An Empirical Study of Object Protocols in the Wild

Nels E. Beckman, Duri Kim, and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, USA
{nbeckman,aldrich}@cs.cmu.edu, duri.kim@alumni.cmu.edu

Abstract. An active area of research in computer science is the prevention of violations of object protocols, i.e., restrictions on temporal orderings of method calls on an object. However, little is understood about object protocols in practice. This paper describes an empirical study of object protocols in some popular open-source Java programs. In our study, we have attempted to determine how often object protocols are defined, and how often they are used, while also developing a taxonomy of similar protocols. In the open-source projects in our study, comprising almost two million lines of code, approximately 7.2% of all types defined protocols, while 13% of classes were clients of types defining protocols. (For comparison, 2.5% of the types in the Java library define type parameters using Java Generics.) This suggests that protocol checking tools are widely applicable.

1 Introduction

Object protocols are rules dictating the ordering of method calls on objects of a particular class. We say that a type defines an *object protocol* if its concrete state can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition (a definition we will expand in Section 2.1). The classic example of an object protocol, often cited in research literature, is that of a file class. Instances of this file class can only have their `read` methods called while the file is open. Once the file is closed with the `close` method, subsequent calls to the `read` method will result in run-time exceptions or undefined behavior. Most popular languages do not give object protocols first-class status, and therefore cannot ensure their correct use statically.

Static and dynamic checking of object protocols is an extremely active area of research in the software engineering and programming languages communities. There have been protocol checkers based on software model checking [3, 12]. There have been type systems and flow analyses for checking object protocols [23, 8, 19, 6]. (Type-based checkers are so common that these properties are often referred to as "typestate" properties.) There have been checkers that focus on the narrower problem of object initialization [10, 22], and checkers that focus on the wider issues of framework conformance [16, 11]. There have even been dynamic checkers [14, 5], and checkers that focus on concurrent applications [4, 17].

While many of these approaches are quite powerful and their designs elegant, we argue that very little is known about how protocols are used in practice. Do they occur often or are they rarely defined? Are they used by many other classes? Are the protocols themselves simple, or complex? These are the kinds of questions we have attempted to answer with this study.

In this paper, we present an empirical study on object protocols in open source Java software. We took several popular open-source projects and the Java standard library, ran a suite of automated analyses that attempted to find evidence of object protocols, and manually investigated the results.

This work contains several contributions. As part of our investigation, we discovered that object protocol definition is relatively common (in about 7% of all types) and protocol use even more so (by about 13% of all classes). We discovered seven behavioral categories of object protocols that account for 98% of all the protocols we discovered. Compared to existing protocol studies which looked at large volumes of code [26, 27, 2], ours is the first to attempt to examine characterstics of the protocols themselves, for example frequency of definition and categories of protocols.

The paper proceeds in the following manner: Section 2 discusses the design of our experiment. This includes important definitions, description of our automated analyses, the data that we gathered and the motivation underlying our approach. Section 2.4 describes the threats to the validity of our experiment. Section 3 presents the data that we gathered during our study, and Section 4 discusses that data and its implications for other researchers.

2 Methodology

Our study proceeded in the following manner: We created a static analysis to detect patterns in source code that we believe are indicative of object protocols. Then, we ran the static analysis on popular open-source Java projects and the Java standard library. Next, we manually investigated the reports issued by the static analysis, marking each as evidence for a protocol or not. During this process, data about the location, classes involved, their super-types, and more was gathered. We also created categories of similar protocols based on our observations. Finally, we used the information about which types define protocols in order to run another automated analysis which gathered information about the usage of those protocols. An earlier version of this study was conceived by one author, Duri Kim, and presented in her masters thesis [18].

The first part of this section will discuss our definitions, namely, what are object protocols? Next we walk the reader through the experimental process, including a description of our analyses and the data we gathered. Finally, we describe the Java programs we analyzed and threats to the validity of our study.

2.1 Definitions and Scope

One of the trickiest parts of discussing object protocols is agreeing on exactly what is meant by the term. While many sanctioned interactions between dif-

ferent pieces of code could be described generically using the term protocol, we choose to focus on a definition that is based around abstract state machines. The definition of object protocol stated here sets the scope for our entire experiment. It is the idea on which our analyses and terms like "false negative" will be based.

Definition A type defines an *object protocol* if the concrete state of objects of that type can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition.

This definition contains several key ideas.

client The states of the protocol must be observable and relevant to clients.
abstract and finite The states must be abstractions of any internal representation, and there must be a finite number.
runtime transitions Methods calls on an object instance after construction will cause it to transition between abstract states.
correct use Failure of clients to obey a protocol can result in run-time exceptions or undefined behavior.

Classic examples of protocols fall under this definition. For example, an instance of the `java.io.FileReader` class can be interpreted as having two abstract states, Open and Closed. Clients must be aware of which state a given instance of the file is in otherwise they might incorrectly call a method such as `read`, which requires the file to be open, when the file is actually closed. `java.util.Iterator` fits our definition as well. Even though it is an interface and does not have its own concrete state, clients must be aware that the `next` method can only be called when a call to `hasNext` would return true.

Our definition includes initialization protocols; objects that must have certain methods called after construction to put them into a valid, *initialized* state. While these protocols may in fact be quite simple, they fit our definition, and are an important piece of the contract of many types.

We additionally include a degenerate form of protocol known as type qualifiers [13, 9]. In this case, object instances enter an abstract state at construction-time that they can never leave. Like other protocols, depending on the state the object is in, certain method calls may be illegal. We will point out type qualifiers in this study even though they do not strictly fit our definition, as we feel they are quite similar to more standard object protocols and because, like object protocols, current languages do not check them statically.

Our definition specifically excludes protocols in which a type has an infinite number of abstract states. This is meant to exclude types such as `java.util.List` on the basis of methods like `List.remove(int)`. This method throws an exception when the argument is greater than or equal to the size of the list. While `List` could be interpreted as having the abstract states, LargerThan0, LargerThan1, LargerThan2, etc., this does not fall under our definition, and will not be considered a protocol.

Scope of this Study. Our definition of object protocol leaves out other object protocols that some readers may consider to be important. For example, it does not include multi-object protocols, in which clients must call an ordered sequence of methods on two or more objects. One of the things we will show is that, even when taking a restricted view of object protocols, they are still rather common. By considering a more inclusive definition, we believe one would find that object protocols are even more common.

We have observed that protocol classes frequently are implemented so that they can detect protocol violations. Generally, violations that are detected will cause an exception to be thrown (e.g., `InvalidStateException`). This is relevant to our study because, within the scope of our definition of object protocol, our automated analysis detects the *subset* for which this is true (see Section 2.2).

Other Definitions. Here are some other terms that will be used throughout the remainder of the paper:

Phase 1 In the first phase of the study we examined the nature of protocol definition.

Phase 2 In the second phase of the study we examined protocol use.

