protocol. However, generic analysis tools do not check for this error.

In this situation, some larger companies have the option of writing their own analysis tools. This takes a lot of investment though, and small to medium sized companies do not have the economic ability to make this happen.

Static analysis frameworks make it possible for these companies to write their own analyses with much less investment. Additionally, a common framework would allow many stakeholders to exchange analyses, thus making the pool of analyses larger for everyone.

In this paper, we will start by fashioning the "ideal" framework that a company in this position would like to have. We will create an evaluation matrix based upon the properties that we believe are important for a static analysis framework. We will then evaluate four existing frameworks and discuss what work is required

Table 1: Evaluation Matrix

Category	**Property**	**Weight**
Licensing		
	Intellectual Property	8
	Open Source	4
Abstract Syntax Tree		
	Mental Model	10
	Specific API	8
	Unambiguous	5
Analysis Types		
	Simple AST Analysis	5
	Dataflow Analysis	10
	Path-sensitive Analysis	5
Utilities		
	Dataflow	10
	Meta-data	2
	Analysis Dependencies	7
Deployment		
	Adding Analyses	10
Understanding		
	Docs	6
	Existing Analyses	6
	Community	4
Total		**100**

to make each closer to the "ideal" analysis framework. We will conclude by looking doing a final comparison of the four frameworks and look at what needs to change to help make analysis frameworks a viable solution.

2 The Ideal Framework

Our ideal framework will help medium-sized software companies write custom dataflow analyses and deploy them to an internal team. We will create an evaluation matrix that describes important properties of an ideal framework. For ease of discussion, we have grouped the properties into higher level categories. The evaluation matrix is in Table 1.

Since not all properties are equally important, each property has a weight to balance them against the other properties. The total weights equals 100 points. When we evaluate the individual frameworks, we will score them on a 0-10 point scale for each property. These points multiplied by the weight equal the total score the framework receives for that property. Therefore, a framework can receive a maximum of 1000 points.

In this section, we will discuss each property to explain why it is important and how we will evaluate the property.

Licensing

Licensing is an important concern for any tool a company uses. The custom analysis techniques may contain proprietary information that the company would like to keep secret. To allow this, the license must let third

parties write new analyses and must let them maintain proprietary access to the new analyses. The license must not force the company to give up their intellectual property rights or their code.

Ideally, the framework itself would be open source. This will help the company by allowing them to make framework modifications that support new analysis techniques. It is acceptable if the license does not give the company proprietary access to these changes; such changes should be generic and would not reveal any company secrets.

Abstract Syntax Tree

The AST that represents the code is one of the most basic abstractions in a static analysis. We define the usability of an AST with three propeties: Mental Model, Specific API, and Unambiguous.

First, the AST must match the "mental model" of what a developer expects for the source language. For example, if the source language is Java, most Java developers expect an AST to have some construct that maps to a `for` loop and some construct that maps to an assignment statement. The mapping from source code to the AST should be intuitive for a developer who is familiar with the source language.

Second, the AST must be represented with a specific API. That is, the API must allow a developer direct access to the expected internals of a given AST node. The developer should not have to search to find their way down the tree. For example, if we have a construct that represents assignment, the API should provide direct access to the left and right expressions. If we have a construct that represents a `for` loop,the API should provide direct access to the initializers, conditional, updaters, and body. Additionally, the access mechanism for these internal constructs should return as specific of a type as possible; they should not return some generic construct that must be casted to a more specific type.

Finally, the AST should be unambiguous. A statement in the code should always be represented by the same construct in the AST, it should not be possible for the same statement to be represented by two different AST constructs.

Analysis Types

We would like for the framework to provide as many kinds of analysis as possible. For this evaluation, we will focus on three specific analysis techniques: AST walker analysis, dataflow analysis, and path-sensitive analysis. For each of these techniques, we will evaluate based upon whether the technique is possible, how easy it is to use, and, if the technique is not possible currently, how much work would be required to add the technique.

Utilities

We would like for the framework to provide many utilities that will make creating analyses easier. Specifically, there are three kinds of utilities that are particularly useful for the analysis techniques mentioned above.

