a method documentation. This result conforms with expectations set by the studies discussed in Section 3.1.

I 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. I 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 [Dekel and Herbsleb, 2009].

Question characteristics. We now return to two of the characteristics discussed in Section 3.2.4. 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.

I 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 (e.g. cannot disconnect a URLConnection) frustrated the participants.

3.4 Threats to Validity

I started the forum mining with a large list of classes from the Java Standard Library taken. These were taken primarily from the results of a single study (Beckman et al. [2011]). 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. Since we seeded our mining with those very same APIs, our study is missing the same classes of protocols. 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.

I 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.

I conducted the studies in this chapter after already designing and building Plaid. It is possible that the barriers uncovered here are biased towards barriers that are addressed by the Plaid design. In particular, state search is something that Plaid supports well. Therefore, I may have inadvertently biased the results toward state search.

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, I was the only person analyzing forum questions, assigning quotes to programmer activity, and categorizing quotes. Another rater may have caught errors and enabled me to asses the reliability of the categorization. The question categories may be poorly defined and the quantitative results may be be skewed by my biases.

3.5 Discussion

In this study, I identified five common characteristics of the questions about API protocols that developers find particularly problematic. Using the tasks that brought

about the problematic questions, I 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, I am 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 chapter I 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. I urge writers of documentation to carefully consider how documentation is used when considering its structure. In addition, I believe there is a research opportunity to generate protocol-specific documentation in all of the locations it is needed from simple specifications.

Finally, I mentioned briefly in Section 3.2.3 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.

Chapter 4

Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming

4.1 Introduction

Many researchers have developed protocol checkers which are designed to make it easier for programmers to correctly use APIs with protocols (e.g. [Bierhoff et al., 2009; Dwyer et al., 2007; Foster et al., 2002]). These tools require programmers to specify protocols using alias and typestate annotations that are separate from code. To automate the annotation process, several tools mine protocol specifications using dynamic analysis [de Caso et al., 2011] or static analysis [Beckman and Nori, 2011; Whaley et al., 2002]. A recent survey of automated API property inference techniques described 35 inference techniques for ordering specifications [Robillard et al., 2013].

However, the qualitative studies described in the previous chapter found that programmers using API protocols spend their time primarily on four types of searches of the protocol state space. Protocol checker output is unlikely to help programmers perform many of these searches.

Instead, in this chapter I introduce a novel documentation generator called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. Plaiddoc is extracted automatically from the standard Javadoc annotations plus

new Plaiddoc specifications. Plaiddoc is named for the Plaid programming language, introduced in Chapter 2, which embeds similar state-oriented features, and from which Plaiddoc could, in principle, be automatically generated. I evaluate Plaiddoc against a Javadoc control in a 20-participant between-subjects laboratory experiment.

The experiment attempts to answer the following five research questions:

RQ5 Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc?

RQ6 Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?

RQ7 Will programmers who use Plaiddoc answer state search questions more correctly than programmers who use Javadoc?

RQ8 Will programmers get better at answering state search questions as they get more practice?

RQ9 Are programmers who use Plaiddoc better than programmers who use Javadoc at mapping general state concepts to API details?

All of the tasks performed by participants asked participants to answer a question. I therefore use the words task and question interchangeably in the rest of this chapter. Most of these questions were instances of the four state search categories introduced in Section 3.3.2. Some of the questions were not state related and were chosen to benefit Javadoc. These were included to measure the extent to which the potential advantages Plaiddoc has on state tasks are counterbalanced by disadvantages on non state tasks. Task ordering was alternated to measure learning effects, and a post-study quiz was administered to gauge concept understanding.

Participants using Plaiddoc completed state tasks in 46% of the time it took Javadoc participants ($p<0.001$). but were approximately equally fast on non-state tasks ($p=0.8$). Plaiddoc participants were also 7.6x less likely to answer questions incorrectly than Javadoc participants ($p=0.002$). Finally, Plaiddoc and Javadoc participants were approximately equally able to map state concepts to API details. Nevertheless, our overall results suggest that Plaiddoc can provide a lightweight mechanism for improving programmer performance on state-related tasks without negatively impacting traditional tasks.

More broadly, the results of this study also provide indirect support for several programming language design choices. This study provides quantitative evidence for the productivity benefits of type annotations as documentation and state-oriented

language features, topics I discuss in detail in Section 4.8.

4.2 Background and Related Work

In their seminal paper entitled "Why a diagram is (sometimes) worth ten thousand words," Larkin and Simon [1987] introduce a computational model of human cognition to compare informationally equivalent diagrams and text. They demonstrate in this model that solving math and physics problems with text-based information can require many more steps than solving the same problems with diagrams. The most important difference between the diagram steps and text steps is that much more effort in text is spent searching for needed details. One particularly noteworthy reason for the search difference is that diagrams often collocate details that are needed together.

Larkin and Simon's theory has been effectively applied to many other (non-diagramatic) information contexts. For example, Chandler and Sweller [1991] show in a series of experiments that integrated instructional material and the removal of non-essential material can facilitate learning in a variety of educational settings. There are many more closely related examples: Green and Petre [1996] develop cognitive dimensions to evaluate visual programming languages, the GOMS [John and Kieras, 1996] model has proven effective at predicting user response to graphical user interfaces (GUIs), and MCRpd [Ullmer and Ishii, 2000] models physical representations of digital objects.

The results of two studies of API design choices are best understood through Larkin and Simon's search lens. It is easier for programmers to use constructors to create instances than factory methods, because constructors are the default and are therefore the start of any search [Ellis et al., 2007]. Methods that are located in the class a programmer starts with are easier to find than methods in related classes [Stylos and Myers, 2008]. The impact of small design changes shown in these papers emphasizes the importance of information seeking on API usability, and suggests that a similar impact may be possible with other small interventions.

All of this research suggests that there is an opportunity to modify an API artifact to create an informationally equivalent alternative that will improve programmer performance with protocol search. Which artifact? Which changes will be most effective? To answer these questions it is useful to look at the interventions that have proven effective with other complex APIs.

One effective way to learn to use an API is to find a related example. Rosson and

Carroll [1996] studied programmers using reusable Smalltalk GUI components and found that participants "relied heavily on code in example applications that provided an implicit specification for reuse of the target class." The significance of examples encouraged researchers to develop example repositories to enable programmers to find examples easily [Neal, 1989; Ye et al., 2000]. Unfortunately, the effectiveness of these repositories was limited by the retrieval mechanism which required too much (and too complex) input from programmers.

More recently, MAPO [Zhong et al., 2009] and Strathcona [Holmes et al., 2005] automatically retrieve examples from the structure of the program the programmer is writing. Zhong et al. [2009] performed a controlled experiment in which participants using MAPO produced code with fewer bugs than participants in other conditions. This result is notable because it shows that API interventions can produce higher quality responses, not just more rapid responses.

The eMoose IDE plugin has proven similarly useful to developers using complex API specifications [Dekel and Herbsleb, 2009].The eMoose tool pushes directives—rules required to use a method correctly—to the method invocation site. The concrete rules that make up a protocol (e.g. one cannot call setDoInput on a connected URL-Connection) are examples of directives. Dekel's evaluation of eMoose demonstrated significant programmer performance improvements during library usage tasks (including one library with a protocol).

Unfortunately, examples and directives are labor intensive for API designers to produce. In large complex APIs it is often impossible to generate examples for every possible use case. Even after they are produced, it is hard to keep them in sync with the API as it changes, because there is no mechanism to enforce conformance. Examples can also serve as a crutch toward learning, and the most effective students learn to generate their own examples [Chi et al., 1989].

The design of Plaiddoc is inspired by all of the research discussed in this section. I modify Javadoc to produce an informationally equivalent documentation format aimed at facilitating speedier state search. Plaiddoc is generated from specifications whose conformance with code can be checked automatically. Plaiddoc specifications, like eMoose directives, are co-located with each method. The specifications themselves contain just the right state details so programmers can generate their own examples of correct API usage. The details of the Plaiddoc design are discussed in the next section.

4.3 Plaiddoc

To follow the rest of this chapter, it is important to understand the design of Plaiddoc. To do so, it is necessary to first explain Javadoc. Javadoc is a tool for generating HTML documentation for Java programs. The documentation is generated from Java source code annotated with "doc comments" which contain both prose description and descriptive tags which tie the prose to specific program features. For example, a doc comment on a method will describe the method in general and then provide tags and associated comments for the parameters, the return value, and/or any exception the method throws.

The webpage generated by Javadoc for a class has six parts. The top and bottom contain navigation elements which allow the reader to quickly browse to related documentation. After the navigation elements, the class description appears at the top of the page. It states the name of the class and links to superclasses and known subclasses. It then follows with an often long description which can include: the purpose of the class, how it is used, examples of use, class-level invariants, relationships to other classes, etc.

After the class description, the page includes four related elements: the field summary, method summary, field details, and method details. The field summary is a table containing the modifier, type, name, and short description of each public field sorted in alphabetical order. The method summary is extremely similar: it shows the modifier, return type, method name, type and name of all parameters, and short method description in alphabetical order. The field and method details show each field (or method) in the order they appear in the source file with the full description including historical information and any tags.

The Plaiddoc generated webpage maintains all of the look and feel of the Javadoc page. The fonts, colors, and visual layout are identical. However, the method summary section is restructured and extra information is added to the method details section. A partial screenshot of the ResultSet page is shown in Figure 4.1. The full page is available on the web.[1] The screenshot shows the method summary for the top-level Result state and the Open state.

As in Plaid, methods in the summary are organized by abstract state. In Javadoc, there is one table containing all of the methods of a class, while in Plaiddoc there is one table per abstract state. For example, the Disconnected state of URLConnection has a table containing all of the methods available in it, including setDoInput and connect.