Candidate, Candidate Code A section of code that may represent evidence of an object protocol, as reported by our static analysis.

Protocol Evidence A candidate that, after manual analysis, is determined to be evidence of an object protocol (a true positive).

Evidence Class A class that contains protocol evidence.

2.2 Experimental Procedure

Our experiment consisted of several steps where we alternatingly performed analyses, manual and automated, and gathered and processed their results. This section presents the entire process from start to finish. For convenience, this process is illustrated in Figure 1. At each step in the experiment, we will say what data is gathered and why that particular course of action was chosen.

Phase 1: Finding Object Protocols In the first phase of our experiment, we start with a set of programs in which we would like to find object protocols. The first step is to run ProtocolFinder, a static analysis that will generate a list of code candidates, locations in code that *may* indicate that a class is defining an object protocol.

We had several goals in mind when developing the ProtocolFinder static analysis. For one, we wanted to keep the rate of false negatives as low as possible. In this case, false negatives are protocols that exist in the programs under analysis that are not found. Manual inspection, of course, can have a very low rate of false negatives but is extremely time consuming, particularly considering the amount of code we would like to investigate. We desired an automated analysis. Dynamic analyses for discovering protocols in running programs exist [15, 28]. Unfortunately, such approaches are quite susceptible to false negatives, since

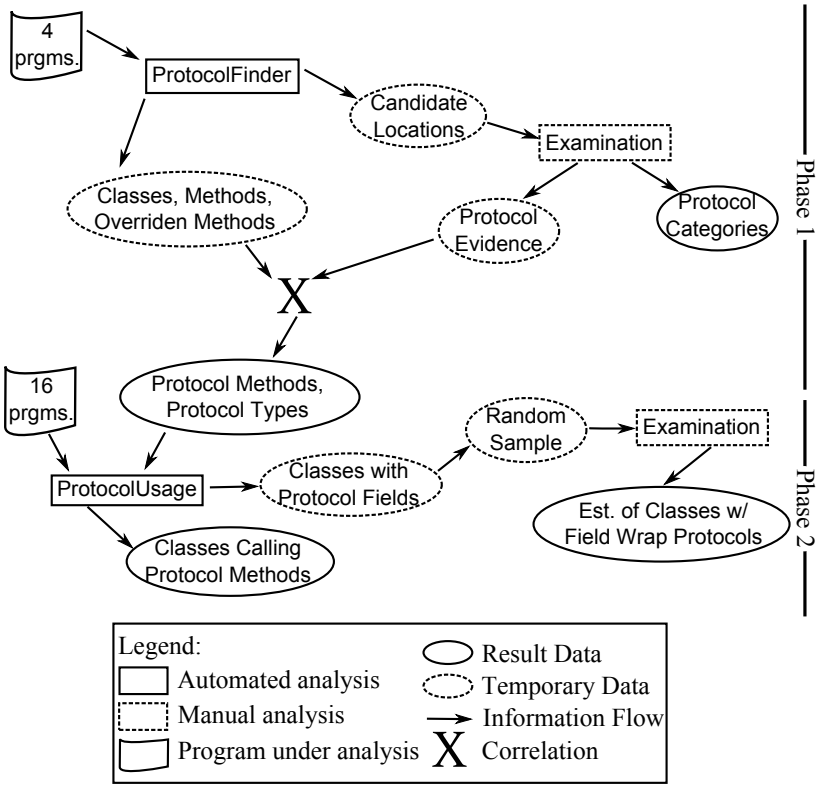

Fig. 1. A schematic explaining the experimental procedure

appropriate test cases must be found to exercise all of the possible protocols in an application. For the same reasons, a dynamic approach would require examining only programs that were accompanied by sufficient test cases, and thus, was ruled out. By comparison, a static analysis can be run on any open-source program. In the end, we decided to develop a conservative static analysis that would eliminate many (although not all) false negatives while reducing manual effort. A subsequent manual examination is used to eliminate false positives.

ProtocolFinder is a static analysis created for this study that attempts to find object protocols by searching for locations in code where protocol violations are detected. Specifically, it looks for locations in code where instance methods throw exceptions as a result of reading instance fields.

The intuition behind the analysis is simple: In our preliminary investigations we noticed that many protocol methods throw exceptions when object protocols are violated. Because our definition of object protocol depends on some abstract state of the method *receiver*, we expect that any exceptions thrown for protocol violation will be thrown in instance methods and as a result of reading an

```
1  // from java.util.concurrent.ArrayBlockingQueue.Itr
2  public void remove() {
3    final ReentrantLock lock = ArrayBlockingQueue.this.lock;
4    lock.lock();
5    try {
6      int i = this.lastRet;
7      if (i == -1)
8        throw new IllegalStateException();
9      lastRet = -1;
10     // ... method continues
11 }
12
13 // from javax.swing.undo.AbstractUndoableEdit
14 public void undo() throws CannotUndoException {
15   if (!canUndo()) {
16     throw new CannotUndoException();
17   }
18   hasBeenDone = false;
19 }
20
21 public boolean canUndo() { return alive && hasBeenDone; }
```

Fig. 2. The ProtocolFinder reports candidate code on lines 8 and 16. Both are classified as protocol evidence. In the first, the field `lastRet` flows through a local variable `i`. In the second, the field value comes from a getter.

instance field. This pattern has been noted and used as the basis for existing protocol detectors [27, 2]. Like ProtocolUsage, described later, the ProtocolFinder analysis is an Eclipse plugin whose source we have made freely available.[1]

ProtocolFinder. ProtocolFinder is a flow-insensitive static analysis that examines every instance method in a given code base. Upon encountering an 'if' block or a conditional expression, the analysis first examines the condition. If the condition expression contains a read of a field of the current receiver (or a call to a "getter" method on the current receiver), the analysis will examine both 'then' and 'else' branches. ("Getter" methods are methods which more or less immediately return the value of a field.) If either branch of the conditional throws an exception the analysis issues a report indicating that piece of code is a protocol candidate. Both the field read in the condition and the throw statement in the branches can be nested arbitrarily deeply. In order to determine whether an expression in the condition is a field read or getter call on the current receiver, the analysis queries a sub-analysis. This flow-sensitive static analysis has a list of all the methods in the current class determined to be field getters, and can track if a value in an intermediate variable flows from a getter or a field.

[1] `http://code.google.com/p/nolacoaster/`

```
1  // from java.awt.Container
2  public void remove(int index) {
3    synchronized (getTreeLock()) {
4      if(index<0||index>=this.component.size()){
5        throw new ArrayIndexOutOfBoundsException(index);
6      }
7    // ... method continues
8  }
```

Fig. 3. In the remove method the ProtocolFinder reports a possible protocol on line 5. Note that the field, component, is nested in a sub-expression of the condition (line 4). By manual examination, we have determined that this candidate is *not* evidence for an object protocol.

The analysis uses a simple procedure to determine which methods are "getter" methods. Any method with a non-void return type where all return statements contain values that flow from field reads are marked as getters.