The first utilities assist dataflow analysis. This includes utilities that allow for many different styles of dataflow (like forward and backward analysis), plus utilities that will help us keep track of lattices and map variable names to bindings.

The second utilities are bookkeeping of meta-data. Many analyses use meta-data to prevent false positives and guide the analysis towards errors. We would like utilities that will help custom analyses search for relevant meta-data. Additionally, some analysis techniques may want to add meta-data for their own purposes or for another analysis technique. There should be utilities that allow analyses to add this meta-data, either

directly to the code or indirectly in storage of the tool.

The third utilities allow developers to specify dependencies between analyses. This is very useful for many complex analyses that rely on information gathered from simpler analyses. For example, a simple analysis could create summaries in the manner of PREfix [5] and then have a more complex analysis use these summaries to produce an inter-procedural analysis.

Deployment

We are concerned with one aspect of deployment. It must be easy to add and change analyses without forcing developers to reinstall the entire tool. Specifically, adding a new analysis should not require changes to the framework code. It should be possible to create a separate deployment entity for custom analyses and simply add those entities to the project. Changing and adding external property files is also an acceptable mechanism for adding analyses.

Understanding

The last important property that we will look at is programmer understanding. We want to make sure that the developers have access to resources that will help them learn the framework and create new analyses. We will evaluate three properties that help understanding: documentation, example analyses, and the developer community.

The documentation for the framework should be available in some standard form. There should be both documentation of each component and documentation on how the components work together. There should also be some tutorials available.

The framework should come with some sample analyses that perform generic checks such as null pointers and buffer overflows. The code for these sample analyses should be available for developers to peruse and modify. The sample analyses should be not be intertwined with the framework; they should only use generic utilities that all analyses can rely on.

The framework should also have a strong developer community so that new developers can ask questions and receive assistance for developing new analyses. An active community will also make it easier to exchange analyses with third parties.

3 Crystal

Crystal is a small, lightweight framework used by the Analysis of Software Artifacts course at Carnegie Mellon University. Its primary purpose is to teach students how dataflow analysis work and to help students understand what types of defects a dataflow analysis can find.

Licensing

Crystal does not have a licensing policy because it is not currently available to the public. Therefore, Crystal receives 0 points for both Intellectual Property and Open Source.

Abstract Syntax Tree

Crystal uses the parser and AST provided by Eclipse. [10] The Eclipse AST does a very good job of matching a Java developer's mental model. It provides separate types for all the basic Java constructs, including `for` loops and assignment expressions. Additionally, the API is very specific. An `Assignment` has the methods `Expression getLeftHandSide()` and `Expression getRightHandSide()`. It would be nice if the API used Java generics so that `ForStatement.updaters()` would return `List<Expression>` rather than just `List`. Additionally, there are some places where one method may return two different things, such as the ability of `ForStatement.initializers()` to return either a single `VariableDeclarationExpression` or a list of `ExpressionStatement`s. Finally, the AST does have some ambiguities that make it tricky to use. For example, the string `x.field`, may be represented as a `FieldAccess`, which is correct, but it also may be represented as a `QualifiedName`.

Since the Eclipse AST does match a developer's mental model, it receives 10 points for that property. The API is very specific but does have a few minor issues such as those mentioned above; it receives 8 points. Finally, the Eclipse AST receives 7 points for the unambiguous property since it has a known ambiguity that a developer must be conscious of when writing an analysis.

Analysis Types

Crystal provides classes that will do an AST walker analysis on either a method or class level of granularity. Each of these abstract classes use a visitor pattern to allow derived analyses to chose which nodes they would like to listen for. Adding a simple AST analysis is fairly straightforward and required only deriving from one of these two classes. However, a simple AST analysis usually requires overriding many methods and then building a small state machine or it requires checking a node's parents and children to see where it is relative to other nodes. Crystal receives 8 points for the simple AST analysis.
Crystal also provides a way to do dataflow analysis. Creating a dataflow analysis requires three steps:

1. Create the lattice
   The developer must derive from `LatticeElement` and provide merge and copy functions for the lattice.
2. Create the transfer functions
   The developer must derive from `FlowAnalysisDefinition` and use the visitor pattern to provide any transfer functions.
3. Visit `ASTNode`s to find errors
   The developer must derive from either `AbstractCompilationUnitAnalysis` or `AbstractCrystalMethodAnalysis`, create a `FlowAnalysisVisitor`, and use a visitor pattern to find the errors.