[1] http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/PlaiddocResultSet.html

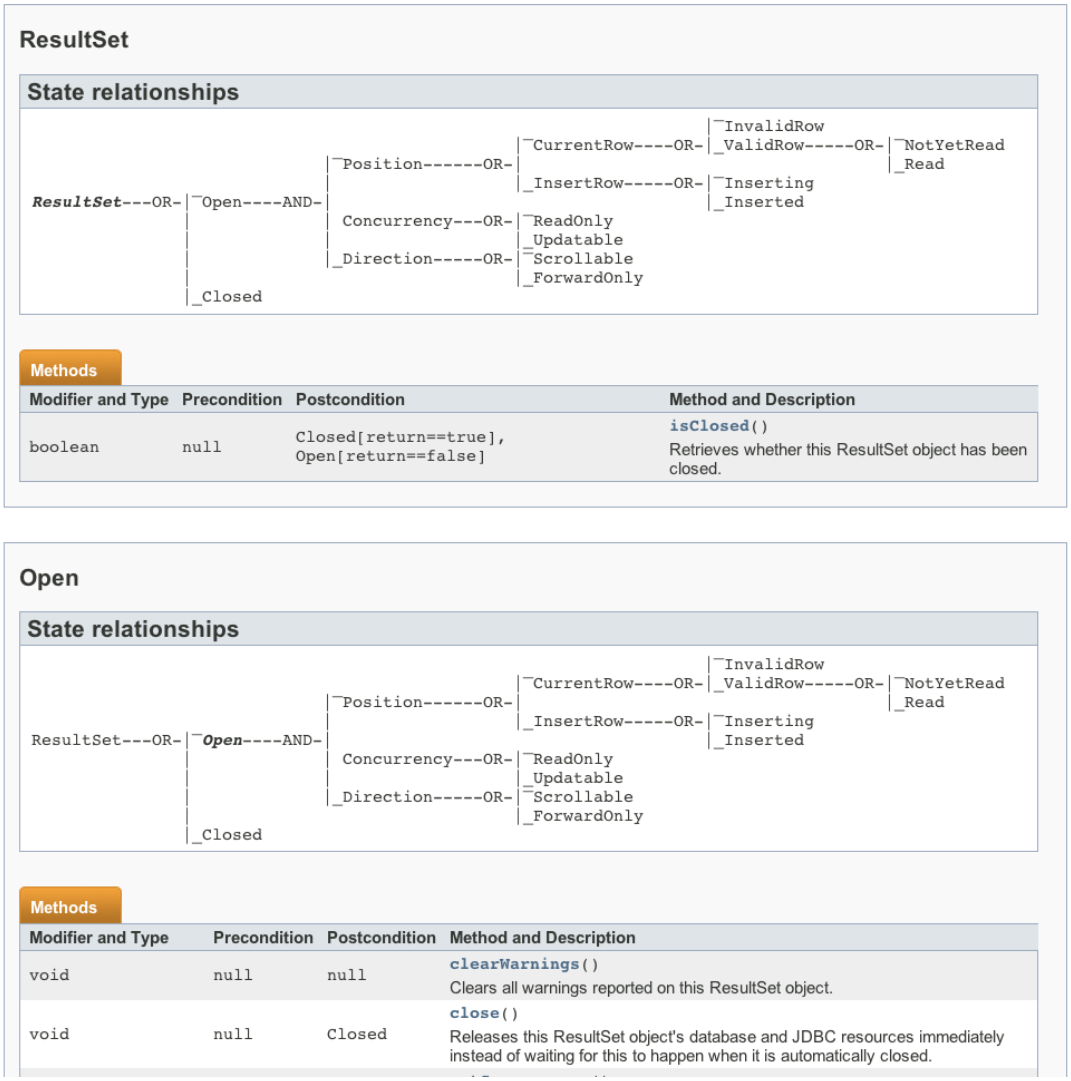

Figure 4.1: Screen shot of part of the Plaiddoc ResultSet page.

One important rule I followed when designing Plaiddoc is that there is exactly one Plaiddoc page per Javadoc page. This rule ensures that the any observed differences between participants using Plaiddoc and Javadoc is a consequence of Plaiddoc's extra features and not the result of differences in page switching. There are two consequences of this rule: 1) All of the possible states of single Java class appear in the same Plaiddoc page.[2] 2) Multi-object protocols appear in multiple Plaiddoc pages. Six of the tasks in this study involve the Timer and TimerTask classes which impose a multi-object protocol. In these tasks, Javadoc participants were given two pages and Plaiddoc participants were given two pages.

[2]e.g. The "Open" and "Closed" states of ResultSet appear on a single page.

An automatically generated diagram which shows all of the states of the class and where the particular state fits in, appears above each state table. The current state is bolded and italicized, while other states are displayed in the standard font. This diagram is *primitive*; it does not contain extensive capabilities like hyperlinks from state names to state tables, collapsing/expanding children, transition arrows, or even a nice graphical look. The diagram is primitive for three reasons: 1) Plaiddoc was designed for this experiment, and was therefore not polished for use outside the laboratory. 2) More capabilities gives participants more potential paths to solve tasks and thus introduces variation into the study. 3) If one adds features it is harder to understand which particular features are important or unimportant. Plaiddoc was designed with the minimum set of features I believed would be an effective group.

The Plaiddoc page also contains two new columns in the method details table. These columns are state preconditions and postconditions. The only valid predicates are state names, state names with a parameter, or combination of the two separated by the AND or OR logical operators. For example, "Disconnected," "Scheduled task," and "Updatable AND Scrollable" are valid preconditions or postconditions but "value > 0" is not. The same information is added to the method summary. The state to which a method belongs is an implicit precondition for that method. For example, the close method shown in Figure 4.1 lists no preconditions, but since it belongs to the Open state, the ResultSet must be in the Open state to call the close method.

To generate a Plaiddoc class page, the Plaiddoc tool requires three inputs: the Javadoc page of the class, a JSON file specifying the state relationships of the class, and a JSON file containing preconditions and postconditions for each method and mapping methods to states.

The JSON files are very simple. The state file must contain a single object whose fields are states, each of which must contain either an "or-children" or "and-children" field. These "children" fields are arrays containing state names, which in turn must be defined in the same file. The methods file must contain an array of method objects which contain four fields: "name" (including parameter types to distinguish statically overloaded methods), "state" (which must map to a state defined in the state file), "pre" for preconditions, and "post" for postconditions.

It is important to map the features of Plaiddoc just described to concepts, in order to understand the implications of the experiment described here on other research (e.g. the Plaid language itself). Plaiddoc organizes methods by state instead of by class, by separating the method summary table by state. Plaiddoc makes state transitions

explicit when state postconditions differ from preconditions. The Plaiddoc preconditions and postconditions make use of state-based type specifications. Finally, rich state relationships are displayed to programmers at the top of each method table. See e.g. the “State relationships" box in Figure 4.1.

4.4 Methodology

The experimental evaluation of Plaiddoc uses a standard two by two between-subjects design, with five participants in each of the four conditions. The experiment compares Plaiddoc to a Javadoc control and presents two task orderings to measure learning effects. The recruitment, training, experimental design, tasks, and post-experiment interview are presented in the following sections. All of the study materials can be found in Appendix C.

4.4.1 Recruitment

All 20 participants were recruited on the Carnegie Mellon campus. Half of the participants responded to posters displayed in the engineering and computer science buildings. The other half were solicited in-person in a hallway outside classrooms which typically contain technical classes. Participants were screened for Java or C# knowledge and experience with standard API documentation. Participants were paid $10 for 30-60 minutes of their time. The 20 participants that made it past the screening all completed the study.

Twelve of the participants were undergraduate students, all of whom were majoring in computer science, electrical and computer engineering, or information systems. The other eight were masters students in information systems or computer engineering programs. Eleven students had no professional programming experience outside summer internships, five students had one year of full-time professional experience, and four had more than one year of experience.

4.4.2 Training

After signing consent forms, participants were given approximately 10 minutes of training. Every participant, regardless of experimental condition, received exactly the same training. The training was read from a script to help ensure uniformity.

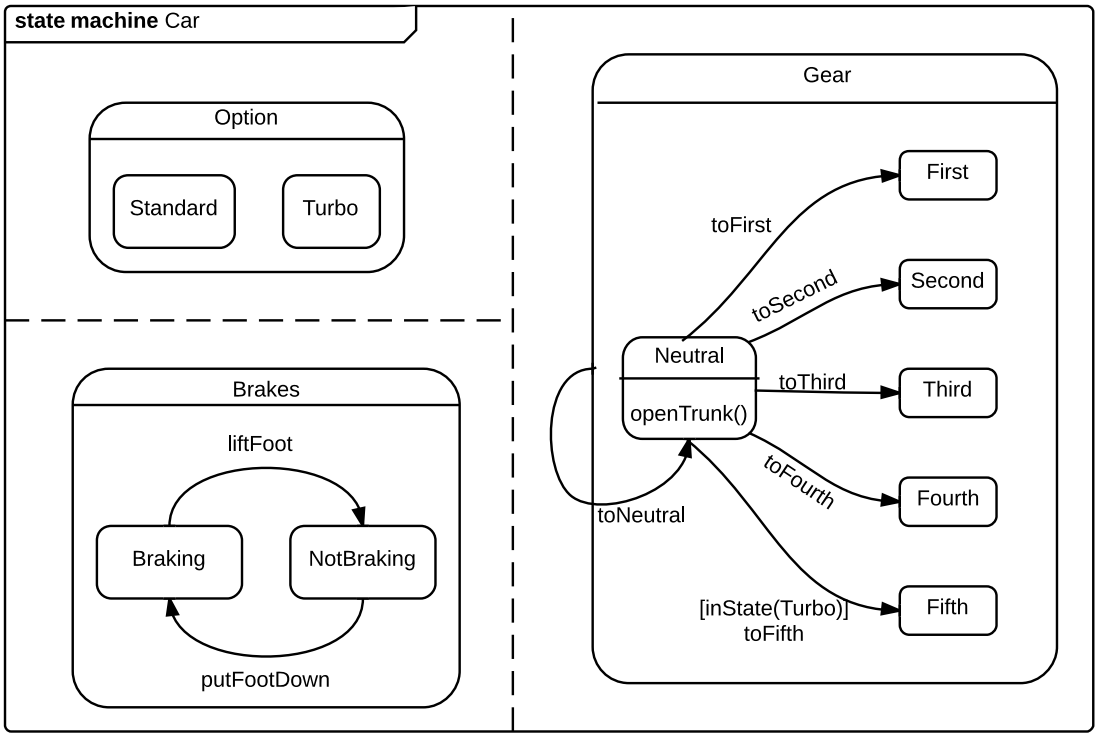

Figure 4.2: Car state machine used for participant training.

