Searching the State Space: A Qualitative Study of API Protocol Usability

Joshua Sushine, James D. Herbsleb, and Jonathan Aldrich
{sunshine, jdh, aldrich}@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA

***Abstract*—Application Programming Interfaces (APIs) often define protocols — restrictions on the order of client calls to API methods. API protocols are common and difficult to use, which has generated tremendous research effort in alternative specification, implementation, and verification techniques. However, little is understood about the barriers programmers face when using these APIs, and therefore the research effort may be misdirected.**

To understand these barriers better, we perform a two-part qualitative study. First, we study developer forums to identify problems that developers have with protocols. Second, we perform a think-aloud observational study, in which we systematically observe professional programmers struggle with these same problems to get more detail on the nature of their struggles and how they use available resources. In our observations, programmer time was spent primarily on four types of searches of the protocol state space. These observations suggest protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently perform state search.

I. Introduction

Application Programming Interfaces (APIs) often define *protocols* — restrictions on the order of client calls to API methods. These protocols are common: more than three times as many types in the Java Standard Library define protocols as define type parameters [2]. Protocols can also be complex: ResultSet from the Java database connectivity (JDBC) library contains 33 unique states dealing with different combinations of openness, direction, random access, and insertions [3]. Protocols also cause significant pain: for instance, in a study of problems developers experienced when using a portion of the ASP.NET framework, three quarters of the issues identified involved temporal constraints such as the state of the framework in various callback functions [16]. Finally, protocols are poorly supported by mainstream languages and tooling — the state of practice is to specify protocols with documentation, implement them with low-level language constructs, and react to violations with exceptions.

All of the factors just mentioned have spurred a tremendous number of research projects aimed at improving the usability of API protocols. There have been many tools and languages designed to specify and verify protocols. Strom and Yemini [28] proposed *typestate* as a compiler checkable abstraction of the states of a data structure. The Fugue system later integrated typestates into an object-oriented programming language [8]. Many tools verify protocols (e.g. Bierhoff et al. [4], Dwyer et al. [9], Foster et al. [11]). These tools require programmers to specify protocols using alias and typestate annotations separate from code. To automate the annotation, many tools mine protocol specifications from program executions or static analysis. A recent survey of automated API property inference techniques uncovered 35 inference techniques for ordering specificationss [25].

This massive research effort has gone on despite the fact that very little is known about precisely what problems programmers have when using APIs with protocols. In this work we attempt to answer four research questions which we hope will provide more solid guidance for future researchers:

- **RQ1** What are the characteristics of protocol tasks that are difficult for programmers?
- **RQ2** How do programmers approach protocol tasks?
- **RQ3** What information do programmers seek and have difficulty locating while performing protocol tasks?
- **RQ4** What resources do programmers use while performing protocol tasks?

To answer these questions, we performed two studies of professional developers. The first study identifies real-world phenomena, and the second investigates the heart of those phenomena in more detail.

In the first study, we searched the popular developer forum, Stack Overflow, for questions related to known APIs with protocols. We then winnowed, analyzed, distilled, and merged the resulting questions into a list of distinct protocol-specific tasks. These tasks represent real protocol programming challenges and we noted five common characteristics in answer to RQ1.

In the second study, we brought seasoned professional programmers into the lab and observed them performing the tasks uncovered by the forum mining. To answer RQ2, we analyzed the transcripts to categorize the activities that programmers performed. Information seeking dominated programmer effort and we therefore noted the information the developers sought while performing the tasks and how they sought it. We found that developer time was spent primarily on state search. We also found that developers debugging protocol violations looked first to the documentation related to the method call occurring at the exception location to solve their problems. These findings address RQ3 and RQ4.

II. Protocols

This paper intends to investigate API protocols, but the term protocol is widely used with conflicting or ambiguous definitions. In this paper, we focus on object-oriented APIs and we borrow the precise definition from Beckman et al. [2, p. 4]:

> 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.

The focus of this definition is on state machines. An object with a protocol must have a finite number of states which are abstractions over concrete internal representations. These states are visible and relevant to API clients. An object transitions between abstract states when particular methods are called at runtime. Clients programs that do not comply with a protocol will cause the API to throw an exception, operate incorrectly, or fail to operate at all. We had this definition in mind while conducting both of the studies discussed in this paper. Therefore, all of the API protocols we studied conform to this state-based definition.

III. Related work

The studies we discuss in this paper focus on the usability of API protocols. This works builds on many recent studies of more general programming obstacles. Two classes of studies have particular relevance to this paper. The first class, which we will refer to as *information needs studies*, includes mostly-qualitative studies that are often conducted in the field. They investigate what information developers look for in their work, how they look for it, and the purpose of the information. The second class of studies, which we will refer to as *API usability studies*, are mostly quantitive and are usually conducted in the laboratory. They investigate the usability of particular APIs, or more recently the usability of API design choices. There is not space to discuss all of the examples in either class. Instead we will delve deeply into a few examples in each class (and one gap-bridger) to highlight important lessons for this paper and motivate the study's design.

A. Information needs studies

In an oft-cited example of an information needs study, Ko et al. [18] observed 17 Microsoft developers as they performed their regular work. During the study, participants searched for information 334 times, which the experimenters abstracted into 21 categories. The abstracted categories are all very high-level, reflecting the breadth of the activities performed. For example, the most common category was "did I make any mistakes in my new code?" Two categories identified by Ko are particularly relevant to protocols: 1) "What code causes this program state?" — A programmer using an API protocol needs to understand how an object transitions to a particular abstract state.[1] 2) "In what situations does this failure occur?" — Debugging a protocol violations requires understanding when a particular method call is invalid. Our studies expand on these results, discovering in more detail when question like these arise and what kinds of state information are needed.