The classes provided map very nicely into the basic dataflow abstractions. There is some confusion with using the TupleLattice due to the generics involved, but overall it is fairly good. Crystal receives 8 points for dataflow analysis.

Crystal does not provide any way to create a path sensitive analysis. To add a path sensitive analysis would require changes directly to the framework. Therefore, Crystal receives 0 points for path sensitive analysis.

Utilities

Crystal does not provide many utilities for dataflow analysis other than the basic abstractions outlined above. It does provide some basic utilities, such as retrieving the bindings of variable names. Crystal does not provide any utilities for meta-data, and it does not provide a way for analyses to show dependencies. Therefore, it receives 2 points for providing some very basic dataflow utilities, 0 points for meta-data utilities, and 0 points for analysis dependencies.

Deployment

Deploying new analyses requires direct changes to the Crystal framework. It should be possible to create a wrapper, but this means dealing with the Eclipse plugin framework and may take a great deal of time. Therefore, Crystal receives 0 points for deploying new analyses.

Understanding

Crystal only comes with some very simple analyses that write out the AST, so it receives 4 points. The documentation is very minimal and earns Crystal only 2 points. Since Crystal has not be released to the general public, it receives 0 points for community.

Overall Evaluation

Crystal does not work very well for our purposes. it received 375 points out of a possible 1000. This stems from two large problems: it is unreleased and it provides very few utilities.

Since Crystal is unreleased, it can not be used by industry. Additionally, there is no reason to produce documentation and sample analyses if it is only used in a classroom setting. However, this is not a technical problem with Crystal; it is just a managerial decision to keep it proprietary.

The technical problem with Crystal is that it does not provide the utilities required to make many kinds of analyses. It needs more support for different kinds of dataflow analyses, and it should also support analysis dependencies. These two changes would greatly help the technical usability of Crystal as a framework.

Overall, Crystal is very good for its current purpose as a teaching tool. It's very simple framework that makes creating dataflow analyses very easy. However, it's not powerful enough to be used in our example scenario.

4 PMD

PMD is an open source tool for finding potential programming errors. Most of the errors that PMD looks for are stylistic in nature; they are not typically errors but are signs of bad programming practice that may lead to errors later. As an analysis framework, PMD is medium size, but it still has a very lightweight feel. There are very few abstractions that the developer needs to know about to get a simple analysis running.

Licensing

PMD is available from sourceforge.net and it uses a BSD-style license. [3] This license is very open and allows modifications and redistributions as long as the IP of the original codebase stays the same. It does not require for users of the framework to take on the same license. PMD receives 10 points for its license and 10 points for being open source.

Abstract Syntax Tree

PMD uses a home-grown abstract syntax tree. The AST is based upon an XML schema that describes a Java source file. While this is a Java AST, it does not always match the mental model of a developer. Figure 1 shows a snipet of code and the AST for this statement as an XML snipet.

The first interesting thing we notice is that there is no assignment node. Instead, there is an assignment operator with an primary expression as the sibling above it and an expression as the sibling below it. This means that searching the the expressions within an assignment may take multiple steps, first we must find the assignment operator, then we must go back and get the expressions that are sibling to it.

The larger problem is that the AST is overly general. The AST does not break down the right hand side of the assignment. Instead, it sets the text `field.x` as the image of a `Name`. If a developer wanted to look for field accesses or array accesses, they would have to parse the string by hand. Once this is complete, the developer would have to search the AST to find the declaration of the desired variables. While PMD does provide a Java-like mental mode, it receives only 5 points since it requires developers to do some of the parsing by hand.

```
Node getFirstParentOfType(Class parentType)

List getParentsOfType(Class parentType)

Node getFirstChildOfType(Class childType)

List findChildrenOfType(Class targetType)

String getImage()
```

Figure 2: Subset of available methods in SimpleNode

The more concerning problem with PMD is the API provided. Nodes can not access specific nodes beneath them. All nodes are derived from the base class `SimpleNode`, and this class provides all the functionality. The specific types of nodes do not add their own behaviors. Figure 2 shows the main API of `SimpleNode`. The problem with this API is that all nodes can only retrieve a list of their children. There is no method to retrieve an object of a specific type with a particular semantic meaning. The API does provide some utilities so that the developer does not have to search for children of a given type by hand, but this still loses the semantic meaning of the returned values. PMD receives 7 points for its AST API because while we do lose some semantic meaning, it still works well enough to create an analysis.