All participants were familiar with Javadoc, but the training included an explanation of both Javadoc and Plaiddoc to ensure baseline knowledge in both formats. The goal of this study is to compare the impact of the documentation formats on state search tasks, not the impact of training. Therefore, we kept training consistent to avoid a confounding factor. All of the state concepts are first taught via UML state machines, then Javadoc, then Plaiddoc.

The training materials introduce participants to the basic concepts of object protocols and to the documentation formats used in the study. The training makes concepts concrete using a Car API I constructed for the purpose. Regarding protocols, participants learn:

- that methods are available in some states and not others
- that some methods transition objects between states
- that states can be hierarchical
- that child states can be either or-children or and-children

These concepts were reinforced by asking participants simple, scripted questions about the Car API. The questions were designed to be answerable very quickly by participants. I created a UML state machine (shown in Figure 4.2), Javadoc documentation,

and Plaiddoc documentation for the Car API and these were printed and handed to participants.

The top-level state for Car objects (named "Car") has three and-children, each of which has two or more or-children: *gear* to represent the car's manual transmission, *brakes* to represent whether the car is braking or not, and *option* to represent whether the car has the "turbo" option or not. I used these states to introduce state hierarchy, or-states, and and-states. I introduced transitions via brakes. One can transition to the "Braking" state from the "NotBraking" state by calling the "putFootDown" method. The openTrunk method, which does not change the gear state, introduces state-dependent methods. In the example, like in many real-world cars, one can only open the trunk when the car is in the neutral gear.

Like all and-children, the car's three substates are independent, in the sense that changing the gear state has no effect on the braking or option states. However, one unique wrinkle in the example is that the turbo state enables a fifth gear substate of gear that is not available otherwise. The toFifth method has two preconditions — the car must be in the neutral gear and it must have the turbo option. In the study tasks discussed later, some of the ResultSet methods also have multiple preconditions.

4.4.3 Experimental Setup

Participants were asked 21 questions about three Java APIs: 1) Six questions about java.util.Timer and java.util.TimerTask. I refer to these questions as the Timer questions throughout the rest of this paper. 2) Ten questions about java.sql.ResultSet. 3) Five questions about java.net.URLConnection. The experimenter read each question aloud and handed the participant a piece of paper with the same question written on it.

Participants were seated in front of a computer, and asked to answer the question by looking at documentation on the computer screen. The experimenter opened the documentation for the participant in a browser window. Both the Javadoc and Plaiddoc documentation were opened from the local file system to present a consistent URL and to prevent network related problems. The computer screen and audio (speech) were recorded with Camtasia.

Half of the participants were shown standard Javadoc documentation for all questions and half Plaiddoc documentation. Participants were allowed to make use of the browser's text search (i.e. Control-F). However, they were not allowed to use internet resources (e.g. Google, StackOverflow).

I chose a between-subjects design to control for cross-task contamination. Many

software engineering studies use within-subjects designs to reduce the noise from individual variability. I guessed based on pilot data that individual variability in our study would be relatively low and I therefore opted for the cleaner between-subjects design. As we will see in Section 4.5, the study was sufficiently sensitive to distinguish between conditions so my guess turned out to be accurate.

Questions were asked in batches — all of the questions related to a particular API were asked without interruption from questions about another API. Within each batch, each question was asked in the same order to every participant. However, half of the participants were asked the Timer batch first and half were asked the UrlConnection batch first. The ResultSet batch always appeared second and the remaining batch appeared third. I wanted the Timer and URLConnection batches to each appear last so I could measure the learning effects on those batches. All other ordering was uniform across conditions to avoid unnecessary confounding factors.

The study had a total of four between-subjects conditions: Plaiddoc with Timer first (condition #1), Plaiddoc with URLConnection first (condition #2), Javadoc with Timer first (condition #3), and Javadoc with URLConnection first (condition #4). Participants were assigned to conditions based on the order they appeared in the study. The nth participant was assigned to condition #n modulo 4. No effort was made to balance participants by experience or any other demographic factor. Therefore, there were exactly five participants in each condition.

4.4.4 Tasks

The 21 questions asked of the participants are shown in Table 4.1. Sixteen of the questions were instances of the four categories of state search enumerated in Section 3.3.2. Since these questions are state specific, I refer to them as the state questions. The remaining five questions were non-state questions, which were designed to be just as easy or easier with Javadoc than Plaiddoc. These questions were not about states or protocols, and I therefore refer to them as the non-state questions.

I selected the state questions with a three-phase process. First, I generated all of the instances of the general categories I could think of for each API. Second, since I did not want the answer or the process of answering one question to affect others, I removed questions which were not independent. Some additional non-independent questions were removed during piloting. Third, I pruned the ResultSet questions to include two instances of each question category by random selection. The study was too long with the full set of ResultSet questions.

Table 4.1: Category, identifier and question text for all of the questions asked of participants in the main part of the study. Questions with identifiers beginning with T involved java.util.Timer and java.util.TimerTask, R involved java.sql.ResultSet, and U involved java.net.URLConnection.

Cat.	ID	Question text
T	T-1	How do I transition a Timer Task from the Virgin state to the Scheduled state?
N	T-2	What is the effect of calling the purge method on the behavior of the Timer?
C	T-3	What methods can I call on a Scheduled TimerTask?
N	T-4	What is the difference between schedule(TimerTask task, long delay, long period) and scheduleAtFixedRate(TimerTask task, long delay, long period)?
O	T-5	What state does a TimerTask need to be in to call scheduledExecutionTime?
C	T-6	Can I schedule a TimerTask that has already been scheduled?
N	R-1	How is a ResultSet instance created?
C	R-2	Can I call the getArray method when the cursor is on the insert row?
O	R-3	What state does the ResultSet need to be in to call the wasNull method?
T	R-4	How do I transition a ResultSet object from the ForwardOnly to the Scrollable State?
O	R-5	Which states does the ResultSet need to be in to call the updateInt method?
A	R-6	What state is the ResultSet object if a call to the next method returns false?
T	R-7	How do I transition a ResultSet object from the CurrentRow to the InsertRow state?
N	R-8	Why does getMetadata take no arguments and getArray take a int columnIndex or String columnLabel as an argument?
C	R-9	Can I call the isLast method on a forward only ResultSet?
A	R-10	What states could the ResultSet object in when a call to the next method throws a java.sql.SQLException because it is in the ResultSet is in the wrong state?
A	U-1	What state is the URLConnection in after successfully calling the getContent method?
C	U-2	If the URLConnection is in the connected state can I call the setDoInput method?
N	U-3	How do I create a URLConnection instance?
O	U-4	What state does the URLConnection need to be in to call the getInputStream method?
T	U-5	What method transitions the URLConnection from the Connected to the Disconnected state?

Category definitions

A Instance of the "What abstract state is an object in?" question category.
C Instance of the "What are the capabilities of an object in state X?" question category.
N Instance of the non-state question category.
T Instance of the "How do I transition from state X to state Y?" question category.
O Instance of the "In what state(s) can I do operation Z?" question category.

The final question set includes three instances of A) “What abstract state is an object in?”, five instances of B) “What are the capabilities of an object in state X?”, four instances of C)“In what state(s) can I do operation Z?’,’ and four instances of D) “How do I transition from state X to state Y?” Participants in all conditions were given a glossary listing all of the states of the API in question with a short description of each. Participants were instructed to answer questions in categories A and C with the name of a state from the glossary. In other words, these questions were multiple choice.

The names of states in the glossary matched those in Plaiddoc. The names themselves were taken from the Javadoc as much as possible. I did not want to disadvantage Javadoc unnecessarily, so I tried to make it as easy as possible for participants to perform the mapping from the prose description in the Javadoc to the state names in the glossary. In two cases there was no obvious name to give the state from the Javadoc. First, I called a URLConnection that has not yet connected “Disconnected,” which is a word that appears neither in the Javadoc nor the Java source code. Second, I called a TimerTask that is unscheduled, “Virgin” even though this word never appears in the Javadoc. In this case we borrowed the word from the implementation code—the state of a TimerTask is encoded with an integer, and the integer constant used for an unscheduled TimerTask is called VIRGIN. Finally, I wrote all of the descriptions to succinctly explain the meaning of the state name.

All of the non-state questions require understanding a non-state detail of the API or comparing two details. Since the Plaiddoc API documentation is larger than the Javadoc documentation one might expect that it would be slightly easier to answer these questions with Javadoc. Two of the non-state question are instances of “how do I create an instance of class X?”, two ask participants to compare two methods (in one case the methods were in different states), and one asks participants to understand non-state details of the behavior of an individual method.