Other studies, also information needs studies, have narrowed the developer tasks slightly to delve more deeply into specific topics. For example, Sillito et al. [26], like Ko, studied professional programmers in their work environments. However, instead of studying whatever the programmers happened to be working on, the programmers were asked to select an "involved" software change task, and never "a simple fix." Silito observed programmers pursued an answer to a higher-level question "by asking a number of other, lower-level questions." Sometimes the programmers even asked the low-level question first and built up to the higher-level question. Our studies will investigate which low-level questions are most useful in learning to correctly use API protocols.

LaToza et al. [20] brought programmers to the lab and asked them to contribute architecture-level design improvements to a 54KLOC open source tool. They noted several high-level differences between experts and novice participants: novices focused more on symptoms of problems, experts on sources; novices spoke in terms of specifics, and experts in terms of abstractions; novices wasted more time understanding implementation details, while experts' focus was wider. Again, these results are interesting and contribute to our general knowledge, but are of little direct utility to most language and tool designers.

Robillard and DeLine [24] surveyed and interviewed Microsoft developers about the obstacles they faced when they last learned to use a public API. The most common obstacles mentioned involved documentation. More particularly, the answers suggested five problematic issues commonly found in API documentation: design intent, code examples, matching APIs with use cases, penetrability, and formatting/presentation. Many of these issues were simply missing from documentation, (e.g. no discussion of performance characteristics), mistargeted (e.g. examples of inapplicable usage), or buried (e.g. most method documentation contains boilerplate repetition of information contained in the method signature).

B. API usability studies

The more traditional API usability studies observe programmers in the laboratory while they use APIs. In most of these studies, the participants performed tasks that were selected by the experimenters as "representative of typical use" of the API. McLellan et al. [22] were among the first to publish a study of a particular API, and they are also credited with spreading the recognition that "the techniques and theory developed for usability should be applied directly to the API" [6]. McLellan's study uncovered many low-level difficulties with the API under investigation, but more importantly for the purpose here, agreed with Robillard about the importance of code examples and documentation.

McLellan's study and those like it are primarily useful for the designers of the API under investigation. To provide guidance to

[1] Ko addressed this category with the Whyline tool [17].

designers of future APIs, Jeff Stylos and colleagues performed a series of studies to evaluate API "design choices" [10, 29, 30]. In Ellis et al. [10], the experimenters compared the usability of constructor-based instance creation with instance creation using a factory method or abstract factory [12], which Ellis refers to collectively as the "factory pattern." The study used both within and between subjects comparisons and found that users required much more time to instantiate objects when the API used the factory pattern rather than constructors.

The design choice studies provide data-driven design guidance, but it is difficult to abstract principles from them. For example, the Ellis study does not provide insight into why it is harder to use the factory method pattern than a constructor.

C. Discussion

The two studies we report in this paper lie between the two classes discussed above. The studies in this paper, like those in the first class, are qualitative and focus on the information needs of developers. Unlike the other information needs studies, we focus on a particular programming domain — API protocols— to add detail and richness to our existing general knowledge so that it can be used for tool building.

Our think-aloud laboratory study shares many elements with the studies in the second class. However, our tasks were mined from developer forums and we therefore expect the study to be more connected to practice. Finally, the laboratory study was not looking for quantitative results like the design-choices studies, nor specific issues with the APIs like the McLellan-type studies. Instead, the results of the second study are principles and understanding which we hope can be applied to any API with protocols. Our follow-on work, which validates this paper's conclusions, evaluates state-structured documentation using programming experiments that are similar to the McLellan-type studies [32].

IV. Forum Mining

We mined Stack Overflow, a widely-used developer forum, primarily to identify the characteristics of protocol tasks that are difficult for programmers (RQ1). We discuss the strengths and weaknesses of StackOverflow data in Section IV-A. We downloaded the entire Stack Overflow database which is freely available to anyone under a Creative Commons license. When this study was conducted there were 2.6 million questions on Stack Overflow. This is far too many to read and digest, so we winnowed the question list with the techniques we discuss in Section IV-B. The goal of the filtering was to focus our efforts on questions that were likely to be protocol-related and significant. Once we had a reasonable-sized list of questions, we manually read each questions to: 1) determine if the question was protocol-related, 2) distill a task, and 3) merge with existing tasks. This study's aim is to characterize recurring protocol problems, but does *not* attempt to estimate commonality. The study also required a lot of manual labor, so we likely excluded many protocol question for the sake of efficiency. The strategies we used in all of these efforts are discussed in Section IV-C.

Fig. 1. Screen snap of the StackOverflow question page.

The most frequent and interesting characteristics of protocol-related questions are discussed in Section IV-D.

A. Strengths and weaknesses of forum data

Forums provide a window into developer practice that is particularly well suited to mining examples. Asking a question on a forum requires significant effort — it requires composing a question, extracting relevant code or documentation, and describing important context. After asking a question, the answers do not come immediately, so developers often wait to post questions until they have struggled for a while. Therefore, the questions usually contain distilled problems of practical significance.

We chose to use Stack Overflow for its wide use, feature set, and openness. Stack Overflow is the most popular developer forum on the web and it therefore contains questions in a uniquely broad set of categories. Parnin and Treude [23] found that StackOverflow covered 84% of the methods in the JQuery API. This was important for us because it allowed us to distill a wide-range of protocol-related tasks. A sample question page with important highlighted features is shown in Figure 1.

According to Mamykina et al. [21] Stack Overflow is also the fastest forum on the web, with median answer time of only 11 minutes. This speed encourages posting on low-level topics, which includes most protocol issues, since questioners can expect a fast answer. Mamykina credits the popularity primarily to the engagement of the Stack Overflow designers with the user community. In addition, the feature set, which includes a "reputation score" earned for asking well-liked questions or providing well-liked answers, incentivizes use [33]. All viewers of a question can categorize the question with a "tag," which helps programmers determine question relevance. Of particular importance to this effort is that questioners are rewarded for "accepting" an answer, which often gives the most important clue about the real problem the questioner

faces. For example, the code search and recommendation tool Example Overflow uses these social features to the determine quality and relevance of programming examples contained in StackOverflow questions. [36].