Finally, PMD receives 10 points for being unambiguous. We should note that PMD was able to create an unambiguous AST by being overly general in the creation of the AST. Any code that might have lead to an ambiguous situation gets lumped into the same node, so PMD does not have any ambiguities. Since we already removed points for being overly general in the Mental Model property, PMD receives full points for unambiguity.

```
 1 public class C {
 2    C field;
 3    int x;
 4
 5    void foo() {
 6       int y;
          ...
 9       y = field.x;
          ...
15    }
16 }
```

```
<BlockStatement allocation="false">
    <Statement>
       <StatementExpression>
          <PrimaryExpression>
             <PrimaryPrefix>
                <Name image=y"/>
             </PrimaryPrefix>
          </PrimaryExpression>
          <AssignmentOperator image="="/>
          <Expression ">
             <PrimaryExpression>
                <PrimaryPrefix>
                   <Name image="field.x"/>
                </PrimaryPrefix>
             </PrimaryExpression>
          </Expression>
       </StatementExpression>
    </Statement>
 </BlockStatement>
```

Figure 1: XML-styled AST for y = field.x

Analysis Types

The main goal of PMD is to make AST walker analyses easy to write. They have been very successful with this. To make a new analysis, the developer just derives from `Rule` and uses a visitor pattern to access the AST nodes as they are walked. Since the AST can be represented in XML form, PMD allows XPath styled searches in the AST. Additionally, one of the existing rules, `XPathRule`, is parameterized over XPaths. Many of the default rules that PMD uses are actually instances of the XPathRule instantiated with an XPath that represents a bad structure for the AST. Instantiating a new XPathRule does not even require any code; the developer only needs to provide an XML file that defines the new rule. Since it is so easy to write an AST walker analysis that code isn't even required, PMD receives 10 points for this type of analysis.

While PMD has good support for AST walkers, it provides very little support for dataflow. Currently, PMD has an experimental dataflow module that they are still working on, and there are very few dataflow rules already written. PMD does not provide any path sensitive analysis, and at this point it looks like it would be difficult to add this. PMD receives 3 points for dataflow analysis and 0 points for path sensitive analysis.

Utilities

Since the dataflow module is still experimental, there are not many utilities available for dataflow. There are also no utilities for meta-data support, and there is no way to signify analysis dependencies in PMD. PMD receives 0 points for all of these utilities.

Deployment

The basic deployment scenario for PMD is good. Custom rules are wrapped up in a jar and dropped onto the classpath. To create instances of the rule, the developer creates an XML file that defines the "ruleset". A ruleset is simply a group of rule instances that are all run together. This XML file defines the name of the rule, the class, and sets any properties. PMD simply reads the file and loads in the appropriate rule.

In most cases this works perfectly fine. There are some problems if the ruleset or the rule have something wrong with them. PMD does not fail cleanly in error situations, so it can be tricky to figure out why PMD suddenly stops working when the custom rule starts up. PMD receives 7 points for deployment since it works very nicely in the success case but does not handle the failure cases well.

Understanding

PMD has a lot of existing rules for simple analyses, but it has very few rules for more complex analyses. The sheer number of existing rules is helpful because there's almost always an existing rule that's similar to what a developer needs. PMD is documented very well in javadoc form, and there is also a book available to help people write their own analyses. The PMD community is small and not very active; many questions on the forum are answered with "buy the book". However, every question gets a few responses, and the maintainers are quick to update the main web page if they notice the same question asked multiple times. PMD receives 8 points for existing analyses, 10 points for the documentation, and 8 points for the community.

Overall Evaluation

The PMD framework is not strong enough for our example scenario. The evaluation matrix for PMD gives it 566 points out of 1000.