Participants were instructed to “find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it.” Whenever a participant answered a question for the first time, the experimenter asked,“is that your final answer?” Participants were limited to ten minutes per task. The experiment proceeded to the next task whenever a participant answered a question and confirmed it or the time limit was reached. Participants were not told whether their answer was correct and the experiment proceeded regardless of answer correctness.

4.4.5 Post-experiment interview

After completing the experiment participants were asked four questions to see how well they could map the state concepts I trained them about before the study (e.g. and-states, or-states, state hierarchy, impact of transitions on and-states) to the particular APIs they saw in the study. For example, I asked "What is an example of two ResultSet and-states?" Participants were also asked to rate their affinity to the documentation they used, and if they used Plaiddoc to compare Plaiddoc to Javadoc on a five point Likert scale. Then they were asked “Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?” Finally, some individuals were also asked additional questions about their task performance at the experimenter’s discretion.

4.5 Results

In this section, I discuss the study results and try to give the best evidence to answer the research questions presented in the introduction. I first compare the task completion performance of Plaiddoc and Javadoc participants. Then I compare the correctness of these responses provided by those same groups. I follow with an evaluation of the learning effects of performing study tasks. Finally I discuss the post-study interview and pilot results.

4.5.1 Task Completion Time

In this subsection I discuss the results related to the task completion time output variable. This output variable addresses RQ5 and RQ6 (Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc? and Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?) by comparing task completion times across conditions.

To determine completion time I analyzed the video and marked when I finished reading the task question and when the participant confirmed his or her “final answer.” The difference between these two marks was noted in the task completion time.

The ten-minute task time limit was reach by many participants on question R-4, but never on any other question. In fact, only two participants exceeded five minutes while answering any other question, and they did so for only one question each. Timeouts are not directly comparable to other timing data, and therefore we evaluate question R-4

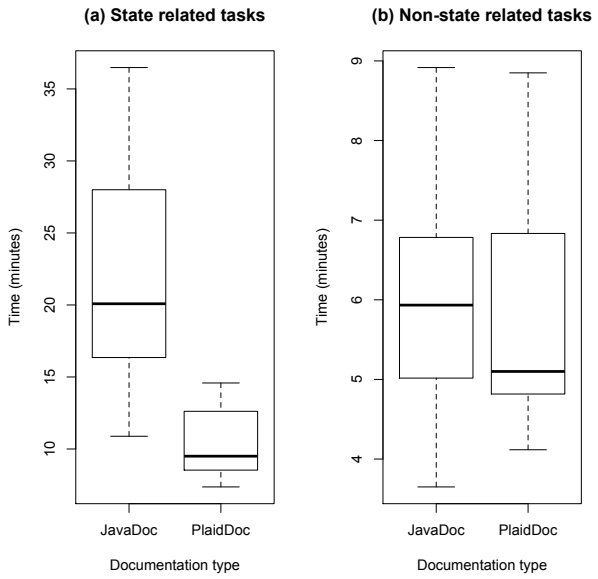

Figure 4.3: Box plot comparing total completion time for each of the Plaiddoc participants to that of Javadoc participants. The median is displayed with a horizontal line, the 25th percentile measure with the bottom of the box, the 75th percentile measures with the top of the box, the minima value with bottom whisker, and the maximum value with the top whisker.

separately, and in detail, in Section 4.5.2. This subsection does not include data from question R-4.

The total completion time for each of the Plaiddoc and Javadoc participants on state questions is visualized by the box plot in Figure 4.3(a), and for non-state question in Figure 4.3(b). A two-factor fixed-effects ANOVA revealed no significant interaction between documentation type and task ordering (p=0.25) on total task completion time. Therefore, I compare all 10 Plaiddoc participants against their 10 Javadoc counterparts.

The mean total completion time of all state search tasks was 10.3 minutes in the Plaiddoc condition, and 22.4 minutes in the Javadoc condition (2.17x difference). An independent samples two-tailed t-test revealed that the difference is statistically significant ($p < 0.001$). The difference between the means was 12.1 minutes, and 95-percent confidence interval was 6.38 to 17.8 minutes.

The mean completion time of non-state tasks was 5.77 minutes in the Plaiddoc condition, and 5.95 minutes in the Javadoc condition. Unsurprisingly, this difference is not statistically significant (p=0.802). The 95-percent confidence interval of the difference is -1.32 to 1.68 minutes.

The state search categories I introduced in Section 3.3.2 can be subdivided into two categories. In two of the search categories, a participant begins his or her search at a

state and tries to find a method.[3] In the other two search categories the participant starts at a method or other detail (e.g. exception, instance creation), and tries to find a state.[4] Since methods are organized in Plaiddoc by state one would expect that Plaiddoc would improve performance primarily for searches that proceed from a state to a method. This hypothesis turns out to be correct — Plaiddoc outperformed Javadoc in these categories by 2.41x. However, one might expect that Plaiddoc would not be helpful in the method first categories, but Plaiddoc outperformed Javadoc by 1.87x in these categories. Therefore, Plaiddoc appears to be more helpful for state-first search than method-first search. I performed two factor, fixed-effects ANOVA in which the two factors are documentation type and search type and the output variable is time. The interaction term between documentation type and search type is only significant at the ten-percent level (p=0.0891).

Demographics

I did not balance participants in conditions by any demographic factor. By random chance, six of nine students with experience and three of four with more than one year of experience were assigned to the Javadoc conditions. However, experience had no significant impact on the timing results. A two-factor ANOVA where the two factors were experience and documentation type showed no significant effects from experience ($p = 0.813$) or the experience by documentation type interaction term ($p = 0.719$).

Feature comparison discussion

Every participant used text-search (i.e. CTRL-F in the browser window) to find method names. They then used the location in a state box, pre-conditions, post-conditions, and state relationship diagrams to answer the question efficiently. Plaiddoc is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. The difference in relative performance between the state categories allows us to (very roughly) compare the benefits of state organization to the other three features. Since the method based search does not benefit from the state-based organization, all of the performance differences observed in the method based search tasks are likely to derive from explicit state transitions, state-based type specifications, and rich state relationships. The extra performance of the state based search is likely to derive from

[3] What are the capabilities of an object in state X? How do I transition from state X to state Y?
[4] What abstract state is an object in? In what state(s) can I do operation Z?

Table 4.2: Documentation type, number of correct answers, number of incorrect answers, and number of question time-outs of each participant on the 16 state search questions.

	Paricipant #																				Total	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Pdoc	Jdoc
DocType	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	J
Correct	15	15	14	16	15	16	15	14	15	15	14	14	15	15	16	16	15	15	11	13	151	143
Incorrect	1	0	2	0	0	0	1	1	0	0	2	2	1	0	0	0	0	0	5	2	2	15
Timed-out	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0	1	7	2

the state-based organization. I do not think it's possible to separate the benefits of the embedded state diagram from the preconditions and postconditions. In one early pilot I did not include the state diagram and the participant struggled to answer questions that required knowledge of state relationships. Similarly, a state diagram without detailed information about the requirements and impact of method calls would likely not be effective.

4.5.2 Correctness

Almost half of the participants provided at least one wrong "final" answer to a state-search question. Among the 320 total answers provided to the 16 state search questions 294 were correct, 17 incorrect, and nine were not provided because the question timed out. In this subsection I compare the correctness of Plaiddoc answers to Javadoc answers (RQ7). The number of right, wrong, and timed-out answers for each participant are shown in Table 4.2.

Only two of the 17 wrong answers were provided by Plaiddoc participants. Plaiddoc participants answered 98.75% of the questions correctly, and Javadoc participants answered 90.5% correctly. The odds ratio in the sample is 7.92.[5] I analyzed the contingency table of Javadoc vs. Plaiddoc and Correct vs. Incorrect using a two-tailed Fisher's exact test. The contingency table is shown in Table 4.2 in the rows labeled "Correct" and "Incorrect" and the columns labeled "Pdoc" and "Jdoc". The test revealed that the difference is very significant (p=0.002). The 95-percent confidence interval of the odds ratio is 1.78 to 72.1.

[5]The odds ratio is a standard metric for quantifying association between two properties. In our example, it is the ratio of the odds of being correct when using Plaiddoc to the odds of being correct when using Javadoc.

Incorrect responses

All of the wrong answers and time-outs were provided to just five of the 16 state questions. No wrong answers were provided to any of the non-state questions. It is worth discussing the content of the wrong answers to provide insight into the types of problems programmers face when answering state-related questions.

In response to question T-3, a Plaiddoc participant (#19) incorrectly suggested that none of the TimerTask methods could be called on a scheduled TimerTask because "the methods are called by the Timer." This participant correctly noted the main mode of usage, but incorrectly assumed this was the exclusive mode of usage.

In response to question T-5, three[6] Javadoc participants incorrectly suggested that TimerTask scheduledExecutionTime can be called in any state when in fact it can only be called in the executed state. Three of these wrong participants noted correctly that scheduledExecutionTime does not specify that it throws an exception. Unfortunately, not every protocol violation results in an exception, a fact that was noted in pre-test training.[7] In this case, the protocol is documented in the description of the return value, which is described as "undefined if the task has yet to commence its first execution." In the post-experiment interview all three incorrect participants said that they did not notice this return value description.

In response to T-6, two Javadoc participants incorrectly replied that one can schedule an already-scheduled TimerTask. Participant #19 answered very quickly (15 seconds) without thoroughly examining the documentation. Participant #8 read aloud from the documentation, noting that the method throws an IllegalStateException "if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated." However, #8 somehow skipped "scheduled or" while reading.