ProtocolFinder reports protocol candidates in the examples shown in Figures 2 and 3, all of which are from the Java standard library. In Figure 2, reports are issued on lines 8 and 16. The first example comes from an implementation of the `Iterator` interface. It is noteworthy because the field value flows from the `lastRet` field to the local variable `i` before the conditional. The second example is noteworthy because the condition involves a call to the getter method `canUndo`, which itself is the result of a combination of fields, `alive` and `hasBeenDone`.

In Figure 3, ProtocolFinder reports a candidate on line 5. This example is noteworthy because the field read that occurs on line 4 is nested within a subexpression of the condition. ProtocolFinder still treats the condition as being dependent on a receiver field.

The output of the ProtocolFinder is thus a list of protocol *candidates*. In the next part of the experiment, we manually inspect each candidate to determine whether or not it is actually evidence of an object protocol. For each report issued, the ProtocolFinder includes the line number and file name of the candidate, the method and class in which the candidate was found, and all methods that are overridden by the method in which the candidate was found. This information helps us find the candidate for the purposes of manual examination, and, in the event that a candidate represents evidence of an actual protocol, will provide us with the data we need to carry out the usage phase of our study.

Manual Examination. After running the ProtocolFinder and gathering a list of protocol candidates, we investigated each candidate by hand.[2] The primary purpose of this manual investigation was to determine which candidates were actual evidence of object protocols and which were not. This was done by looking at

[2] The bulk of the work of manual examination was performed by Duri Kim, a masters student. Nels Beckman, a Ph.D. student, performed spot checks on these results.

the code location and the surrounding class and trying to understand its behavior. Where possible, documentation was also examined. After understanding the candidate and the conditions under which an exception would be thrown, we consulted our own definition of object protocol in order to determine whether or not the candidate represented protocol evidence.

As an example, consider the code snippets in Figures 2 and 3. Both were returned as candidates by the ProtocolFinder. During manual analysis, both candidates in Figure 2 were classified as evidence for actual protocols. Iterators have *RemovalPermitted* and *RemovalNotPermitted* abstract states, transitioned to and from by the `next` and `remove` methods. `remove` can only be called on instances in the *RemovalPermitted* state. `AbstractUndoableEdit` defines several abstract states but the `undo` method can only be called if an instance is both *Alive* and *HasBeenDone*. The candidate in Figure 3, on the other hand, was not categorized as protocol evidence. The exception is really thrown in response to the state of the argument, not the receiver. Even if we wanted to abstract the concrete state of the receiver to prevent the exception, the only reasonable abstraction would require an infinite number of states.

For every candidate that is manually classified as evidence of a protocol, certain information is recorded and used in the second phase of the study. For each piece of protocol evidence, we record the method in which it appears and the class in which that method appears. These classes are referred to as *evidence classes*. But we consider a larger set of types to be protocol-defining. The methods in which protocol evidence appears, and every method they override or implement are considered to be *protocol methods*. Additionally, any public method that calls a private protocol method is considered to be a protocol method (*if* we determine the private method to be part of the "state check" pattern, described below). Finally, the types declaring each of the protocol methods are known as *protocol types*. When we say in the introduction that 7.2% of types declare protocols, these are the types that we are referring to. As these terms will be used frequently in the rest of the paper, we summarize:

Protocol Methods The methods containing protocol evidence, any methods they override and, if a method containing protocol evidence is private, any public method that calls it.

Protocol Types The classes and interfaces containing protocol methods.

Our inclusion of private methods and overridden methods is worth further discussion. Regarding our inclusion of overridden methods, our logic here is that, because of subsumption, any subtype may be known statically as its supertype. When a subtype method is part of an object protocol, overridden methods are also frequently part of a protocol, or at best clients must be aware that some subtypes have usage protocols. Therefore, we want to consider calls to those overridden methods as potential client-side uses of protocols.

This strategy addresses one limitation of the ProtocolFinder, that it cannot detect Java interfaces that define protocols. If the implementing methods of an interface have behavior that the ProtocolFinder recognizes as a protocol, then the overriden interface methods will be added to our list of protocol methods.

```
java.lang.Runnable.run()
java.lang.Thread.run()
java.lang.Object.toString()
java.util.List.add(int,Object)
java.util.List.remove(int)
java.util.AbstractList.add(int,Object)
java.util.AbstractList.remove(int)
```

Fig. 4. Superclass and interface methods automatically considered to be protocol methods due to a subclass that we *removed* from our list of protocol methods. This was done because these methods are widely used, but their contracts do not imply a protocol.

In a few cases, we removed overridden methods that were added to the set of protocol methods by this process because we felt that the methods are widely used and not normally considered to be part of a protocol. For example, in the Java implementation of the Kerberos authentication protocol, the `KerberosTicket` class defines a protocol of which its `toString` method participates; if a Kerberos ticket has been destroyed, calling its `toString` method results in an `IllegalStateException`. However, `Object.toString` should not be considered a protocol method since most implementations do not have such behavior, and it is so widely used that considering it to be one would result in vastly distorted results. (In such situations, one may reasonably conclude that behavioral subtyping was broken.) Figure 4 contains the full list of supertype methods that were removed from the list of protocol methods because they do not in general represent protocols. As far as we can tell, no other widely used supertype methods were misclassified in this manner.

We included the public callers of private methods because we noticed a common pattern in many classes we encountered. Private methods cannot be called outside of the class in which they are defined and as a result will never appear as client usage in the second phase of our study. However, many classes have private "state check" methods which verify that the instance is in some particular state. These methods are called by multiple public protocol methods as a way of avoiding code duplication. For example, the `java.util.PrintStream` class defines a simple Open/Closed protocol, and once the stream has been closed, there are essentially no methods that can be called on the stream. In order to implement this without code duplication, the `PrintStream` method defines a private `ensureOpen` method that is called first thing inside every public method of the class. We want to make sure that we consider those public methods to be protocol methods, even though our analysis does not report them, so we add them when our manual analysis confirms this pattern.

During manual analysis of protocol candidates, two final pieces of data are generated. One of the goals of our study is to determine if object protocols share similar characteristics. Anecdotally, most protocols seem to be rather simple, and somewhat similar (e.g., Open/Closed, Initialized/Uninitialized) and we wanted to determine if this was generally true. While manually examining each

potential protocol, we did our best to observe similarities and group them into categories based on these similarities, using single coding. Rather than defining the categories a priori, we constructed them as new similarities were observed.

Lastly, we are very interested in whether or not protocols are used in multithreaded applications. We would like to understand the relevance of protocol checkers that work even in the face of concurrency, such as our own work [4] and that proposed by Joshi and Sen [17]. So, for each piece of protocol evidence, we recorded whether synchronization primitives (e.g., locks, monitors) were used in the surrounding code. Such use indicates that the class has been designed to be used on multithreaded programs.