While PMD is a aimed at industry and provides good support, it is not complex enough to handle the kind of analyses like the one in the original scenario. In order for PMD to become a stronger contender, it needs to provide a stable dataflow analysis module, and it needs more "generic" dataflow analyses to show what can be achieved. Additionally, the AST needs some enhancements to make it more complete, and the API for the AST needs to be more specific. Changing the AST would break currently analyses though, so it may not be feasible for PMD to make these changes. While PMD makes AST walker analyses very simple, it will not fulfill the needs of the original scenario.

5 FindBugs

FindBugs was originally a research tool from the University of Maryland, and it is now an open source tool on sourceforge.net. [2] The original purpose of FindBugs was to show that unsound and incomplete analyses can still be useful if they provide the user with a detailed error message. FindBugs relies on "bug patterns" to achieve this. When a user sees an error message, it comes along with a code to the "bug pattern". If the user does not understand the bug, they can look up the bug pattern and quickly figure out what is wrong with the code and what can be done to fix the problem. [7]

FindBugs provides an extensive framework for creating static analyses. The framework is thin and wide to make it easy to add new analyses as well as new analysis techniques.

Licensing

FindBugs is protected by the LGPL. [9] This license does allow companies to use FindBugs as a library and keep their own analyses proprietary. However, any modifications to FindBugs itself must be released under the LGPL. Since framework modifications should be generic to many analyses, this should not expose any properietary information. FindBugs is open source and is available at sourceforge.net. We will give FindBugs full points for using a license that allows proprietary software to use it and for being open source.

Abstract Syntax Tree

FindBugs does not use a typical AST. Rather than providing a Java AST, FindBugs uses the BCEL, a bytecode library. [1] Since most developers do not program in bytecode, this does not match a developer's mental model for a Java AST. We will give FindBugs 3 points for matching the mental model of the source language since it is reasonable to expect a Java developer to learn bytecode.

BCEL appears to provide two different visitor patterns with different APIs. Interestingly, the API that the FindBugs analyses typically use seems to be more difficult to understand. This API provides a visitor pattern on higher level constructs such as classes, fields, and methods. However, when it gets to the actual code, there is a single callback for all the nodes, `void sawOpCode(int seen)`. The existing analyses all override this method and implement it with a switch statement on the opcode and a state machine to look for a particular pattern.

There is a second API in the BCEL that provides a more standard visitor pattern. This API has a seperate callback for every opcode, and each opcode is represented with a subclass of `Instruction`, the base class for opcode types. Some examples of this API are `void visitBranchInstruction(BranchInstruction obj)` and `void visit ALOAD(ALOAD obj)`. The abstractions provide specific information about the instance of the opcode, for example, `BranchInstruction` provides the target of the branch in addition to the integer opcode. Additionally, there are derived versions of `BranchInstruction` to represent different kinds of branches, like ifs, gotos, and jumps.

There seems to be no reason why the second API could not be used instead of the first since these visitors are started by another customized class. Since the second API does provide clear, specific abstractions like `ALOAD` and `BranchInstruction`, FindBugs receives 10 points for a specific API.

Finally, FindBugs receives 10 points for no ambiguities since every byte code has a distinct mapping into the API.

Analysis Types

Creating an AST walker analysis in FindBugs requires the developer to implement `Detector`. Usually, this same class derives from one of the visitors. A `Detector` works on a class level and receives a callback for every class. The analysis then starts up the visitor pattern. This is not as easy as in PMD, but it does use a standard pattern. FindBugs receives 8 points for the AST walker analyses.

To create a dataflow analysis takes the following steps:

1. Create a `Detector`
   The `Detector` creates a dataflow analysis and uses it as it walks through the AST.
2. Create a `DataflowAnalysis`
   The `DataflowAnalysis` defines the flow functions and the block order. There are many kinds of dataflow analyses that already determine the block order, so most analyses derive from one of these.
3. Create a `Fact` if one does not exist
   A `Fact` is an object that is passed along in the flow functions. In many analyses, this is the tuple lattice.
4. Instantiate a `Dataflow` in the `Detector`
   A `Dataflow` binds together `Facts`, `DataflowAnalysis`, and a `CFG`. The `Detector` can query the `Dataflow` for facts before and after each instruction.