Three Javadoc participants and one Plaiddoc participant incorrectly answered U-5. The question asks, "What method transitions the URLConnection from the Connected to the Disconnected state?" There is no such method, as 16 participants correctly noted. The three incorrect Javadoc participants suggested one could transition the URLConnection to the Disconnected state by calling its setConnectionTimeout method with 0 as the timeout value argument. This method "sets a timeout value, to be used when opening a communications link to the resource referenced by this URLConnection. If the timeout expires before the connection can be established, a java.net.SocketTimeOutException

[6]Participant #19 also answered T-5 incorrectly because, as in question T-3, #19 thought all TimerTask "methods are called by the Timer" including scheduledAtFixedRate.

[7]The openTrunk method's protocol is documented by its description of the return value Javadoc training materials.

is raised." Therefore, setConnectionTimeout has no impact at all on a URLConnection instance that has already connected. Participant #1, a Plaiddoc participant, incorrectly answered that the non-existent "disconnect" method could be used to transition the URLConnection. This was the last question that participant #1 answered, so perhaps #1 was ready to leave and so didn't investigate this question thoroughly.

Finally, R-4 produced the most varied responses. The question asks the participant to transition a ResultSet object from the ForwardOnly to the Scrollable state. However, no transition is possible since ForwardOnly and Scrollable are *type qualifiers* and therefore are permanent after instance creation. Seven Plaiddoc and two Javadoc participants never answered this question because they timed out. One Plaiddoc and five Javadoc participants answered the question incorrectly. Many of the timed-out Plaiddoc participants considered but then ultimately rejected the incorrect answers provided by the Javadoc respondents. This suggests that the specifications provided by Plaiddoc participants can provide confidence that an answer is *incorrect*. The Plaiddoc participants likely traded no-answers for incorrect answers.

Four Javadoc participants incorrectly answered that the setFetchDirection method will transition a ResultSet object from the ForwardOnly to the Scrollable state. Unfortunately, this method does no such thing, instead it "gives a hint as to the direction in which the rows in this ResultSet object will be processed." These four participants did skim the description, but it seems that they relied primarily on the method name to make their determination.

One Javadoc and one Plaiddoc participant noticed the following sentences in the class description: "A default ResultSet object is not updatable and has a cursor that moves forward only ... It is possible to produce ResultSet objects that are scrollable." which is immediately followed by a code example in which the createStatement method is called on TYPE_SCROLL_INSENSITIVE as an argument on a *connection* instance. Upon reading this, both participants immediately answered that the createStatement method should be called on a *ResultSet* instance. The Plaiddoc participant even suggested that the createStatement was missing from the method details list because "Plaiddoc is just a prototype."

Questions U-5 and R-4 both ask participants to find a method that does not exist. These questions, like all state-search questions in the study, are derived from the questions participants asked in the observational study discussed in Chapter 3. However, participants in empirical studies are well-known to be compliant to experimenter demands. Therefore, some may therefore consider them to be "trick" questions. If

these questions are excluded, then Plaiddoc participants answered 140 state-search questions correctly (100%) and 0 incorrectly while Javadoc participants answered 133 correctly (95%) and 7 incorrectly. A two-tailed Fisher's exact test of this contingency table is statistically significant (p=0.014). Since Plaiddoc participants in this sample answered every question correctly, the odds ratio is infinite. The 95-percent confidence interval of the odds ratio is 1.48 (the corresponding value is 1.78 when including every state-search question) to infinity (7.92 when including state-search question). Therefore, Plaiddoc participants were significantly more likely to respond correctly than Javadoc participants even when excluding "trick" questions.

Discussion

Three themes emerge from the incorrect and timed-out answers provided by participants. First, all of the time-outs occurred in question R-4 when participants were asked to find a non-existent method to transition between two states. Therefore, to answer this question correctly, participants needed to prove the absence of something to themselves.[8] Some participants felt the need to perform a brute force search of the method documentation to ensure that no methods were available that perfumed the transition. Of particular note, Plaiddoc participants didn't seem to trust that the ForwardOnly section of the Plaiddoc contained all of the potential methods.

It's also worth noting that question U-5 is in the same category but resulted in no time-outs. One possible explanation is that the ResultSet interface is much larger than the the URLConnection class[9], so it is easier to be confident that no such method exists. In addition, participants seemed to intuit that the URLConnection transition is missing, but not intuit that the ResultSet transition is missing.

Second, the questions required the participants to digest a lot of text. Participants commonly relied on heuristics and skimming to answer questions quickly. For example, the five Javadoc participants who answered R-4 with setFetchDirection matched the method name to the task and quickly confirmed the match in the description, but did not fully digest the description text. The participant who missed the word "scheduled" in the exception details was being similarly hasty. This phenomenon may partially explain why Plaiddoc participants were so much quicker than Javadoc participants, as we saw in Section 4.5.1. Plaiddoc presents a natural heuristic to participants — when

[8]Recall from Chapter 3 many forum questioners had similar problems with missing state transitions.

[9]Using the standard Google Chrome settings, printing the ResultSet Javadoc results 97 pages while printing the URLConnection Javadoc results in only 23 pages.

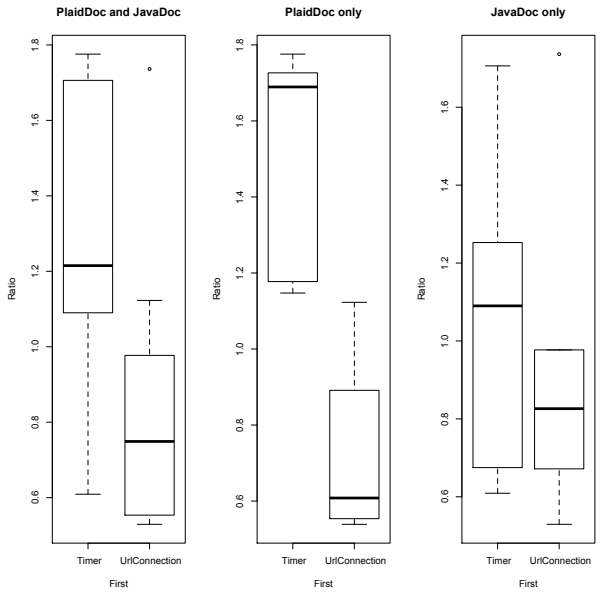

Figure 4.4: Box plots comparing ratio of Timer to UrlConnection task completion times.

examining a method, look first at the state it is defined in, then at its preconditions and postconditions.

Third, participants were tripped up by non-normal modes of use. We saw that participant #19 thought only the Timer could call TimerTask methods because that is the normal mode of use. Similarly, most protocol violations throw exceptions and are documented in the method or exception descriptions. However, scheduledExecutionTime somewhat abnormally documents the protocols in the return value description which confused three participants. Finally, abstract states normally map well to the primitive state of object instances. However, a URLConnection that has been disconnected from the remote resource is not in the Disconnected[10] abstract state, as expected by three participants.

4.5.3 Learning

To answer RQ8, which asks whether state search performance improves with practice, I alternated the order that question batches were asked of participants. As I describe in Section 4.4.3, half of the participants first received URLConnection questions and half

[10]In this particular case a better name (e.g. NotYetConnected) for the Disconnected abstract state may have avoided this confusion. However, poorly named identifiers are a fact of programming life. As much as possible the documentation (and other components of the development environment) should be robust to poor naming.

Table 4.3: Analysis of observed variance of T/U Ratio. The fixed-effects sources of variation considered are documentation type and batch order.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
DocType	1	0.06695	0.06695	0.4560	0.50914
BatchOrder	1	0.96519	0.96519	6.5737	0.02081
DocType:BatchOrder	1	0.51496	0.51496	3.5073	0.07949

first received Timer questions. The output variable I discuss in this section is the ratio of total Timer batch completion time to total URLConnection batch completion time (the "T/U ratio"). If learning occurs, then the T/U ratio should be larger for participants who performed the Timer batch first than for those who performed the URLConnection batch first. The T/U ratio is shown for each condition in Figure 4.4.

In the Javadoc condition, the mean T/U ratio of the Timer first sub-condition is 1.07 and .948 in the UrlConnection first sub-condition. This difference is not statistically significant (p=0.695). On the other hand, in the Plaiddoc condition the mean T/U ratio of the Timer first sub-condition is 1.50 and 0.743 in the UrlConnection first sub-condition. An independent samples two-tailed t-test shows that this difference is statistically significant (p=0.003).

I performed two factor, fixed-effects ANOVA in which the two factors are documentation type and batch order and the output variable is the T/U ratio. The results are show in Table 4.3. This ANOVA reveals that there is a marginally significant interaction between documentation type and batch ordering (p=0.079). This should be interpreted as weak evidence that task-completion speed improved more for Plaiddoc participants than for Javadoc participants. However, more data is needed to know for sure.

Discussion

The Plaiddoc participants performance improved significantly during the study, which is perhaps unsurprising since Plaiddoc was new to all of the participants. I would like to say with confidence that state-search performance of programmers using Plaiddoc would improve over time relative to programmers using Javadoc. However, the learning observed in the Plaiddoc condition was not significantly stronger than the learning observed in the Javadoc condition.

4.5.4 State concept mapping

To investigate RQ9, I asked four questions to map the concepts they learned about in training to the Timer, TimerTask, ResultSet, and URLConnection. Plaiddoc participants responded correctly 23 of 40 times, while Javadoc participants answered correctly 25 times. This difference is not statistically significant.

Discussion

I hypothesized that Plaiddoc participant would be better at mapping API specifics to general state concepts. I thought this because Plaiddoc makes many state concepts more salient. There is no evidence for this hypothesis in the data. Javadoc participants spent much more total time with the documentation and they read much more of the detailed prose contained inside the documentation. Perhaps this extra time and detail compensated for the state salience of Plaiddoc.

I told all of the participants that timed out while trying to find a method to transition the ResultSet from ForwardOnly to the Scrollable state, that the method did not exist. I asked if they had any ideas about how to better represent missing state transitions. Most didn't give any suggestion, but one suggested that methods that perform state transitions should be separated from other methods so they're easier to find. This suggestion is worthy of further investigation.