Despite the numerous benefits of forum questions as a data source, and Stack Overflow in particular, the questions there are by no means representative of all programming problems. Vasilescu et al. [34] found that women are substantially less likely to participate in Stack Overflow than men. Furthermore, women that did participate were less likely to participate heavily or earn reputation points. More generally, Kuk [19] found that forum participants act strategically in a number of ways including by helping those who are likely to reciprocate and by seeking out career advancement opportunities. This strategic behavior results in a question and answer pool that is largely authored by a heavily active elite. Finally, the quality and difficulty of StackOverflow questions vary dramatically [13]. Therefore, one cannot count questions of a certain type to gauge commonality of that type. In summary, Stack Overflow is a useful resource for finding real-world programming problems but the participant and question population is not representative, nor are the questions sets directly comparable.

B. Winnowing the Question List

We wanted tasks that both are protocol-related and caused problems for real developers. Therefore, we started by assembling a list of 109 Java Standard Library classes that contain a protocol. The bulk of the classes are listed in two studies, Beckman et al. [2] and Whaley et al. [35], that identified protocols via semi-automated static analysis. Neither Beckman nor Whaley identified any protocols in interfaces, so 9 interfaces were added from other sources (e.g. Bierhoff et al. [4]). These interfaces are not implemented in the Java Standard Library, but they are implemented by many third parties, and so the interface protocols can be very widely used.

We downloaded a data dump from Stack Overflow that contained questions and answers that were created through March 2012.[2] We discarded 40 of the classes because their protocols were very familiar and simple. In particular three protocol patterns were removed: 1) Boundary protocols in which a method named next or starting with next (e.g. nextInt) cannot be called after the end of an underlying list (e.g. java.util.Iterator). 2) Deactivation protocols in which many methods cannot be called after the close method is called (e.g. java.util.Scanner). 3) Redundancy prevention protocols in which the cause of a Throwable or Exception cannot be set more than once[3]

We then searched for questions about each of 69 remaining classes systematically, to ensure that later analysis was done fairly. The SQL queries used are described in detail in Sunshine [31].

For most classes the search returned fewer than five related questions, and only nine had more than 100. In order to include only well-used APIs in the results, we focused our efforts on these nine classes.[4] We examined all of the questions and answers related to these nine classes, looking for protocol-related questions. We discuss how we determine if a question is protocol related in detail in Section IV-C. Of the nine, five had protocol-related questions: URLConnection, its close cousin HttpURLConnection, Timer, ResultSet, and Socket had protocol-related questions. The results in this section are drawn from questions related to these classes.

C. Analyzing a Question

We manually examined a total of 5,039 questions related to nine classes. The first order of business was to eliminate questions that were unrelated to protocols. The single fastest heuristic we used was to examine how the search keyword was used in the post. The keyword was often found in an import statement, method return type, type of an unused variable or argument, comment, throwaway reference, etc. but never used again. This phenomenon was especially common in cases where long code blocks were attached to a question for context. The vast majority (more than 90%) of questions were discarded by this heuristic alone. For example, in question #5302656, "java.sql.ResultSet" appears exactly once in a list of possible types of values accepted by a particular value. In question #2609535, ResultSet only appears in an import statement and is never used.

If the keyword heuristic did not eliminate a question, we examined the question more thoroughly. We focused on the accepted answers to questions, the exception types and error messages described, and searched all text and code for protocol violating methods. More details can be found in Sunshine [31, ch. 3].

Excluding questions. If none of the protocol violating methods appeared and none of the earlier strategies were useful, then we excluded the question from the study. It is therefore possible we incorrectly excluded questions this way, especially if the protocol issue was not in code but buried in difficult to parse prose. However, the large number of questions required us to be expedient. The goal of the study was not to estimate the commonality of protocol problems, but to characterize recurring patterns—which justifies the expediency.

Brute force. In rare instances, none of the above strategies worked. These instances usually included large blocks of code with many method calls and exceptions. When none of the earlier strategies worked, we carefully read the full text of the post, including all the answers, to understand the problem or problems faced by the questioner.

Distillation. If a question was found to be protocol related, we then distilled a concrete protocol-based task from the

[2]This was the latest data dump available at the time this part of the study was conducted.

[3]In unpublished experiments conducted by Ciera Japan, tasks involving these protocol patterns were very simple for expert developers, but still challenging for novices. It seems that experts have memorized or otherwise internalized the steps needed to use these libraries correctly. In these experiments, experts completed tasks involving these patterns very quickly and the observations therefore yielded little insight.

[4]We cutoff questions with fewer than 100 questions because 100 is a round number and there was a sizable gap in the data at that point. All of the APIs included in the final study had a minimum of 210 questions.

question being asked. We focused our efforts on discovering the particular difficulty the programmer had with the protocol. Protocols are composed of rules, and in most cases, the programmer violated one of these rules. In these cases, the distillation involved identifying the specific rule that was violated. We excluded all domain specific information from the task. For example a Timer running on Android is the same as a Timer running on a PC.

D. Results

After completing the winnowing, analysis, and distillation we selected 28 Stack Overflow questions. We merged these 28 question into 13 distinct topics. The results are summarized in Table I. The most common distilled question was about the violation of a protocol rule. There were 23 such questions and these were merged into nine topics, one for each distinct protocol violation (marked "Cannot" in the table).

Three questioners confused two rules that compose the protocol. These three questions represent two distinct confusions and they were therefore merged into topics (marked "Confusion" in the table). Finally, two questioners requested the APIs add a new protocol-related feature. These were distinct and therefore represent two topics (marked "Wanted" in the table). In both cases, the questioners requested state-tests, which we will discuss further in the next section. All of the questions, except in two topics, asked for help debugging a protocol violation.