While this is a lot of steps, many of the existing classes can be reused. The abstractions map nicely to theoretical dataflow abstractions while allowing analyses to deviate from the standard abstractions. For example, FindBugs does not require a dataflow analysis to use a tuple lattice, they may use whatever object they like to store state from one node to the next. For these reasons, we give FindBugs 10 points for dataflow analysis.

FindBugs does not currently provide a path-sensitive analysis. However, since it provides a thin layer into the framework, it would be straightforward to create a new `Dataflow` that executes the `CFG` in a path-senitive manner. Since it would be possible to add this without changing the framework, we give FindBugs 3 points for path-sensitive analysis.

Utilities

FindBugs provides many utilities for dataflow analysis. They have created subclasses of `DataflowAnalysis` that handle forwards analysis, backwards analysis, and frame based analysis among others. FindBugs receives 10 points for providing useful utilities for dataflow analysis.

FindBugs does not currently provide a standard mechanism for analysis meta-data. While many of the existing analyses use annotations to provide better results, the support for these analyses is hardwired into the framework. FindBugs receives 0 points for meta-data support.

FindBugs does provide a mechanism for analyses to create dependencies. FindBugs will guarantee that an analyses dependencies will run first. This is very useful for finding annotations and for running interprocedual analyses, so we will give FindBugs 10 points for providing this functionality.

Deployment

Deployment of a FindBugs analysis is similar to deployment of a PMD analysis. The developer simply jars up the appropriate files and drops them in the classpath. A handful of XML files provide information about the analysis dependencies, the bug patterns found with the analysis, and error messages. There are a few problems because the developer can not simply add a new XML file, they must write append the old XML files with the necessary information. Since this is the only problem, we give 9 points for deployment.

Understanding

FindBugs provides a fair amount of documentation in the form of JavaDocs, papers, and articles. Some of the articles are a little out of date, so we will give 8 points for documentation.

FindBugs provides many analyses that a developer can use as a basis for a custom analysis. There are a few places where these analyses are directly tied into the framework though. For example, the null pointer analysis needs to find out which methods can return null based upon a previous analysis. This previous analysis stored in the information in an `AnnotationDatabase`. However, the null pointer analysis must retrieve this database from a direct call to the framework! This can confuse users who wish to create custom analyses without changing the framework. Since it has a few problems like this but overall provides many examples, FindBugs receives 8 points for sample analyses.

The best source of help for FindBugs is the user community. The FindBugs mailing list is active, and the community responds to questions quickly. The community earns 9 points for FindBugs.

Overall Evaluation

FindBugs received a final score of 827 points out of 1000. In a class, this equates to a B, and that is a fair grade of FindBugs as an analysis framework. It will work for the scenario we outlined earlier, but the largest hurdle will be using a bytecode representation instead of a Java AST. Some parts of the framework and the existing analyses need refactoring, but it will perform the necessary task and it provides some hooks for more complex analyses in the future.

6 Soot

The Soot framework is a large research framework for optimizations, code transformations, and analysis. [4] Many of the tools are intended for the compilers community, and it is not meant to be a general framework for industry. However, we will examine it since it is one of the larger and more mature analysis frameworks.

Licensing

Like FindBugs, Soot is open source and released under the LGPL. [9] Accordingly, it has the same number of points as FindBugs, 10 points for the license and 10 points for open source.

Abstract Syntax Tree

Soot represents Java source in a homegrown format called Jimple. Jimple code is somewhere between Java source and bytecode, Figure 3 gives an example of this code. Soot will read in Java files, produce Jimple files, and then run the analyses using the Jimple representation as input.

While it seems reasonable to expect an analysis developer to learn bytecode, it does seem unreasonable to learn Jimple. However, Jimple is close enough to a Java representation that the learning curve is not too bad. Therefore, we will give Soot 3 points for the mental model of the Jimple AST.

The API for Jimple provides two basic abstractions: `Unit` and `Value`. A `Unit` maps to a statement, whereas a `Value` maps to an expression, variable, or constant. There are derived classes to represent specific statements and expressions, such as `AssignStmt` and `AddExpr`. Each of these derived classes provides a specific API to retrieve internal `Unit`s and `Value`s. Additionally, the `Unit` API provides methods to get a lists of all `Value`s used and defined. This is nice because we have a specific API for when we have detailed type information, but we also have a general API for analyses that work on a higher level of abstraction. Since we get both options in Soot, it gets 10 points for the API.