4.5.5 Participant preference

In the post-experiment interview I also gauged participant preferences. Nine of ten Plaiddoc participants said that a different documentation format would have been more helpful in performing the study. Seven selected UML state diagrams and two selected Javadoc. The Javadoc participants also primarily selected UML State diagrams (five of ten), followed by Javadoc (3), and Plaiddoc (2).

Discussion

The results in this study demonstrate persuasively that Plaiddoc participants outperformed Javadoc participants. Therefore participant preferences does not match the measured outcome. Why do so many Plaiddoc participants prefer another documentation format? The simplest explanation is that Plaiddoc is unfamiliar, while Javadoc is familiar. In addition, one participant in the Plaiddoc condition who preferred Javadoc

explained that he "felt lost" while using Plaiddoc. A Plaiddoc page is divided into many more subsections (one for each state) than a Javadoc page. Improved visual cues indicating the which state is being viewed might alleviate this problem. Another possible reason, is that the Plaiddoc state diagram is produced in ASCII and therefore looks old and amateurish. The state diagram does not match well with the modern look of the rest of the page. Regardless of the reason for the preference, this study's results are a cautionary tale for researchers who rely only on user preferences to evaluate tools.

4.5.6 Discussion of pilot studies

The final study that appears here was preceded by a failed alternative study. I started out trying to evaluate the effectiveness of the protocol-checking tool Plural [Bierhoff et al., 2009] in the debugging of protocol violations. Unfortunately, during pilots the variance of programmer performance was large and the effect size was both small and appeared negative. In other words, participants in our small sample performed very slightly better without Plural than with Plural.[11]

After that failed effort I decided to do the qualitative work that appears in the previous chapter to learn more and guide the design of a better, more measurable intervention. Based on the prevalence of search tasks in the results of the observational study described in Chapter 3, I decided to ask participants to perform these smaller search tasks, rather than the programming tasks that typically appear in software engineering studies. In our pilots of Plaiddoc on these search tasks the variance between participants was much lower than in our Plural studies and the difference between Plaiddoc and Javadoc was large.

Originally I had wanted to evaluate Plaiddoc against both Javadoc and UML state diagrams. Javadoc is the status quo documentation format and UML state diagrams are in many ways a best practice. Since UML state diagrams do not include non-state methods, I designed a Plaiddoc-diagram hybrid. In our pilots, UML state diagram participants were much faster at answering questions about state transitions, which is logical because these are prominent in the diagrams. However, they were much slower than even Javadoc participants at answering other questions because they were constantly switching back and forth between the documentation and the diagram. It is possible that a better designed hybrid could have alleviated this problem, but for timing's sake I dropped the UML condition from the study.

[11] See Section 5.3 for further discussion on the results of this pilot.

4.6 Threats to Validity

In this section I discuss threats to validity of my causal claims. I divide this section using the canonical categories of validity: construct validity, internal validity, and external validity.

4.6.1 Construct validity

I trained all participants equally, including training of Javadoc participants to use Plaiddoc. There is some risk in this design that Javadoc participants will be disappointed that they did not get to use Plaiddoc. They were familiar with Javadoc so they may have preferred to try something new. Therefore, Javadoc participants may have performed worse because they experienced what Shadish et al. [2002, p. 80] call "resentful demoralization." Two facts suggest that demoralization had at most a small effect on the results: First, only two of 10 Javadoc participants said they would have preferred to use Plaiddoc in the post-experiment interview. Second, both Javadoc and Plaiddoc are documentation formats and neither is particularly exciting. The classic examples in which "resentful demoralization" was measurable include much more severe differences between the control group and the experimental group. Fetterman [1982] describes an experiment evaluating a job-training program in which the control group includes participants who were denied access to the training program. Walther and Ross [1982] compare an experimental group that is paid a substantially higher participation reward to a control group paid much less. I would not expect to see anywhere near as much demoralization in our study as in these studies, even for participants who would have preferred to use Plaiddoc.

Although participants were never told explicitly, it is likely participants realized that Plaiddoc was my design. Therefore, Plaiddoc participants may have performed better and Javadoc participants worse because of "experimenter expectancies" [Rosenthal and Rosnow, 2008, p. 224]. In other words, the very fact that I expected Plaiddoc to outperform Javadoc *and* the participants could possibly infer this expectation, may have impacted in the result in the direction I expected.

4.6.2 Internal validity

The focus of this study's design is internal validity. Participants were randomly assigned, participants were isolated from outside events in equivalent settings, I used a between-

subjects design, and there was no attrition during the study. All that being said, one threat to internal validity is worth mentioning. Participants were assigned to conditions randomly, but it could be that the participants in the Plaiddoc group were better equipped to answer the questions in the study. I discussed the distribution of programming experience in Section in Section 4.5.1 and showed that it did not seem to have an effect on outcomes. However, it could be the groups differ along a different dimension—for example, programming skill, experience with protocols, intelligence—that we did not measure and this impacted the results.

4.6.3 External Validity

The qualitative studies in Chapter 3 and the experiment discussed here have opposing strengths and weaknesses. The qualitative studies emphasize external validity with realistic tasks and professional participants, but cannot be used to draw conclusions about causal relationships. The experiment in this chapter focuses on internal validity with a carefully controlled experimental design that allows strong causal conclusions. However, the external validity of the experiment is enhanced because participants performed tasks in which they were required to tackle protocol programming barriers observed in the qualitative studies. Therefore, the experimental results are likely to translate to real-world problems and the processes that programmers use to solve them. All that being said, the threats to external validity discussed in Chapter 3 extend into this study. The interested reader is referred to 3.4 for more information.

The state search tasks are connected to our qualitative results—they use the same APIs that were problematic for Stack Overflow questioners and they are instances of the state search categories that were observed repeatedly in the observational study. However, the non-state search tasks did not come from developer forums or any other real-world programming resource. Instead they were designed to simply *not* make use of Plaiddoc's novel state features. In our results, Plaiddoc participants did not perform worse on these tasks than Javadoc participants. However, it could be that there are other important categories of tasks for which Javadoc is better than Plaiddoc.

Another noteworthy external validity concern in the experiment here has to do with the student population studied. None of the participants seem to have struggled with the concept of preconditions and postconditions which are used heavily by Plaiddoc. This may be because the concept as used in the study is simple, but it may also be that the Carnegie Mellon student population I studied is especially exposed to formal methods. The very first course in the Carnegie Mellon undergraduate computer science sequence

teaches students to verify imperative programs with Hoare-style contracts.

4.7 Type annotations as documentation

Many research groups have developed specialized type-based annotation systems for particular domains. Prominent examples include information flow [Sabelfeld and Myers, 2003], thread usage policies [Sutherland and Scherlis, 2010], and application partitioning [Chong et al., 2007]. In the vast majority of these systems, including all of the examples just cited, the primary benefit of the annotation systems touted by their creators is either verification or automated code generation. The preconditions and postconditions that appear next to methods in Plaiddoc are essentially state-based type annotations. Therefore, this study provides indirect evidence that type based annotations can have benefits as documentation. The rest of this section discusses related work evaluating the benefits of types and suggests a template for evaluating the benefits of specialized type annotations.

4.7.1 Related studies on the benefits of types

In the last few years, there have been a flurry of studies, mostly from Stefan Hanenberg, comparing the benefits of static and dynamic types. In his first study, Hanenberg [2010] compared a synthetic statically-typed programming language to an equivalent dynamically-typed one using a between-subjects design involving two large groups of student participants, each of which spent 27 hours developing a Scanner and Parser. Participants in the dynamically-typed group produced the relatively-simple Scanner significantly faster then the participants in the statically-typed group. However, there was no similar difference for the whole project nor was there a significant quality difference between the code produced by either group.

In the next study, Stuchlik and Hanenberg [2011] narrowed the focus to type casts, used a within-subject design (a design choice repeated in each subsequent study), and found that small programming tasks involving significant casts using a statically-typed language were substantially slower than equivalent tasks using a dynamically-typed language. No difference was observed in larger tasks. Subsequent studies have found significant benefits of static types when performing software maintenance tasks [Hanenberg et al., 2013] and found no difference between generic and raw types on participant performance [Hoppe and Hanenberg, 2013]. All of this research suggests

that dynamic types have an advantage for small, greenfield tasks, while static types have an advantage for larger, maintenance tasks.

The most closely related study, performed by Mayer et al. [2012], evaluated the benefits of type annotations in undocumented software. The results were mixed—types were significantly helpful in some tasks, and significantly harmful in others. One possible interpretation of the results is that types were helpful in tasks that were more complex (involved more classes) and harmful otherwise. Our results provide a clearer picture — Plaiddoc provided benefits in every state-search category.

Mayer et al. [2012] differs methodologically in many ways from the study presented in this chapter. In their study, programmers performed programming tasks using two "structurally identical,"[12] synthetic,[13] undocumented APIs. In my study, programmers answered search questions with well-documented real-world APIs. One important consequence of these differences, is that our study evaluates types *only* for their documentation purpose, while theirs evaluates the collective value of both static-checking and types as documentation.

As in all but the first of the studies discussed in the first paragraph of this section, they use a within-subject design to control for the increased variance of programming tasks and use a repeated-measures ANOVA to correct for learning effects. We use a between-subjects design which does not require statistical correction. They investigate traditional Java-like types, while we investigate typestates which are an example of higher-level types specifications that commonly appear in the research literature but are almost non-existent in practice. It is my guess that these higher-level types provide more software engineering benefits as documentation than traditional types.