1) Characteristics: The questions and corresponding topics had five common and interesting characteristics that we highlight here. These characteristics address RQ1, "what are the characteristics of protocol tasks that are difficult for programmers?" In each case we discuss the evidence for each characteristic in the data and then discuss its significance. After all the characteristics are introduced, we discuss the significance of the full collection.

Missing state transition. Many questioners hoped for or assumed a state transition that the protocol did not allow. For example, questioner #4278917 explicitly asks if there is a method that allows a client to "disconnect" and thereby reuse a URLConnection (there is none). Similarly, one way of looking at all six questions about rescheduling a TimerTask, is as a question about the ability to transition the TimerTask from the scheduled to the virgin state. Finally, two of the questioners trying to call scrolling methods on a forward-only specifically looked for a method to transition that ResultSet to the scrolling state. Documentation is particularly ill-suited to addressing this type of question. It often requires a global search of all of the method and class documentation to discover that a transition is not available.

State tests. For three of the four libraries, questioners asked for a method to test the abstract state of the object. The state test questions for Timer (#13880202) and URLConnection (#7614408) are listed in Table I. In addition, questioner #2741276 requests a method to test if a ResultSet has been closed. However, this question was not included in the results because an isClosed method was added in Java 6. Presumably, the questioner was using an earlier version of Java. There were no similar questions about Socket, but for good reason — Socket includes state tests for every state it defines.

State independence. In some cases, objects with protocols can occupy multiple states simultaneously. For example, a ResultSet object, whose UML state machine is shown in Figure 2, occupies the *and-states* Direction and Position simultaneously. State transitions on and-states act independently, and this independence confused several questioners. For example, the connectedness and openness of a socket are independent. Questioner #3701073, perhaps unsurprisingly, thought that a closed socket could not be connected, but this is incorrect. Similarly, the four forward-only questioners did not seem to understand that the act of calling a scrolling method did not change the Direction state.

Multi-object protocols. All four of the APIs we looked at closely inspired questions about the relationship to other APIs. For example, a ResultSet object is closed if the Statement object that created it is closed or reused. Four questioners in the sample struggled with this one issue (4646561, 4864920, 5840866, 10118129). Questioners also asked about the following other relationships: Timers with threads, Sockets with data streams, and URLConnections with Sockets. We did not include these multi-object protocol issues in the primary results to focus on the vast majority of protocol-specific tooling that does not support multi-object protocols.

Terminology Confusion. Many of the questioners seem to be confused by terminology. This type of confusion is extremely common and not protocol-specific. However, the frequency of its appearance in the data warrants a brief discussion. Questioners often assumed a particular definition for a term, and when the definition was wrong they struggled. For example, questioner #9497100 assumed that canceling a TimerTask would *always* abort the Task. The questioner therefore tried to cancel the task in the task's own run method, in a failed attempt to halt execution immediately. Other questions misinterpreted Socket.isConnected, Timer.schedule, and URLConnection.inputStream.

Discussion. All of the characteristics just highlighted, except terminology confusion, are protocol-specific. This suggests that protocol-targeted tooling or languages may be necessary to improve the usability of API protocols.

The challenge of missing state transitions suggests that documentation should include a list of state transitions in an easily digestible form. This would enable programmers to quickly learn which transitions are, and are not, available. The very existence of state test questions suggests the usefulness of state tests. Josh Bloch, the designer of much of the Java Standard Library including several of these classes, suggests that all APIs with protocols "should generally have a state-testing method indicating whether it is appropriate to invoke state-dependent method[s]." [5, p. 242].

That repeating occurrence of multi-object protocols in the forum mining data buttresses the evidence collected by Jaspan and Aldrich [14] that multi-object protocols are important. Therefore, this study motivates the those working on relationship types [15, 1]. Unfortunately, many protocol-targeted tools

TABLE I
LISTS THE APIS, QUESTIONS AND MERGED TOPICS DISCOVERED IN THE FORUM MINING.

API	Topic	#Qs	Question IDs
URLConnection	Cannot: Set request property after connected	2	331538, 5368535
	Cannot: Reuse connection	1	4278917
	Wanted: IsConnected state test	1	7614408
Timer	Cannot: Reschedule TimerTask	6	1041675, 1801324, 4388353, 6813654, 7631542, 8404736
	Cannot: Change Scheduled time of TimerTask	4	5014132, 6555583, 6762099, 8173147
	Confusion: Timer.cancel() vs. TimerTask.cancel()	2	1801324, 6477608
	Cannot: Cancel running TimerTask	1	9497100
	Wanted: State Test for TimerTask	1	13880202[a]
Socket	Confusion: Closed vs. Connected	1	3701073[b]
ResultSet	Cannot: Read after end	1	3502005
	Cannot: Call next on InsertRow	3	4874574, 6684753, 9836972
	Cannot: Call scrolling methods on forwardonly	4	6367737, 6871641, 8032214, 9007051
	Cannot: Read before calling next()	1	8039233

[a] This question was discovered after the forum mining, but matches all of the criteria used to select the other questions.
[b] This is the only Socket protocol question, but as of Sep. 2013 it had the highest reputation score in this table, suggesting its importance.