Phase 2: Finding Protocol Usage In the second phase of the study, we examined how often the protocols we discovered in the first phase were actually used. The input of this phase is the list of protocol methods and protocol types generated in the preceding phase. After running an automated analysis on a suite of code, we were left with a list of all classes that called protocol methods as well as a list of all classes that have fields whose types are protocol types, and an estimate of the number of those classes that pass their fields' protocols along to their clients. The static analysis itself is rather simple.

ProtocolUsage. ProtocolUsage is a flow-insensitive static analysis. It proceeds by visiting every method call site in a given code-base. At every method call site, regardless of the receiver, the method binding is statically resolved, and the method's fully qualified name is noted. If the method is in the list of protocol methods, a report is issued, *unless* the method call site is inside the same class as the protocol method being called. Such a call would more accurately be described as an internal interaction rather than a client-provider interaction.

Note that if a class calls protocol methods of its super-class this is considered to be an client interaction with a protocol-defining class, even though at runtime there is only one object. A sub-class can validly be considered to be a client of its super-class, in the sense that a programmer extending another class must be aware of and understand the super-class's rules of use.

ProtocolUsage also looks for instance fields whose types are protocol-defining. In this part of the analysis, at every field declaration, the field's type is resolved. If this type is contained in the list of protocol types, a report is issued.

We are interested in fields of protocol type because they may potentially represent an even closer level of interaction with a protocol-defining type. Since objects referenced by fields are in the heap and may be accessed at any time by member methods, it is more difficult for programmers to obey their protocols than objects that are simply passed and returned amongst methods. Additionally, in our experience it is often the case that classes with fields that define protocols expose those protocols to their own clients.

Manual Examination. While we did not have the time to investigate all of the fields of protocol type, we did want to get an estimate of the number of classes

acting as protocol wrappers, passing on the protocols of their fields to their clients. To this end we took a random sample of the classes containing fields of protocol types (approximately 7%) and we manually investigated those classes to see whether or not the protocols of the fields were passed on to their classes. We recorded whether or not this was the case, and used the rate of protocol passing-on to get a rough estimate for the entire suite of phase two programs.

This is the end of the second phase of our study.

2.3 Programs Under Analysis

We ran the ProtocolFinder tool on four open-source programs, in order to find out how many protocols they defined. We then ran the ProtocolUsage analysis on those four plus twelve additional programs to determine how often code acts as a client to protocol-defining code.

All of the programs we analyzed in both phases are shown in Table 1, along with their sizes descriptions. With the exception of the standard library and our own analysis framework, Crystal, they all come from the Qualitas Corpus [24]. We attempted to select relatively large, popular open-source programs, and to have a mix of library/framework software as well as end-user applications. By choosing a wide variety of programs we reduce the risk that the programs we analyzed contain abnormally many (or few) protocols. The desire to include both libraries/frameworks and end-user applications is based on our own intuition. We hypothesized that libraries and frameworks are more likely to define types with object protocols since they may be wrapping some underlying system resources that is inherently stateful (e.g., sockets and files).

During the course of the study we examined 1.9 million lines of Java, of which 1.2 million was used in the first phase of the study, and of that portion, one million of which is the Java standard library. Examining the Java standard library for object protocols was a high priority. Because of its wide use in most Java programs, knowing which types in the standard library define protocols enables us to analyze client usage of protocols in many more programs. In fact almost all of the client-side protocol usage in our study was usage of standard library types. This makes sense since, for example, Ant is unlikely to use any protocols defined in PMD, Azureus or JDT and we do not know any of the protocols it defines, since it was not part of the first phase of our study.

2.4 Risks

There are a number of potential risks and threats to validity in the study as designed. Here we discuss some of those risks, as well as mitigating factors.

Some of the most interesting risks in our study are due to our use of static analysis. The use of static analysis is motivated by our desire to examine as large a corpus of programs as possible. Unfortunately, this means the study is subject to the false negative and false positive rates of our static analysis, particularly the ProtocolFinder. For the ProtocolFinder, false negatives are instances where the analysis is run on a piece of code that defines an object protocol and yet the

Table 1. The programs analyzed as part of this study, along with their sizes and descriptions

Program	L/F or App.	Version	LOC	Classes (Interfaces)	Description
Phase I: Programs analyzed for protocol definition and usage.					
JSL	L/F	jdk1.6.0_14	1,012,860	8,485 (1,761)	Java standard library
PMD	A	3.1.1.0	26,586	396 (27)	Static analysis
Azureus	A	3.3.2	102,119	900 (354)	BitTorrent client
Eclipse (JDT core)	L/F	3.3	99,691	300 (41)	IDE Framework
Phase II: Additional programs analyzed for protocol usage.					
ant	A	1.7.1	91,679	962 (71)	Build tool
antlr	A	2.7.7	41,880	186 (35)	Lexer/parser tool
aoi	A	2.5.1	81,597	438 (26)	3D modeler
columba	A	1.0	68,267	982 (109)	GUI email client
crystal	L/F	3.4.1	17,052	187 (66)	Static analysis framework
drjava	A	20050814	59,114	639 (79)	Educational IDE
freecol	A	0.7.4	62,641	434 (21)	Civilization clone
log4j	L/F	1.2.13	13,784	178 (16)	Logging library
lucene	L/F	1.4.3	25,472	276 (15)	Text search library
poi	L/F	2.5.1	47,804	417 (28)	Microsoft document library
quartz	L/F	1.5.2	22,171	121 (25)	EJB scheduling framework
xalan	L/F	2.7.0	161,008	1,004 (65)	XSLT XML transformation engine
Total	8×A 8×L/F		1,933,725	15,905 (2,739)	

L/F=Library or Framework A=Application

analysis does not report a candidate. False positives are the protocol candidates that are not classified as protocol evidence. False positives are mitigated by manual inspection. Every candidate reported by the ProtocolFinder has been manually inspected to determine whether or not it represents protocol evidence.

However, we can imagine several potential sources of false negatives. The first source is that the ProtocolFinder can only investigate code, and that code must be written in Java. This rules out protocols that are defined by Java interfaces, which contain no code, and native methods, which are written in other languages. We mitigate the former case with our inspection process: when a method is determined to be a protocol method, we note the supertype methods it overrides and add them to our list of protocol methods for use in subsequent phases of the study. For native methods, though, there is not much that we are able to do.

Still, of the 120,085 methods we analyzed in the first phase of the study, only 739 were native methods, suggesting that we might not be missing much.

Another source of false negatives comes from code that does not attempt to detect protocol violations, in other words, protocol-defining code that does not fit the pattern that the ProtocolFinder is looking for. The ProtocolFinder requires code to check or use the value of a receiver field inside a conditional expression and then throw an exception in one branch of the conditional. APIs that fail in an undefined manner when their protocols are violated likely would not fit this pattern. As an example, consider a class defining an initialization protocol, which will throw a null pointer exception if its methods are called before initialization due to null fields. Such a protocol would likely not be detected by our analysis. We believe that well-designed code will generally attempt to detect violations of its own protocols. However, this scenario is likely a source of real false negatives.