4.7.2 Research template

The studies discussed in Chapter 3 and this chapter suggest a template for studying the benefits of high-level type annotations. One important challenge with all programing experiments is the skill-variance between individual participants. As I observed in Section 4.5.6, this can be exacerbated when performing complex tasks like protocols. The targets of the other high-level type annotations I mentioned at the beginning of this section are at least as complex: security-critical communication (information

[12]The second API was "derived from the first by renaming classes, methods, variables, etc. This was done in a manner that ensured the two APIs, despite having the same complexity, seemed to address two different problems."

[13]The APIs were created for the study.

flow), parallel programming (thread coloring), and enterprise architecture (application partitioning). I do not think sophisticated statistical corrections will be enough to correct for the variance in these domains.

This research suggests an alternative path. First, use appropriate qualitative research techniques to find the most important barriers to programming in the domain (in our research, these are the question categories). Then, define small tasks derived from developer forums or other field work (the merged forum questions). These should be components of the real programming tasks in the domain (solving state search questions is required to debug protocols). Then, distill the intervention, for example the tool or type system to be evaluated, to its essential parts (Plaiddoc). Finally, compare participant performance on these small tasks when using the distilled system to participants using status quo system.

4.8 Conclusion

In this study I demonstrate the effectiveness of Plaiddoc documentation relative to Javadoc documentation in answering state-related questions. The barrier to entry for using the Plaiddoc tool are minimal—only 1-3 annotations are required per method. I annotated all three APIs in less than one day of work. The main barrier to using Plaiddoc in production is training programmers to consume the documentation effectively. Untrained participants in pilot studies were not able to use Plaiddoc effectively. Even basic protocol concepts were foreign to my participants before training. That said, the training I provided was very quick and required no specialized knowledge. Regardless, it seems clear that any mainstream language that adopts first-class state constructs should also adopt a Plaiddoc like documentation structure. More generally, our study shows that state-based type annotations provide documentation-related benefits even for well-documented code. Thus, our results open the door to future work investigating the documentation-related productivity benefits of type-like annotations in a broad range of domains.

Chapter 5

Implications of Empirical Results on the Design of Plaid

In this chapter I will assess the design of the Plaid programming language, introduced in Chapter 2, based on the empirical results discussed in chapters 3 and 4. In addition to the Plaid syntax and dynamics semantics discussed in this thesis I will also comment on Plaid's type system.

5.1 Plaiddoc vs. Plaid

We saw in Chapter 4 that Plaiddoc documentation provided substantial productivity and correctness benefits above Javadoc documentation for state-related tasks. It is natural to ask whether the Plaid language, which embeds similar concepts, would have a corresponding advantage over the Java language. Unfortunately, this question is too large to be answerable as stated. Each language (particularly Java) includes a huge ecosystem with runtime systems, libraries, and tooling which affect their usability.

Instead, let us a consider a much smaller question. How would programmers given a well-commented Java interface (no implementation code) answer state search questions relative to programmers given a well-commented, state-type annotated Plaid interface? To know for sure one would have to run an experiment aimed at answering exactly that question. However, consider that there are four main differences between this smaller question and the actual experiment conducted in Chapter 4. First, the look and feel of the documentations differ substantially. Second, the web documentation includes clickable hyperlinks that allows for quick navigation. Third, the web documentation includes a section that summarizes each member and a separate section that provides

details for each member. Fourth, The Plaiddoc ASCII state diagram does not appear in Plaid code.

The first three differences between web documentation and interface code seem unlikely to change the *relative* performance of the state-oriented language and Java. However, the ASCII state diagram is very different from the scattered keywords that define state relationships in Plaid. Our pilot results suggest that the state diagrams were very important for tasks that require understanding complex state relationships. Fortunately, most tasks do not require this understanding. Furthermore, at least in the case where all related states are known at compile time (an important detail we discuss further in the next section), state diagrams are reconstructible from the type annotations. Therefore, the differences found between Plaiddoc versus Javadoc are likely to translate to Plaid versus Java.

5.2 Plaid extensibility

The experiment in Chapter 4 presented all of the states composing each Java class in a single webpage. The more natural way to present Plaid, in which states are declared separately, would be to present one page per state. However, I am fairly certain that presenting Plaiddoc in this way would have significantly hampered participant performance. For example, participants with multi-page Plaiddoc would not have been able to use simple text search to find particular methods. They would also have wasted time switching between pages. Therefore, I think it is preferable to present documentation for related states together. I think this advice also applies to Java: I think it would be preferable for the method and field documentation from superclasses and super-interfaces to be included directly in the class documentation instead of merely linked.

One other feature of Plaiddoc does not relate well to Plaid — Plaiddoc included an ASCII state machine diagram with all of the states of each class. Unfortunately, since state composition is extensible, the state machines which client programmers see may be different than the state machines desired by API designers. One could generate Plaiddoc like pages from Plaid code via a whole program analysis run by client programmers. However, I think API documentation writers will want to see the *exact* documentation that clients see. For example, a writer may want to explain relationship between states with prose. Instead, I think it is preferable to add a new mechanism to Plaid which enables API designers to prevent extension of particularly

important components of the state machine. This mechanism would be like sealed in Scala [Odersky et al., 2006], or comprises in Fortress [Ryu et al., 2010]. It might even be preferable for this composition to be the default composition mechanism, although I worry that such a default would cut off unanticipated reuse.

Finally, I think the composition mechanism itself could use some improvement. The ASCII state diagram was critical for the success of Plaiddoc programmers. Similarly, readers of Plaid code will need to understand the state relationships to understand Plaid code. However, in existing Plaid code the state relationships are scattered. For example, in the ResultSet code described in Section 2.2.3, the composition is scattered across every substate in dozens of "case of" or "with" statements. These statements are necessary, but I think readers of Plaid code will find it very difficult to recover the full set of state relationships from these scattered statements. Instead, I suggest we add a new composition mechanism for Plaid, which, like the JSON file used to generate Plaiddoc, specifies the full state relationships in a single place. We can then change the meaning of the existing keywords so that they imply requirements on the eventual composition and do not enact the composition. For example, "ForwardOnly case of Direction" would mean that any time ForwardOnly appears in an object it must also include Direction as an or-parent.

5.3 Access permissions

Plaid's type system is not the focus of this thesis, but the studies discussed in Chapters 3 and 4 have implications for the type system. I therefore introduce a few Plaid type system concepts here to help the reader understand those implications.

I co-developed Plaid's first type system [Saini et al., 2010] based on the Plural type system [Bierhoff and Aldrich, 2007]. Others have since extended that initial work with a gradual type system [Wolff et al., 2011] and a modified permission system [Naden et al., 2012]. It is therefore not possible to write about **the** Plaid type system because there are many systems that could potentially claim that title. Instead, I am going to write about the features they have in common.

The software engineering benefit sought in all of these research projects is to check at compile time that API protocols are followed. In particular, if an API has a protocol then the type system guarantees that: 1) API client programmers do not violate the protocol and 2) API implementors correctly implement the protocol. All of these type systems are *sound*, so any protocol violation that *could* happen at runtime causes an

Table 5.1: Access permission definitions.

	Other aliases to this object can...		
This variable can be used to...	no aliases	only read	read and change state
only read	-	immutable	pure
read or change state	unique	full	share

error at compile time.

These systems require programmers to include two types of annotations in their API implementation (or interface) code:

1. *Typestate annotations*, which specify the abstract state of the object referred to by a program variable (including the receiver, method parameters, and fields). For a method, they are used to specify the abstract states that arguments must have when the method is called and which abstract states they will have when the method returns. The names of these annotations correspond to the names defined in Plaid state declarations.

2. *Access permissions*, which specify whether program variables can refer to objects that are *aliased*[1] and whether the object's abstract state can be changed by any aliases that do exist.

Typestate annotations directly support the software engineering goal of checking for protocol violations. For example, methods are sometimes only available when the receiver is in a certain abstract state, which is indicated by a typestate annotation on "this" where the method is declared. However, typestate annotations only appear at module boundaries (e.g. method preconditions and postconditions) and therefore the type system tracks the state of program variables inside method bodies. This tracking is necessary, because unlike traditional types, which do not change at runtime, typestates are *flow-sensitive* and do change at runtime. To make the tracking work, the type system needs to know if any of these variables are aliased and changeable by one of these aliases. Without this knowledge, it is impossible to determine whether, for example, a method call might use an alias to change the state of an object that is pointed to by a local variable and not passed to that method as an argument.

Plural provides the five types of access permissions shown in Table 5.1. A variable with a "unique" permission cannot be aliased at all, and it can be used to both read and change the state of the object to which it refers. A variable with an immutable permission can be infinitely aliased but cannot be changed. A variable with a pure

[1] An aliased object is an object that is pointed to by two different variables.

permission can only be used to read from the object but may be changed by a different alias. A variable with a full permission can both read and change the object and there may be other read-only aliases. Finally, a shared permission is readable and changeable and allows unlimited read/change aliases. The shared permission is exactly what is provided by standard type systems since they do not provide any alias control. Each of the iterations of the Plaid type systems provides a different subset of those five permission types.

Recall from Section 4.5.6, that participants in our Pilot studies of Plural performed slightly better without Plural than with Plural. Many programmers, including the mathematically sophisticated participants I recruited for those pilots, seemed to have trouble with the combination of access permissions and typestate annotations in Plural. In particular, participants confused Plural access permissions with the typestate annotations. For example, in one task all three participants thought "pure" was an abstract state.

This confusion was likely due in part to the fact that the study required participants to learn two new concepts (access permissions and typestate annotations) at once. More generally, these early results are a worrying sign for those hoping to layer specialized verification systems on top of one another.