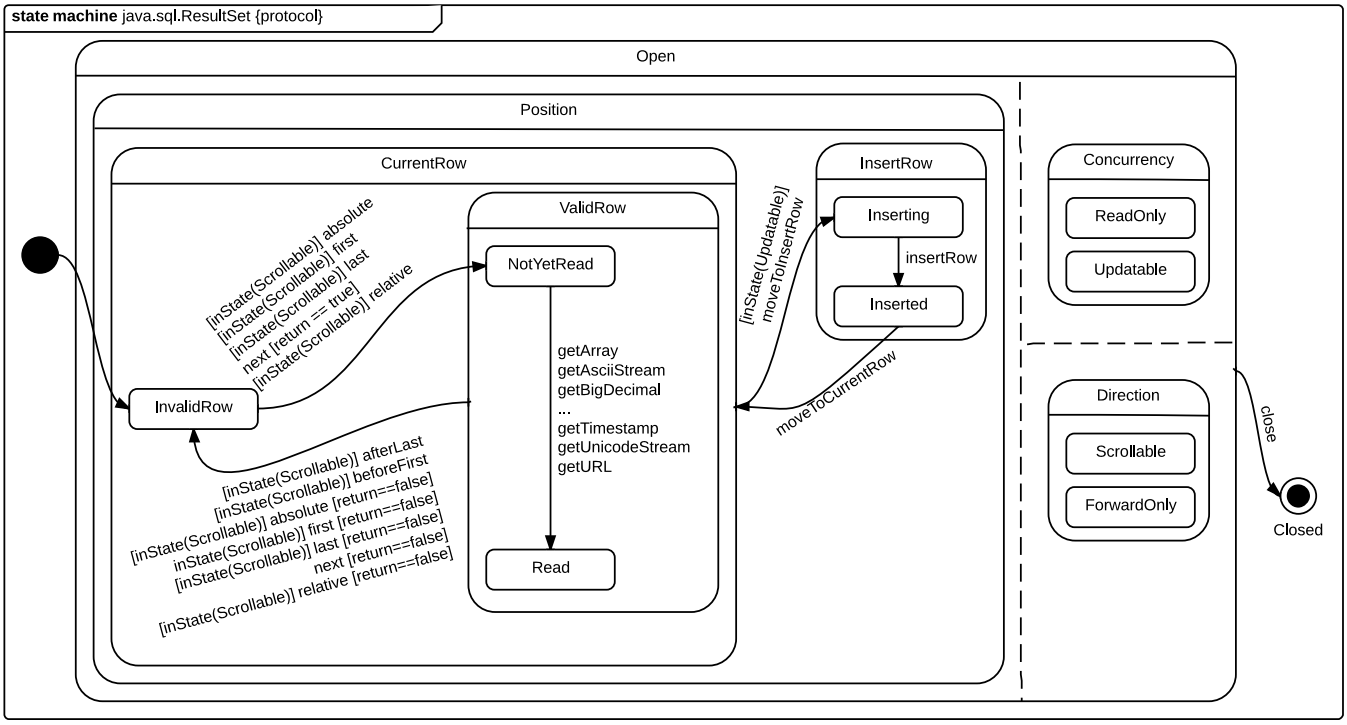

Fig. 2. UML State Machine for ResultSet.

do not support and-states. The data suggests and-states are particularly problematic, which in turn suggests that these tools are missing an opportunity to address an important usability challenge. Finally, the prevalence of terminology confusion, suggests that API protocol designers should carefully name state-related methods to ensure that the standard English definition matches its use in the protocol.

These characteristics share one significant weakness with the source from which they were derived. Each forum post represents a snapshot of a single programmer's thinking. It is difficult to know whether these characteristic problems are challenging for most programmers or just a tiny minority. Similarly, it is difficult to know what common programmer challenges were missed because they were resolved before a question was ever asked. Finally, and most significantly, the forum mining has given us a better idea of what is hard, but we still need to understand why they are hard. What do programmers do when trying to address these tasks? Why are their tools and documentations inadequate? We address these weakness in the laboratory observations we discuss next.

V. LABORATORY OBSERVATIONS

In this section, we describe the methodology and results of the laboratory study. The aim of this study is to learn how

programmers approach protocol tasks (RQ2), with particular focus on the information they seek (RQ3) and the resources they use (RQ4). In this study, the tasks are taken from the forum mining and therefore connected to practice. We discuss how we transform the topics mined from Stack Overflow into tasks in the next section. We then discuss the study design. Next, we highlight observations from one particular task—inserting a new row into a ResultSet—which we will use to illustrate the important results from this study. Finally, we summarize the results from all of the tasks including quantitative and qualitative analysis.

A. Methodology

1) Topics to tasks: We converted each of the topics uncovered by the forum mining study, as summarized in Table I, into a corresponding programming task. The tasks were derived from the code contained in the topical question(s). The tasks did not include project context such as package names, or code that was not protocol related. Each task included instructions and a method annotated with pre and post-conditions. The source files are available on the web.[5] In some cases, a test case is included with the task to trigger the bug. This was necessary whenever the method was passed a Socket, TimerTask, ResultSet, or URLConnnection instance.

The code in the method body was most commonly taken directly from one of the questions related to a topic. However, some topics required more creativity because the questions did not include code. For example, the state-test related questions did not contain code which motivated the questioner's need for the state test. Therefore, we created tasks that required knowledge of the state. These tasks each involved writing a method which takes a Timer or URLConnection instance as an argument and uses the instance in a state-specific manner.

2) Example task: To understand better how tasks were constructed, let us look at an example task in more depth. We focus on a task corresponding to the topic "Cannot: Call next on InsertRow." The task involves inserting a new row in a database table via a ResultSet instance and then trying to call the next method.

The ResultSet protocol prohibits scrolling (e.g. calling the next method), while the "cursor is on the insert row." To understand this better, let us look at the state machine diagram show in Figure 2. The cursor position is modeled by the abstract state Position. The Position state has two or-children, CurrentRow and InsertRow, which represent the state of the ResultSet when the cursor is on existing row or on the insert row respectively. Note that the method moveToInsertRow transitions the ResultSet from the CurrentRow state to the InsertRow state. In reverse, the method moveToCurrentRow transitions the object back to the CurrentRow.

A slightly abbreviate version of the code participants were given is shown in Listing 1. Programmers were asked to fix a bug, revealed by a test case, in the insertHarryBovik method.