Similarly, APIs that define protocols due to their delegation to other, protocol-defining APIs may be missed by our ProtocolFinder. For example, an enhanced stream that wraps another stream, and delegates calls may define a protocol that is very similar to the underlying stream. While we do not have a direct way of finding these protocols, we are attempting to gage how likely they might be by reporting the number of classes whose fields themselves define protocols. Then, based on a manual examination of a sample of those classes, we estimate the number of unexamined classes that delegate the protocols of their fields.

Lastly, we have the typical threats of any empirical study: that our selection of programs may be biased, not representative, or too small to draw meaningful conclusions. We have done our best to draw a variety of programs from a respected corpus of popular Java programs [24] that was as large as possible given our time constraints.

3 Results

In this section we present the results of our study[3], with little additional discussion. Discussion of the results is postponed until Section 4. The results of running the ProtocolFinder analysis are discussed in Section 3.1, categories of protocols we found are discussed in Section 3.2 and the results of running the ProtocolUsage analysis are discussed Section 3.3.

The summary is that a little over 2.2% of all classes on which we ran our ProtocolFinder define protocols. 7.2% of all types are considered to define protocols when we include supertypes, and approximately 13.3% of all the classes on which we ran our ProtocolUsage analysis use object protocols as clients. 98% of the protocols we found fit into one of seven simple categories.

3.1 Protocol Definitions

Table 2 contains the results of running the ProtocolFinder analysis on the four code bases in phase one. The first column contains the number of candidates

[3] All the data gathered during this study can be found at the following location: `http://www.nelsbeckman.com/research/esopw/`

Table 2. The results of running the ProtocolFinder on the four phase one code bases

Program	Protocol Candidates	Protocol Evidence	E.C.	P.T.	T.S.E.C.	Precision	%E.C.	%P.T.
JSL	2,690	613	195	842	54	22.8%	2.3%	8.2%
PMD	32	7	3	10	0	21.9%	0.8%	2.4%
Azureus	136	24	19	32	4	17.6%	2.1%	2.6%
JDT	62	4	4	5	0	6.5%	1.3%	1.5%
Total	2,920	648	221	889	58	22.2%	2.2%	7.2%

T.S.E.C.=Thread-Safe Evidence Classes E.C.= Evidence Classes
P.T.= Protocol Types

reported by the ProtocolFinder analysis. These varied from around 2,600 for the Java standard library to 32 for PMD. The next column shows how many candidates were manually classified as protocol evidence. The next column shows the number of classes containing protocol evidence, followed by a column showing the number of types classified as protocol types. (Recall that our list of protocol types includes classes and interfaces containing methods overridden by methods containing evidence of protocols.) Next, "Thread-Safe Evidence Classes" displays how many classes containing protocol evidence use mutual exclusion. Since we are interested overall in how well our analysis is performing, the next column shows the precision of the ProtocolFinder: the ratio of protocol evidence to protocol candidates. The last two columns show the percentage of classes containing protocol evidence relative to the total number of classes and the number of protocol types relative to the total number of types. The last row displays cumulative values for each column, and percentages recalculated from these sums.

3.2 Protocol Categories

Of the 613 candidates in JSL that were manually determined to be protocol evidence, we noticed a number of similarities in their structure and intent. In fact, almost all of them could be characterized in one of seven protocol categories, which we will describe in this section. Due to the means by which our analysis produces candidates, the categories we present are largely categories of errors: conditions under which operation of a class will result in an error. Table 3 summarizes the results for each category. More details on each category can be found in Duri Kim's masters thesis [18].

Initialization (28.1%) Some types must be initialized after construction time but before the object is meant to be used. In the *initialization* category, calls to an instance method m after construction-time will result in an error unless an initializing method i has been called at least once before. Types may (or may not) allow i to be called multiple times, however, it is a feature of this category that objects cannot become uninitialized after they have already been initialized (i.e., initialization is monotonic).

A typical example of this category is the protocol defined by the Java library class `AlgorithmParameters` in the package `java.security`. After an instance of algorithm parameters is constructed, it is not ready for use until one of its three `init` methods is called. Before initialization, calls to the `toString` method will return null, and calls to `getEncoded` and `getParameterSpec` throw an exception.

Deactivation (25.8%) Some types permit deactivation, after which point instances can no longer be used. In the *deactivation* category, calls to an instance method m will fail after some method d is called on the same instance, and it will always fail for the rest of the object's lifetime. Like *initialization*, types may or may not permit d to be called more than once.

A typical example is the `BufferedInputStream` in the package `java.io`. Once a stream is closed, no further methods can be called on the stream, and it cannot be reopened. A somewhat more interesting example is `FreezableList` from `com.sun.corba.se.impl.ior`. This is a normal mutable list that, at some point during its lifetime, can be made immutable by calling the `makeImmutable` method. After this point mutating methods, like `remove`, can no longer be called. (This is in direct contrast to other immutable lists, like those created by `Collections.unmodifiableList`, which are immutable for the entire object lifetime.)

Type Qualifier (16.4%) Some types disable certain methods for the lifetime of the object. In the *type qualifier* category, an object instance will enter an abstract state S at construction-time which it will never leave. Calls to an instance method m, if it is disabled in state S will always fail. This category is so-named since it is similar in spirit to flow-insensitive type-qualifiers [13, 9].

Protocols in this category show two distinct behaviors. In some cases, the abstract state that newly constructed instances inhabit can be set by parameters to the constructor. For example, instances of the `ByteBuffer` type in the `java.nio` package may or may not be backed by a byte array. Whether or not they are depends solely on whether or not a backing array was provided at construction-time. If one was not provided, any calls to the `array` method will fail with a run-time exception. In other cases, the instantiating class itself determines the abstract state that all instances will inhabit, relative to the abstract states defined in a super-type. For example, consider the instances returned from calls to `Collections.unmodifiableList` in the Java standard library. All such instances are *unmodifiable* relative to the super-type `List`, which permits both mutable and immutable lists. In both case, clients must be aware of which methods are enabled.

Dynamic Preparation (8.0%) Certain methods cannot be called until a different method has been called to ready the object. In the *dynamic preparation* category, an instance method m will fail unless another instance method p is called before it. If we think of types in this category as having two states, *ready* and *not ready*, this category is distinguished from the initialization category in that an object may dynamically change from ready to not ready at numerous points in its lifetime (i.e., it is not monotonic).

The most familiar example of this category is the `remove` method on the `Iterator` interface. An iterator's contract states that the `remove` method cannot be called until `next` has been called, and clients must continue to call the `next` method at least once before each time the `remove` method is called.