There do not appear to be any ambiguities in Soot. The only confusion here is that since the optimizations and transformations run first, the Jimple files may not be what the developer expected based upon the Java source. It does not appear to be ambiguous with regards to the Jimple files themselves, so Soot receives full points for not being ambiguous.

```
TestNull meth(TestNull)
{
    TestNull this, tn, result, $r0, $r1;
    int y, ndx, i, $i0, $i1, $i2;
    TestNull[] arr;

    this := @this: TestNull;
    tn := @parameter0: TestNull;
    y = this.<TestNull: int x>;
    arr = newarray (TestNull)[5];
    ndx = 0;
    i = 1;

 label0:
    if ndx >= 5 goto label1;

    result = arr[ndx];
    $i0 = ndx - 1;
    $r0 = arr[$i0];
    y = $r0.<TestNull: int x>;
    $i1 = ndx + 2;
    $r1 = arr[$i1];
    $i2 = $r1.<TestNull: int x>;
    this.<TestNull: int x> = $i2;
    ndx = ndx + 1;
    i = i - 1;
    goto label0;

 label1:
    return this;
}
```

Figure 3: Jimple code snipet

Analysis Types

Creating an analysis in Soot is slightly unusual due to the way violations are reported. Rather than calling a framework service method that adds a violation, the developer inserts an "tag" into the Jimple abstractions.

The tool later reads the tags and converts them into violations. Since this actually changes the Jimple abstraction, the analysis is actually a transformation that happens to only add tags rather than transform other parts of the code.

To create an AST walker, the developer must derive from a `BodyTransformer` to do an intra-procedural analysis or a `SceneTransformer` to do an inter-procedural analysis. The transformer receives a callback when it starts in on a new `Body` or a scene. The transformers do not use a visitor pattern, so a `BodyTransformer` must iterate over all `Unit`s in the `Body`. Likewise, a `SceneTransformer` must iterate over all classes and then run iterations lower down. While this is not too much additional work, it does make the code more difficult to read, so we give Soot 7 points for the AST walker analysis.

To create a dataflow analysis, the developer takes the following steps:

1. Create a class derived from `Transformer`
   This class uses the dataflow analysis to add tags to Jimple.
2. Create a class derived from `FlowAnalysis`
   This class provides the flow functions and provides the lattice functions.
3. Instantiate a `FlowSet`
   This class is like only the data part of the tuple lattice. It can map nodes to data, but it does not know how to merge or copy the data.

The abstractions do not map exactly to the theoretical dataflow abstractions since they are split in slightly different places, but it is straightforward once the developer understands what each piece is reponsible for. The benefit of adjusting the split is that the developer only needs to create two classes. The FlowSet is simply a map, so there is no need to add functionality to it. We will give Soot 8 points for dataflow analysis.

Soot does not currently support path sensitive analysis, but, like FindBugs, the framework is very thin so it should be possible to add this. We will give it 3 points for path sensitive analysis.

Utilities

Soot provides only some basic utilities for dataflow analysis. It does provide forward and backward analysis, but that appears to be all. We will give Soot 6 points for dataflow utilities. Soot does not provide any support for finding existing meta-data and provides no support for analysis dependencies, so it receives no points for either of those properties.

Deployment

Soot has an unusual deployment mechanism. To add an analysis, it must be programmatically added to one of Soot's phases. The developer can do this be creating a wrapper around Soot's main method. The wrapper will simply add the analysis to a phase, then call Soot's main. As it turns out, this works because the GUI front end of Soot allows the user to change the main method to run. Soot receives 9 points for deployment since while this works, it means that we must change our main method every time we create a new analysis.

Understanding

Soot provides ample documentation in javadocs, a wiki, tutorials, and published research papers. Some of the tutorials leave out a lot of useful information, but they are good enough to get a very simple analysis running. Soot receives 9 points for documentation.

There are a few existing analyses in Soot, however, they are all either very complex or very simple. It is difficult to make the jump from the simple analysis to the next step up. Soot receives 6 points for existing analyses.