[5]http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/qualitative-study-tasks.zip

```
1  /**
2   * Precondition: rs is a CONCUR_UPDATABLE ResultSet
3   * to an attached table with at least one row and String
4   * columns labeled, ''first'' and ''last''
5   *
6   * Postcondition: Insert a new row with ''Harry'' in the
7   * ''first'' column and ''Bovik'' in the last column. Update
8   * next row's last name to ''Carnegie''.
9   */
10 public void insertHarryBovik(ResultSet rs) {
11     rs.moveToInsertRow();
12     rs.updateString("first", "Harry");
13     rs.updateString("last", "Bovik");
14     rs.insertRow();
15     rs.next();
16     rs.updateString("last", "Carnegie");
17     rs.updateRow();
18 }
```

Listing 1. Source code for example task.

In particular, running the test case results in an SQLException when the next method is called on line 15.

To fix the bug participants needed to add just one line in the code. Before calling next, the ResultSet needs to be transitioned to the CurrentRow state by calling the method moveToCurrentRow. As we will see in Section V-B1, this task was surprisingly difficult even for the expert programmers performing the study.

The rest of the tasks have a similar flavor. They require programmers to write new small programs or fix existing small programs involving protocols. All require programmers to navigate the state machine of an underlying object.

3) Study design: We have found that protocols are very challenging for novice programmers or programmers without significant experience using object-oriented libraries and frameworks written in statically typed languages. Therefore we recruited 6 programmers with at least 3 years of professional experience with Java or C#. However, these programmers had never used any of the particular libraries under evaluation. The programmers were recruited via personal contacts.

Participants performed the tasks in a campus laboratory. They worked with a computer that had been prepared with Eclipse and a browser opened to the relevant JavaDoc. Participants were asked to "think aloud." The analysis of this study relies on correctly interpreting what participants were looking for while performing the tasks. Therefore, we followed Ko et al. [18] and asked "what are you looking for?" when participants forgot to think aloud, or their statements were unclear. Participants screens and speech were recorded. The study itself took between 1 and 3 hours, almost all of which was spent performing programming tasks. Task instructions were read to each participant and also provided in written form.

B. Results

In this section, we discuss the results of our observations. These observations address RQ2, "How do programmers approach protocol tasks?" We first describe detailed observations

from one particular task and then present the aggregate results from the full study.

1) Example task observations: We introduced the ResultSet insertion task the participants performed in Section V-A2. This task was the most time consuming for the participants — time to completion ranged from 16 minutes to 49 minutes. In addition, the participant observations of this task illustrate well the major results we will discuss in the next section.

Recall from Section V-A2 that participants are debugging a protocol violation. In particular, the next method is called while the ResultSet's cursor is on the insert row. However, none of the participants immediately knew this was the source of the problem.

All participants immediately read and interpreted the error message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown." Most participants articulated a rapid-fire set of questions about the details of the error message: e.g. "What is FETCH NEXT?," "Why is the cursor position unknown?" The participants seemed to leave these questions unanswered and focus on the beginning of the error message, "invalid cursor state." The participants recognized that this was protocol related and they asked one of two questions: "What is the cursor status of [ResultSet] rs?" (4 participants) or "Which cursor state does rs need to be in to call next?" (2 participants). As we will discuss later in detail later in this section, these two questions are instances of common question categories.

Regardless of the question asked, all six participants looked first at the method documentation for the next method to see if it could help them answer their question. Unfortunately, the next method documentation does not answer either question. Three participants noted that the documentation states that a SQLException is thrown "if a database access error occurs or this method is called on a closed result set." All three immediately decided neither cited source was the cause of the bug in this case.

The participants' searches diverged from this point forward. Three general categories of searches were used: linear scan of task lines, linear scan of method documentation search, undirected/random search through class documentation.

The fastest strategy, employed by two participants, was to look at the method documentation of each method in the source code one by one. They started at next (line 15) and moved upward to insertRow, then updateString,[6] and finally to moveToInsertRow. These participants looked at the documentation by hovering over the method name inside the Eclipse code editor. This strategy is reasonably natural in an IDE that supports hover documentation, but would require constant switching between editor and webpage documentation if a more traditional editor is used.

The fourth sentence of the ResultSet documentation for moveToInsertRow helps participants identify the state that the result set is in: "Only the updater, getter, and insertRow methods may be called when the cursor is on the insert row." All 3 participants that read this documentation articulated a new understanding of the exception message and articulated a follow up question. One participants said, "Aha! The cursor is on the insert row. How do we get the cursor off the insert row to call next?"

Fortunately for the participants that reached the moveToInsertRow documentation the answer to the follow-up question was immediately evident. To call next, one must call moveToCurrentRow, which both has a parallel name and appears after moveToInsertRow in the documentation.

One participant read the method documentation in the order they appeared on the JavaDoc webpage (the previously discussed participants scanned in the order they appear in the task code), which was the slowest search strategy. This participant looked at the next documentation in the Javadoc generated web page. On the web page, the next method appears first in the Method Detail list. The order of the method documentation matches the order that methods appear in the ResultSet source code. The participant scanned all of the documentation between next and moveToInsertRow which represents 2240 lines of the ResultSet source code and more than 100 methods. Thankfully, much of it is repetitive and could therefore be skimmed. After reaching the moveToInsertRow documentation, this participant acted similarly to the task line searchers.

The remaining three participants, like the method documentation scanner, read the next documentation on the web page. From there these participants skipped around somewhat randomly on the webpage. All three of these participant read at least a few irrelevant sections of method documentation. However, these three eventually found themselves at the top of the webpage at the class level documentation. The penultimate section of this documentation provides a code example that "moves the cursor to the insert row, builds a three-column row, and inserts it into rs and into the data source table using the method insertRow."

After reading the example, the participants compared the example code to the buggy code and noticed the missing call to moveToCurrentRow in the buggy code. The participants read the method documentation for moveToCurrentRow before adding it to insertHarryBovik. One explained he was "trying to figure out if you could call next on the current row?" The observations from this task are illustrative of the aggregate results we discuss next.