Boundary (7.9%) Some types force clients to be sure that an instance is still "in bounds." In the *boundary* category, an instance method m can only be called a dynamically-determined number of times. Calling m more times will result in an error. Typically such types will provide some method c to clients so that they can determine if an subsequent call to m is safe, although clients are not required to call it. We can abstract this into a finite number of states by having *is in bounds* and *isn't in bounds* abstract states.

A widely known example of this category is the iterator. In an iterator, the `next` method can only be called if the iterator is at a location in the iterated collection where there are subsequent items. Iterators provide the `hasNext` method allowing clients to check dynamically if this is the case.

Redundant Operation (7.3%) In the *redundant operation* category, a method m will fail if it is called more than once on a given instance.

For an example of this category, consider the `AbstractProcessor` class, located in the `javax.annotation.processing` package. If the `init` method is called more than once, the second call will fail. One might wonder, given the name of the method, why this is not considered to be part of the initialization category. The answer has to do with the fact that our categories are oriented towards errors. In the initialization category, methods on an object will fail if the object has not already been initialized. Here, the failure occurs when the `init` method is called a second time.

Domain Mode (4.8%) The *domain mode* category captures protocols for objects that very closely model a domain. In these objects, various "modes," which are domain-specific, can be enabled and disabled, which in turn cause certain methods related to those modes to be enabled or disabled.

As an example, consider the `ImageWriteParam` class in the `javax.imageio` package. An image may be written with or without compression. `ImageWriteParam` objects control whether and how compression is used for other image objects. The `ImageWriteParam` class defines several compression modes, "no compression," "explicit," and "writer-selected." The parameter's `setCompressionType` method can only be called when the parameter is in "explicit" compression mode.

Others (1.9%) Finally, there were a smattering of protocols that did not fit any of the previously-mentioned categories, although even these protocols themselves have certain similar characteristics. As examples, we encountered a few instances of types that defined methods that must be called in strict alternation (a single call to method A enables a single call to method B and vice versa). We also found a limited number of protocols that we would describe as *lifecycle*

Table 3. Categorization of each of the 648 reports issued by the ProtocolFinder that were evidence for actual protocols.

Category	Protocol Evidence	%
Initialization	182	28.1%
Deactivation	167	25.8%
Type Qualifier	106	16.4%
Dynamic Preparation	52	8.0%
Boundary	51	7.9%
Redundant Operation	47	7.3%
Domain Mode	31	4.8%
Others	12	1.9%

methods, where a type defines more multiple abstract states through which an object transitions monotonically during its lifetime. For example, the `GIFImageWriter` and `JPEGImageWriter` classes in the Java imageio library seem to have this behavior. While we did not encounter many lifecycle protocols, our own experience with Object-Oriented frameworks suggests that they may be more common elsewhere.

3.3 Protocol Usage

Table 4 shows the results of running the ProtocolUsage analysis on the sixteen candidate programs from phase two of the study. The goal here is to see how often classes act as clients of other protocol-defining types. The table contains the following information: The first column after the list of programs is the number of classes in that program that contain calls to protocol methods. The next column shows the percentage of classes in each program that use protocol methods. These numbers range from 4% of all classes using protocols, on the low end, to 28% of all classes on the high end. The next two columns show the number and percentage of classes that have fields whose types are protocol-defining types. The column, "Exposes Protocol Rate" shows the percentage of the classes with protocol fields that were found to expose the protocols of those fields to their own clients, of the 7% of classes with protocol fields that we sampled. The column, "Est. Classes From Total" is an estimate of the total number of classes that expose protocols defined by their fields based on this rate. The last two rows show the totals and cumulative percentages for the entire suite, as well as the numbers excluding the Java standard library.

We were also interested in finding out which protocol methods were being called most frequently, and Table 5 summarizes this information. This table contains a list of the fifteen most frequently called protocol methods. During our examination of the sixteen open-source code bases used in phase two, we found 7,645 calls to protocol methods. We took all the protocol methods that were called, and ordered them by how many times they were called. Table 5 shows the 15 most frequently called protocol methods along with the number of times

Table 4. The results of running the ProtocolUsage analysis on the sixteen candidate code bases.

Program	Classes Calling Protocol Methods	%	Classes w/ Prot. Fields	%	Exposes Protocol Rate	Est. Classes From Total
JSL	1012	12%	1082	13%	15%	157
PMD	85	22%	29	7%	0%	0
Azureus	198	22%	763	8%	31%	234
JDT	13	4%	18	6%	0%	0
ant	269	28%	187	19%	20%	37
antlr	20	11%	16	9%	0%	0
aoi	25	6%	37	8%	0%	0
columba	120	12%	246	25%	8%	18
crystal	9	5%	2	1%	0%	0
drjava	49	8%	107	17%	0%	0
freecol	94	22%	117	27%	0%	0
log4j	39	22%	32	18%	0%	0
lucene	30	11%	27	10%	0%	0
poi	41	10%	13	3%	100%	13
quartz	16	13%	10	8%	0%	0
xalan	91	9%	142	14%	13%	17
Total	2111	13%	2141	13%	17%	356
W/O JSL	1099	15%	1059	14%	18%	196

that method was called in our candidate programs and the percentage of the 7,645 protocol method calls that particular method constitutes. For example, the `next` method of the `Iterator` interface was the most-frequently called protocol method in our study. Of the 7,645 calls to protocols we found, over 2,200 were calls to `Iterator.next`, almost 30% of the calls.

4 Discussion

After running our experiment, we noticed some interesting results. Protocols were defined with small, but significant, frequency and almost all of those protocols fit within a small number of categories. All of the protocols we expected to find we did find, which gives us some confidence in our approach. And a significant number of classes in our study use protocols as clients, even though almost all of the protocols we were looking for were defined in the Java standard library. Interestingly, but not surprisingly, there are a few protocols that are much more widely used than others.

4.1 Sanity Check

As discussed in Section 2.4, we were curious about the ProtocolFinder's false-negatives: protocols that were defined in the code under analysis but not discovered due to the design of the analysis. One quick sanity check we can do is

Table 5. The 15 most-frequently called protocol methods, out of a total of 7,645 calls to protocol methods, and percentage occurrence of each method relative to the total.

Method	Calls	% Calls
java.util.Iterator.next()	2226	29.11%
java.util.Enumeration.nextElement()	1022	13.37%
java.lang.Throwable.initCause(Throwable)	850	11.12%
org.w3c.dom.Element.setAttribute(String,String)	460	6.02%
java.util.Iterator.remove()	211	2.76%
java.io.Writer.write(int)	182	2.38%
java.io.OutputStream.write(int)	165	2.16%
java.io.InputStream.read()	162	2.12%
sun.reflect.ClassFileAssembler.cpi()	138	1.81%
org.omg.CORBA.portable.ObjectImpl._get_delegate()	90	1.18%
java.io.InputStream.read(byte[],int,int)	89	1.16%
java.util.ListIterator.next()	80	1.05%
java.io.Writer.write(char[],int,int)	77	1.01%
java.io.PrintWriter.flush()	76	0.99%
java.io.OutputStream.flush()	75	0.98%

to make sure that all the protocols we already know about are found by our analysis. This is not perfect, since our ProtocolFinder was designed with these protocols in mind. Still, it is somewhat comforting to see that all of the protocols we have encountered in our own work, and in similar works are found by our analysis.