The user community for Soot is entirely researchers and students in compilers courses. The mailing list does not appear to be active, but the Soot team does maintain a list of all Soot users and their contact information. Soot receives 2 points for the community.

Overall Evaluation

Soot receives 728 points out of 1000 for its framework usability. Soot is better suited for a scenario where we also wanted to do optimizations and transformations. However, in our scenario, these extra features get in the way. The largest hindrance to using Soot in our scenario is the lack of focus in analysis. The AST is not suited for creating analyses of Java source, there is not an abundance of utilities for the kinds of analyses we would like to write, and the user community does not have an industrial or analysis focus. While Soot is extremely powerful, it is not the appropriate framework for this scenario.

7 Conclusion

We have evaluated four static analysis frameworks against an ideal framework to see if any will work with our sample scenario. The evaluation matrix we used is in Table 2. The final evaluation tells us that no framework is ideal, but FindBugs and Soot are close enough to meet our needs, whereas PMD and Crystal will not. However, this evaluation only looked at the framework usability, and not the tool as a whole. When we also evaluate the tools, we find out that Soot does not act like most analysis tools; it focuses on the Jimple files rather than on the Java files. While it might be acceptable to have the analysis writer learn Jimple, the user of the analysis should not have to know that the Jimple files even exist. This leaves us with FindBugs as the forerunner.

This result is not completely satisfactory though. While FindBugs was the best of those we evaluated, it does not meet our ideal framework. In order for customized static analysis to become viable for small to medium sized companies, we need a framework that has an AST that matches the mental model of developers so there will be little overhead to create an analysis. Without this, the developer will have to spend many hours learning a new model, and a smaller company may not have the resources to invest into this.

This work has produced a solid way of evaluating future iterations of these frameworks and others, so we hope that this matrix will guide development decisions towards making a more usable static analysis framework.

Table 2: Framework Usability Evaluation Matrix

Category	**Property (Weight)**	**Crystal (Value) Total**	**PMD (Value) Total**	**FindBugs (Value) Total**	**Soot (Value) Total**
Licensing					
	Intellectual Property (8)	(0)0	(10)80	(10)80	(10)80
	Open Source (4)	(0)0	(10)40	(10)40	(10)40
Abstract Syntax Tree					
	Mental Model (10)	(10)100	(5)50	(3)30	(3)30
	Specific API (8)	(8)64	7(56)	(10)80	(10)80
	Unambiguous (5)	(7)35	(10)50	(10)50	(10)50
Analysis Types					
	Simple AST Analysis (5)	(8)40	(10)50	(8)40	(7)35
	Dataflow Analysis (10)	(8)80	(3)30	(10)100	(8)80
	Path sensitive Analysis (5)	(0)0	(0)0	(3)15	(3)15
Utilities					
	Dataflow (10)	(2)20	(0)0	(10)100	(6)60
	Annotation (2)	(0)0	(0)0	0(0)	(0)0
	Analysis Dependencies (7)	(0)0	(0)0	(10)70	(0)0
Deployment					
	Adding Analysis (10)	(0)0	(7)70	(9)90	(9)90
Understanding					
	Existing Analyses (6)	(4)24	(8)48	(8)48	(6)36
	Docs (6)	(2)12	(10)60	(8)48	(9)54
	Community (4)	(0)0	(8)32	(9)36	(2)8
Total	**1000**	**375**	**566**	**827**	**728**

References

[1] Bcel project page. http://jakarta.apache.org/bcel/.

[2] Findbugs project page. http://pmd.sourceforge.net.

[3] Pmd project page. http://pmd.sourceforge.net.

[4] Soot project page. http://www.sable.mcgill.ca/soot/.

[5] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000.

[6] David Hovemeyer. The architecture of findbugs. In FindBugs release documents.

[7] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, December 2004.

[8] Patrick Lam. Using the soot flow analysis framework. In Soot documentation notes.

[9] GNU Orginization. Gnu lesser general public license. http://www.gnu.org/licenses/lgpl.html.

[10] Eclipse Project. Ast heirarchy. http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org tree.html.

[11] Nick Rutar, Christian B. Almazan, and Jeffery S. Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, 2004.