2) Aggregate results: To address RQ3 ("What information do programmers seek and have difficulty locating while performing protocol tasks?"), we transcribed the audio recordings, noting the time of every statement made or question asked by the participants. We will refer to anything the participant says as a *quote*. We then watched the video recording and mapped these quotes to blocks of time. Whenever we believed the activity on screen was motivated by a quote, we assigned the block in which it was performed to the quote. This mapping allows me to estimate how much time was spent on each quote.

[6]One participant actually moved down to the updateRow documentation before proceeding upward again to updateString. However, the strategies were otherwise identical.

In the vast majority of cases, the mapping was based on simple temporal ordering — if the activity was performed during or after quote A and before any other quote it was assigned to quote A. In a small number of cases, an activity did not seem to match the preceding quote, and therefore the activity left unassigned. This phenomenon was rare because the experimenter usually noticed when this happened and asked the participant to explain his or her actions. In total, we assigned 87% of participant time to a quote.

We then performed open-coding [27] on the quotes, looking for similar quotes that tended to repeat. Four categories of quotes were particularly common. Each of these categories represents a state search task. In total, 82% of the assigned time (or 71% of the total time) was spent working on the following four categories of search. We list here each general category followed by two specific instances of that category drawn from the transcripts:

A What abstract state is an object in?
- "Is the TimerTask scheduled?"
- "What is the cursor state of [ResultSet] rs?"

B What are the capabilities of an object in state X?
- "Can I schedule a scheduled TimerTask?"
- "What can I do on the insert row?"

C In what state(s) can I do operation Z?
- "When can I call doInput?"
- "Which ResultSets can I update?"

D How do I transition from state X to state Y?
- "How do I get off the insert row to the current row?"
- "Which method schedules the TimerTask?"

These search problems are all specific to protocols, and therefore the protocol tasks are dominated by state search.

To clarify the coding process, consider the two instances of category A listed above. The instance, "Is the TimerTask scheduled," contains the name of an abstract state of TimerTask, "scheduled," so that part of the instance was generalized to "state X." "The TimerTask" refers to a an object so that part of the question was generalized to "an object." Therefore, the question was first coded as "Is the object in state X?" In the second instance, "What is the cursor status of [ResultSet] rs," the "cursor status" refers to the state of the ResultSet. This instance maps directly to "What abstract state is an object in?" The code for the first instance was later merged into this more general category.

Many concrete questions are compositions of several categories. Answering, "What do I need to do to the conn to set doInput?" requires answering general questions C and D. The method doInput can only be set in the disconnected state (C), and the only way to get a disconnected connection is to create a new connection (D). Similarly, answering "What methods can I call on [the object referenced] by [variable] conn?" requires answering a combination of A and B.

We break down the questions and time spent in Figure 3. These charts break down only the 71% subset of time spent on state search activities. As you can see, the only combination categories that appeared in the quotes were A+B and C+D.

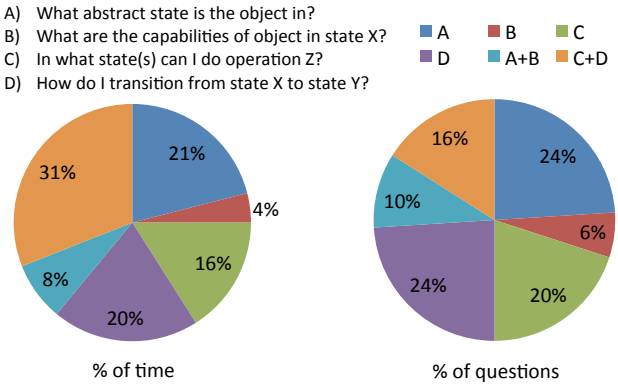

Fig. 3. Question type frequencies.

It's possible to come up with other combinations (e.g. B+D: "I wonder what would happen if I find a transition to state Y?) but harder to envision how they would be useful.

The question types appeared with almost equal frequency, except for category B which was relatively infrequent. We expect category B, which is relatively exploratory, to be more useful in greenfield tasks than the tasks in this study.

A reader who compares the two pie charts will observe that the category C+D questions were relatively time consuming (31% of time was spent on 16% of questions). This relationship held for all 6 participants—C+D questions had the highest average time spent for everyone. When category D questions occur alone, it is possible to guess the method name that will transition the object to the wanted state. To give one trivial but common example, if the state is called "connected" it is likely that you want to call a method called connect. However, when you do not know what state you want to transition to, the implication of the category C component of the question, answering question D requires a global search of the class methods.

Resources. This subsection addresses RQ4, "What resources do programmers use while perform- ing protocol tasks?" Participants were allowed to use any resource they liked. However, participants spent 76% of their total time on documentation webpages or hovering over a method documentation. This result conforms with expectations set by the studies discussed in Section III.

We also noted patterns in the particular documentation looked at by programmers. In 56 out 74 cases (including all 6 programmers in the Result Set insertion example) the programmer looked first to the documentation related to the method call occurring at the exception location to solve their problem (next in the Result example). In 13 of the remaining 18 cases the programmer looked first at the method documentation one line above or below the exception location.The participants never looked at the documentation related to the parameter types, including the receiver type, of the method being called when the exception occurs.

Unfortunately, the exception-location method documentation was not the right place to look for the information developers

were seeking. We already discussed the problem with the Result.next documentation, but the ResultSet.get* methods were similarly unhelpful for the "Cannot: read after end" task. Equally commonly, the information needed is buried in the very last element of the documentation, the @throws annotation. This information is not displayed in Eclipse hover documentation by default. It was also often skipped by developers reading the documentation in the web page, even when they were looking for the source of an exception! These findings support tools that push rules necessary for invoking methods to developers, like eMoose directives [7].