We expected to see sockets, files, streams and iterators in our results, since those types are widely discussed in related work. And with the exception of the actual `java.io.File` class, which does *not* define a protocol, we were not disappointed. Socket, Readers, Writer, Streams and all their related classes did turn up in our analysis. (Interestingly, `ZipFile` does define an Open/Closed protocol.) We were also previously aware of the `Throwable` and `Timer` protocols.

Additionally, we were happy to see that well-known protocol-defining interfaces, like `Iterator`, were discovered through our process, since, for interfaces, the ProtocolFinder has no code to examine.

4.2 Widely Used Protocols

We were quite interested, although not surprised, by the fifteen most frequently called protocol methods, shown in Table 5. The iterator protocol, examined in several recent works [6, 20], appears at the top of the list, and the `next` method of the iterator protocol accounts for nearly a third of all protocol method calls.[4]

[4] It is worth noting that the `hasNext` method, which we would generally consider to be part of the Iterator's protocol, does not show up at all in our list of protocol methods. This is due to the fact that the implementations of `hasNext` do not normally partake in protocol violation detection by throwing an exception.

While this seems rather uninteresting, it does suggest two points. One, that the time spent evaluating protocol checkers against the iterator interface may be well-spent, since a good iterator-checker can check a large portion of the protocols that are used in practice. Second, all of the calls recorded are actual calls to `Iterator.next`, and not instances of Java 5's enhanced for loop. While at present, these do represent actual protocol uses, where the client needed to understand the Iterator's protocol in order to use it, one suspects that many of these calls could be replaced by the enhanced for loop, which would dramatically reduce the number of protocol clients we observed. (The same cannot be said for calls to `Iterator.remove`.)

The remaining frequently called methods quickly drop off in the frequency of their use. The most-frequently called list leaves something like forty percent of all protocol method calls off. This suggests that most protocols, like most APIs in general, have a small number of clients. Most of the commonly used protocols are quite recognizable: readers, writers, streams and certain collections defining abstract states. Interestingly, when we remove recognizable types (e.g., streams, sockets, files, iterators, throwables and their subclasses) we found that what was left accounted for 21% of all protocol usage. This means there is still a fair amount of use of non-obvious protocols.

4.3 Protocol Categories

We were pleasantly surprised to discover that a small number of categories (seven) could be used to classify almost all of the protocols that we encountered (98%). This is useful because it suggests a new evaluation criteria for developers of typestate checkers. Unless a typestate checker can verify protocols from each of these seven categories, it is unlikely that it will work on most practical examples. Of course, many of the interesting challenges in protocol verification come from the context in which the protocols are used, for example whether or not the relevant objects are aliased [7]. Still, these categories can help to guide analysis evaluation.

It is also interesting that the categories produced during this study have the flavor of "protocol primitives," and this may have something to do with how the study was carried out. To illustrate, one may have noticed that none of the categories that we found have more than two abstract states. Yet this does not mean that none of the types we investigated had more than two abstract states. Our study proceeded by investigating each location of interest as determined by the ProtocolFinder. We tried to understand only enough of the implementation to determine whether or not we were seeing evidence for a protocol, the state that the class should be in in order not to have that particular exception thrown, and which state the class is in if the exception is thrown. But classes can have different pieces of a protocol that fit into different categories or even multiple protocol pieces that are all in the same category. As an example of the latter case, consider the `Socket` class in `java.net`. A socket instance can be open or closed, its "write-half" can be open or shut down. Both aspects of the protocol

are categorized as *deactivation check* protocols, but if one to is to consider the class' protocol in total, it would have at least four abstract states.

All of this is to say that there may be interesting characteristics shared by protocol-defining types that are not captured by our categories. Coming to a better understanding of protocols at a larger level of granularity, while an interesting topic for future work, is out of the scope of this study.

4.4 Other Observations

A number of other points can be made by examining the results of our study. First, object protocols are relatively common. Without context, 7.2% of the types analyzed may not sound like an enormous amount, but consider that, according to a simple analysis, just 2.5% of the 10,246 types in the Java library define Java "Generic" type parameters, a widely heralded new feature of the language.

One point suggested by the data is that protocol use (13% of all classes) is more common than protocol definition (7.2% of all types). This information suggests that client-side protocol checking may be more important than implementation-side checking. Certain protocol-checking approaches have the ability to verify both the correct use of protocols by clients and the correct implementation of protocols by their providers. Such is the case for the approach presented by Bierhoff and Aldrich [6]. While provider-side checking may be important in some situations, a good client-side protocol checker may give programmers the most bang for the buck.

In Table 4 we showed that 13% of all classes have fields whose types are protocol types. From the 7% of those classes we manually examined in our random sample, 17% of them were found to expose the protocols of their fields to their clients. Extending this rate to the entire set of classes with protocol fields, we estimate that something like 356 of the classes in the phase two programs define object protocols simply because of the ways in which their fields must be used. This represents about 2% of all of the classes we examined in the entire study, and, if accurate, is a significant increase in the percentage of protocol types.

Of all the classes defining protocols, the percentage implemented with synchronization primitives was significant. Out of 221 classes containing protocol evidence, 58 of them, or 26.2% were designed to be accessed by multiple threads concurrently. If protocol checking is considered an area of research interest, this suggests that those checkers should be designed with multi-threading in mind.

We did not observe conclusively that protocols were more likely to be defined by libraries and frameworks than by applications. However, the Java standard library when considered separately, has a much higher percentage of its types classified as protocol-defining (8% vs. approximately 2%). There could be some truth to the idea that code wrapping underlying system resources is more likely to define protocols. However, given our process of gathering protocol types, it might alternatively suggest that the standard library has a deeper type hierarchy.

For protocol usage, there was some difference observed. In programs that we classified as applications, 17.4% of classes acted as clients of protocol-defining methods. For library and framework code, that rate was 11.4%.

We were interested in the variety of types that define protocols. As evidenced by the small number of protocol categories, these protocols were often quite similar, but in fact the contexts in which they were defined vary greatly. This answered one of the questions that helped to motivate this study: Are there *any* protocol types beyond files, sockets and iterators? We can say, confidently, that the answer is yes. The following list shows just a few of the examples we found:

Security `com.sun.org.apache.xml.internal.security.signature.Manifest`, `java.security.KeyStore`
Graphics `java.awt.Component.FlipBufferStrategy`, `java.awt.dnd.DropTargetContext`
Networking `javax.sql.rowset.BaseRowSet`, `javax.management.remote.rmi.RMIConnector`
Configuration `javax.imageio.ImageWriteParam`, `java.security.AlgorithmParameters`
System `sun.reflect.ClassFileAssembler`, `java.lang.ThreadGroup`
Data Structures `com.sun.corba.se.impl.ior.FreezableList`, `java.util.Vector`
Parsing `net.sourceforge.pmd.ast.JavaParser`, `org.eclipse.jdt.internal.compiler.parser.Scanner`

4.5 Future Work

Our study suggests a number of potential avenues for future work. For one, the simple static analysis, ProtocolFinder, developed for this study, while useful, is not sound with respect to our own definition of object protocol. Better analyses will likely find even more protocol definitions in the same code base. Alternatively, widening the definition of object protocol to include more object behaviors, will also likely result in finding more object protocols in the same code base, and a wider definition may be of interest to certain researchers.

As discussed in Section 4.3, our current protocol categories are in some sense "micro-categories:" primitive categories from which larger behavioral patterns might emerge. An interesting task for future work is to examine these larger behavioral entities to see if they share common characteristics.

Finally, even if object protocols are common, an interesting question to ask is whether or not they lead to program defects. Studying the correlation between protocol definition and use in a code base and the quality of that code may help to answer this question.

5 Related Work

The problem of finding classes that define protocols is one of protocol inference, and there has been some work in this area. The two most closely-related studies were done by Weimer and Necula [26] and Whaley et al. [27].

Weimer and Necula [26] performed a study on open-source software that in some ways is similar to ours. In their work, they were looking for violations of resource-disposal protocols. For example, a connection to a database that *must*

be closed eventually, ideally as soon as it is no longer needed. They examined over four million lines of open-source Java code and found numerous violations of these sorts of protocols. This study, while quite interesting, differs from ours in a number of ways. First off, their focus was on finding violations of protocols rather than characterizing the nature and use of protocols (correct or otherwise) as we have done. While they did look for protocol violations, they made no systematic attempt to discover automatically the types that define such protocols. Rather, they started their experiments with a known list. Additionally, their notion of protocol and our notion of protocol do not quite overlap. They consider protocols to be instances on which some operation must eventually be performed. The protocols we consider, protocols in which calling a method at the wrong time will lead to an error, are not considered in their work.

Both Whaley et al. [27] and Alur et al. [2] have developed effective tools for statically inferring protocol definition. Whaley et al. [27] present a dynamic and a static analysis for inferring object protocols. Their static analysis is inspired by the same reasoning that ours is, and the description contains an in-depth discussion of the practice of "defensive programming," which is what we have described here as detection of protocol violations. The dynamic analysis they propose can infer more complex protocols than the static analysis. While our experiments cover some of the same ground as theirs (both examine the Java standard library) our focus is different. Their primary focus is on the analyses themselves, with the frequency and character of the protocols taking a back-seat. Their largest studies were performed using the dynamic analysis, and so in some ways are not comparable since not all of lines of code are executed during dynamic analysis. Our best estimate is that their study covered approximately 550 thousand lines of source, compared with 1.2 million lines of source covered in phase one of our study. Numbers are only reported for the Java standard library experiment. They report that 81 of 914 classes define protocols. Our experiments for version 1.6.0_14 report that 195 of 8,485 classes define protocols, and show how much the Java standard library has grown since version 1.3.1! Still, their work contains some discussion of the relevant methods and interesting features of these object protocols. Our work contains a more systematic description of the protocols encountered, including a classification of those protocols. Lastly their static analysis seems to be more precise. It can detect protocol violations that result in null pointer exceptions, which ours cannot.

Alur et al. [2] propose a related static protocol detector that also seems to be more precise than ours. They also looked at the Java standard library, albeit just a handful of classes. While either of these static analyses might have made a better candidate for our own study, neither are publicly available.

In fact, a large number of other approaches have been proposed for automatically inferring object protocols, both static and dynamic (e.g., [1, 25, 15, 28], Pradel et al. [21] give a good overview). While a more precise static analysis may help the accuracy of our findings it does not affect our overall conclusions. It is our position that dynamic inference is inappropriate for our needs, since using these analyses requires, at a minimum, test cases to exercise parts of code that

use protocols. In our attempt to find as many protocols as possible in as much code as possible, finding test cases has proved to be quite difficult.

6 Conclusion

In this paper we presented an empirical study that examined several popular open-source Java programs. The goal was to determine the true nature of object protocols; how often they are defined, how often they are used, and in what way those protocols are similar. In order to examine as much code as possible, which can help us draw broad conclusions, we developed two static analyses, ProtocolFinder and ProtocolUsage, which help us find where protocols may be defined and where they are used. ProtocolFinder in particular may be subject to false negatives, but regardless was able to find many of the most commonly discussed object protocols.

We found that object protocols are occasionally defined (on average, 7.2% of all types were found to define protocols) but more commonly used (on average, 13% of classes acted as clients of protocols). A small number (seven) of rather simple protocol categories were used to classify almost all of the found protocols.

Acknowledgments Support provided from ARO grant #DAAD19-02-1-0389 to CyLab, and CMU|Portugal Aeminium project #CMU-PT/SE/0038/2008.

References

1. M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07*, pages 25–34. ACM Press, 2007.
2. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL '05*, pages 98–109. ACM Press, 2005.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, pages 103–122. Springer-Verlag New York, Inc., 2001.
4. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA '08*. ACM Press, 2008.
5. K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *ESEC-FSE '05*, pages 217–226, Sept. 2005.
6. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07*, pages 301–320. ACM Press, 2007.
7. K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, pages 195–0219, July 2009.
8. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5):59–69, 2001.
9. J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL '04*, pages 281–292. ACM Press, 2004.

10. M. Fahndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA '07*, pages 337–350. ACM Press, 2007.
11. G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA '06*, pages 75–88. ACM Press, 2006.
12. S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.
13. J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.
14. M. Gopinathan and S. K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *OOPSLA '08*, pages 245–260. ACM Press, 2008.
15. A. Heydarnoori, K. Czarnecki, and T. T. Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *ECOOP '09*, pages 344–368. Springer-Verlag, 2009.
16. C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *ECOOP '09*, pages 27–51. Springer-Verlag, 2009.
17. P. Joshi and K. Sen. Predictive typestate checking of multithreaded Java programs. *ASE '08*, pages 288–296, Sept. 2008.
18. D. Kim. An empirical study on the frequency and classification of object protocols in Java. Master's thesis, Korea Advanced Institute of Science and Technology, 2010.
19. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004.
20. N. A. Naeem and O. Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08*, pages 347–366. ACM Press, 2008.
21. M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *ICSM '10*, 2010.
22. X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL '09*, pages 53–65. ACM Press, 2009.
23. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
24. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010. Corpus version 20090202r.
25. A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07*, pages 35–44. ACM Press, 2007.
26. W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. *SIGPLAN Not.*, 39(10):419–431, 2004.
27. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02*, pages 218–228. ACM Press, 2002.
28. H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *ECOOP '09*, pages 318–343. Springer-Verlag, 2009.