Question characteristics. We now return to two of the characteristics discussed in Section IV-D1. Participants performed two tasks that specifically required the participants to determine the state of an unknown instance. In both cases, all participants expressed hope for or requested a state test method. More surprisingly, participants requested state test method in 5 other instances. This further reinforces the advice that state test methods should always be provided.

We mentioned that missing state transitions caused frequent questions. However, type qualifier protocols—in which objects never support certain methods after construction—were very easy for participants. Participants seemed to intuitively understand that a ResultSet is created as scrolling or forward only and cannot be changed thereafter. On the other hand, lifecycle protocols, in which the state transitions only moved in one direction frustrated the participants.

VI. Threats to Validity

We started the forum mining with a large list of classes from the Java Standard Library. These were taken primarily from the results of a single study [2]. Beckman's study used a static analysis to find candidate protocols for manual investigation. This analysis missed protocols whose violations do not result in a thrown exception, nor protocols that check for protocol violations in non-standard ways. The interested reader is referred to Section 2.4 of that paper for further details. More generally, all of the APIs in our study are both libraries and from the "resource programming" domain. The protocol barriers may be different for other types of APIs.

We also do not know exactly how representative the Stack Overflow questions are of actual problems encountered in practice, nor if they really are the most difficult problems. For example, programmers may look to other sources to solve their hardest problems. Similarly, the particular demographic that uses Stack Overflow the most may have different problems than a more representative sample.

The developers who performed the laboratory study were professional engineers, but they were all personal contacts. It is therefore possible that they are very unrepresentative of the population of all skilled developers. Furthermore, the developer sample size was very small. A larger, more representative sample of developers may have needed very different information or very different resources.

Finally, a single experimenter analyzed all of the forum questions, assigning quotes to programmer activity, and categorizing quotes. Another rater would have enabled a reliability assessment and may have caught errors. The question categories may be poorly defined and the quantitative results may be skewed by experimenter biases.

VII. Conclusion

In this study, we identified five common characteristics of the questions about API protocols that developers find particularly problematic. Using the tasks that brought about the problematic questions, we found that experienced developers spent the majority of their time (71%) addressing four types of state searches, some of which are poorly supported by current approaches to documentation.

Our observations suggest that protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently answer the four state search questions. Unfortunately, many of the tools in this area do not directly address any of these questions.

That said, when a protocol is violated some of these tools provide an error message that tells the developer what part of the protocol has been violated. In particular, the messages usually say what abstract state the object is in, thereby answering question A. Unfortunately, we are unaware of any tool that gives the developer this information when there is not an error. This is probably achievable fairly simply for tools that rely on type systems or static analysis, but is much more difficult for dynamic checkers.

The research community has provided substantially less support in answering the other three state search questions (B, C, and D). However, some programming languages support separating members by abstract state which will likely make it easier for developers to answer B and C. Similarly, a first class state change operation in a programming language makes it easier to answer D.

Throughout this paper we discussed many examples in which the information needs of developers do not match the documentation at the location it is needed. In most of the instances the relevant instructions are simply misplaced. We urge writers of documentation to carefully consider how documentation is used when considering its structure. In addition, we believe there is a research opportunity to generate protocol-specific documentation in all of the locations it is needed from simple specifications.

Finally, we mentioned briefly in Section IV-C that answerers sometimes suggested alternative libraries to questioners. These answers were often accepted and/or received many "up-votes" from the Stack Overflow community. This suggests that developers who struggle with protocol violations abandon the APIs. Researchers and practitioners are very interested in what causes tools to be adopted by developers. This study provides evidence that potential adopters can be driven away by difficulty using an API correctly.

VIII. Acknowlegements

This work was supported by NSA lablet contract #H98230-14-C-014, and NSF grant #CCF-1116907.

References

[1] Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP 2011 – Object-Oriented Programming*, pages 358–382. Springer Berlin Heidelberg, 2011.

[2] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011 – Object-Oriented Programming*, pages 2–26. Springer Berlin Heidelberg, 2011.

[3] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 217–226, New York, NY, USA, 2005. ACM.

[4] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, second edition, 2008.

[6] John M Daughtry, Umer Farooq, Jeffrey Stylos, and Brad A Myers. API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 2771–2774. ACM, 2009.

[7] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490, London, UK, 2004. Springer-Verlag.

[9] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[13] Benjamin V. Hanrahan, Gregorio Convertino, and Les Nelson. Modeling problem difficulty and expertise in stackoverflow. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 91–94, New York, NY, USA, 2012. ACM.

[14] Ciera Jaspan and Jonatathan Aldrich. Are object protocols burdensome? an empirical study of developer forums. In *Evaluation and Usability of Programming Languages and Tools Workshop (PLATEAU '11)*, 2011.

[15] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP 2009 – Object-Oriented Programming*, pages 27–51. Springer Berlin Heidelberg, 2009.

[16] Ciera N.C. Jaspan. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116.

[17] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[18] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[19] George Kuk. Strategic interaction and knowledge sharing in the kde developer mailing list. *Management Science*, 52(7):1031–1042, 2006.

[20] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[21] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[22] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[23] Chris Parnin and Christoph Treude. Measuring api documentation on the web. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pages 25–30. ACM, 2011.

[24] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011.

[25] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[26] J. Sillito, G.C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on*, 34(4):434–451, 2008.

[27] Anselm L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, June 1987.

[28] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

[29] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.

[31] Joshua Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, December 2013. CMU-ISR-13-117.

[32] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *European Conference on Object Oriented Programming (ECOOP)*, 2014.

[33] Christoph Treude, Ohad Barzilay, and M-A Storey. How do programmers ask and answer questions on the web?: NIER track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 804–807. IEEE, 2011.

[34] Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. Gender, representation and online participation: A quantitative study of stackoverflow. In *International Conference on Social Informatics. ASE*, 2012.

[35] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.

[36] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 38–42. IEEE, 2012.