In follow-up questions, participants seemed to also have more trouble with the access permission concept than the typestate concept. Participants were able to make the logical jump from the primitive state they are used to thinking about to the abstract states specified with typestate annotations. Participants were not used to thinking about aliases at all. This could be because of poor training or inexperience, but the participants we used were all PhD students with several years of professional experience. Two possibilities seems more likely: either it is fundamentally harder to reason about aliases than state, or programmers already reason about state frequently but only rarely reason about aliases.

In the face of all this, I think it would be best to redesign the Plaid type system such that the access permissions are not exposed to client programmers at all. This would obviously be limiting in certain use cases and a thorough expressiveness evaluation is necessary to know if this idea is even viable. That being said, I think Plaid with access permissions would be much more costly to learn and more burdensome to use than Plaid without access permissions. If it is impossible to isolate access permissions from clients, then I suggest instead limiting the access permissions to only the very simplest (e.g. only immutable and unique).

5.4 Missing transitions

Missing state transitions were a significant burden to programmers in all three of the studies I conducted. One particularly common source of missing state transitions are *type qualifier states* which objects enter into at construction time and then never leave. It is impossible to transition from one type qualifier state to another without creating a new instance. A turbo car option, forward-only result set, and unmodifiable list are all type qualifiers. These type qualifiers are likely symptoms of Java's language design. A language with a trait-like reuse mechanism or structural types like Plaid will likely produce fewer type qualifiers.

However, sometimes missing state transitions are unavoidable. For example, *life-cycle protocols* in which objects proceed monotonically through a series of steps do not allow "backward" transitions. This type of protocol is very common in software frameworks (e.g. an android "screen" object is initialized in steps by the framework). An example in our studies of this kind of protocol is URLConnection, which cannot transition back to the disconnected state. This kind of missing transition will exist in Plaid programs as much as Java programs.

To alleviate the problems associated with missing transitions, I suggest that Plaiddoc separate state transitioning methods from regular methods.[2] It will be easier to scan transition methods if they are separated and I see no competing advantage to keep them together. In Plaid code I suggest we extend the composition mechanism I suggested in 5.2 to include state transitions. State transitions will thereby be grouped in Plaid code as they are in the improved Plaiddoc.

5.5 Protocol barriers

In Section 3.2.4 I introduced five recurring barriers to using API protocols: missing state transitions, state tests, state independence, multi-object protocols, and terminology confusion. We've already discussed missing state transitions at length, but I will now discuss how Plaid interacts with each of the others.

Plaid includes a pattern matching construct which allows programmers to test the state of any object at runtime. It is therefore impossible for an API designer to forget (or intentionally exclude) a state test.

Independent states, which Harel refers to as *and-states*, elicit frequent questions in

[2]This idea was actually first suggested by an anonymous participant.

the forums. This issue also caused several incorrect answers among Javadoc participants in our user experiment. However, in Plaid the relationship between states is much more explicit. I noticed no particular challenge with this concept among Plaiddoc participants and it is therefore likely this barrier is "lower" in Plaid.

Plaid has no explicit support for multi-object protocols, so this barrier is likely to be high even in Plaid. However, some multi-object protocol designs could be converted into single-object protocols relatively easily. The trait reuse mechanism in Plaid makes this especially easy. Many multi-object protocols are the result of delegation- based reuse, which is very naturally converted to trait-based reuse.

Finally, I think Plaid can have a big impact on the terminology confusion barrier. A lot of terminology confusion results from references to unnamed abstract states. For example, the HSQLDB error message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown" makes references to primitive details of their API implementation. A Plaid error message, which would be automatically generated, would instead read something like "Cannot call the next method; the ResultSet must be in the ValidRow state." This message is clearer than the HSQLDB version on its own, but it also eliminates terminology confusion, because "ValidRow" is a named state in Plaid code (and by extension in in Plaid documentation). This type of terminology confusion caused several incorrect answers in the laboratory study, when Javadoc participants confused the Disconnected abstract state with a URLConnnection whose connection has timed out. None of the Plaiddoc participants suffered from this confusion because the Disconnected state is clearly identified with its particular meaning in Plaiddoc.

5.6 Conclusion

In this chapter, I argued that the benefits of Plaiddoc over Javadoc are likely to translate to Plaid vs. Java. I then used the empirical data to suggest three changes to Plaid: a "sealed" like mechanism for controlling extensibility, separation of state transitions from regular methods, and isolating access permissions from client code. Each of these changes requires substantial further investigation. Finally, I argue that the Plaid design lowers three categories of barriers uncovered in the forum mining: state tests, independent states, multi-object protocols, and terminology confusion.

Chapter 6

Future work

In this chapter I discuss future work. The research ideas I discuss in this chapter focus on addressing the limitations of the work in this thesis, extending the work in new directions, and adapting the work for new domains. The items in this chapter are not concrete proposals, so they focus more on questions than approaches.

6.1 Addressing limitations

As with all research, there are many limitations of the work in this thesis, but I think some specific questions are especially worthy of further investigation: 1) How common are protocol problems? 2a) What other classes of protocols exist? 2b) Are the challenges they present similar? and 3) Do the benefits of Plaiddoc extend to Plaid? I discuss each of these questions in turn.

6.1.1 How common are protocol problems?

The study discussed in Chapter 3 does not attempt to evaluate the significance of protocol problems. Protocols are the target of massive research effort, and it is therefore worthwhile to understand protocols more accurately. That said, our forum mining study only found 28 state questions among the 5000 that were looked at carefully. This does suggest the possibility that protocols problems are not as problematic as the research community believes. Therefore, it is worthwhile to conduct a different study aimed directly at determining the commonality of protocol problems. I believe the biggest problem of conducting such research will be finding a reasonable sample of *all* programming problems to serve as a denominator. However, defining a research

methodology that enables estimation of the importance of a programming problem has extremely broad applications and is therefore a worthy goal.

6.1.2 What other classes of protocols exist? Are the challenges they present similar?

All of the APIs we looked at in Chapters 3 and 4 are libraries and involve *resource programming.* This was a natural result of narrowing our focus to the Java Standard Library. However, it is likely using frameworks or programming in non-resource domains will present very different challenges to developers. For example, using a framework often involves implementing an interface whose methods will be called by the framework. In many cases, values in fields are initialized in several steps during the lifecycle of the object. None of the APIs I looked at in this thesis presented this kind of initialization challenge. Therefore, conducting observations of uses of framework APIs may result in very different information seeking. Another potentially interesting class of protocols not investigated in this thesis are those in distributed systems. Distributed systems often impose ordering constraints on the messages that pass through communication channels. Since these are very different than the object protocols I studied in this thesis they are likely to present very different barriers to programmers.

6.1.3 Do the benefits of Plaiddoc extend to Plaid?

As I argued at length in Section 5.1, many of the benefits of Plaiddoc evaluated by my laboratory experiment are likely to extend to Plaid. However, the only way to know for sure is to conduct another experiment comparing Plaid to Java. There are many challenges that need to be overcome to make such an experiment work. Assuming the goal of this experiment is compare the languages themselves and not their associated tooling, the experiment would likely require an "artificial" development environment and programming tasks that do not give Java an advantage based on library support. In addition, the tasks themselves would need to be balanced for difficulty to ensure that a difference can be observed while still being relevant to the world outside the laboratory. Finally, training participants to use Plaid is likely to be much more challenging than training them to use Plaiddoc. It will be especially difficult to train programmers so that their proficiency in Plaid roughly matches their proficiency in Java.

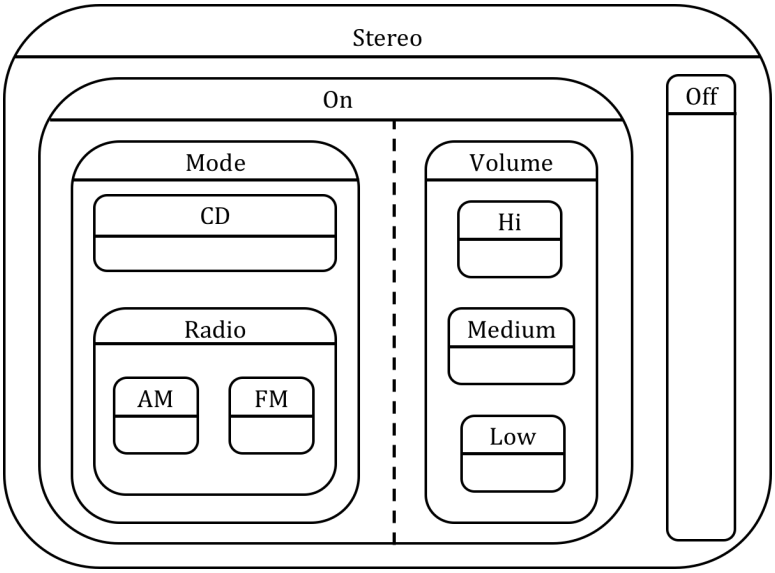

Figure 6.1: Stereo state machine.

6.2 Extensions

6.2.1 How can we better support state history in Plaid?

The Plaid language embodies all of the features of Harel state charts, with one notable exception — history states. History states allow state charts to model "memory" conveniences of real-world systems. For example, consider a stereo modeled by the state machine shown in Figure 6.1. Imagine Pat is using the stereo and is listening to AM radio, then switches to FM radio, and then switches to the CD. If Pat then hits the radio button, Pat expects to return to FM radio—a state that should be "remembered" by the stereo. This kind of behavior is modeled by history states.

In Plaid, fields are reset to their default value after reentry into a state. Programmers are therefore forced to keep track of field values they want to maintain after state change. This is very inconvenient and therefore it makes sense to add explicit support for history states in Plaid to enable more convenient tracking of states.

6.2.2 Which features of Plaiddoc provide the most benefits?

Plaiddoc contains four major features: state type preconditions, state type postconditions, the ASCII state diagram, and method organization by state. The experiment in Chapter 4 was not designed to figure out which of these is most important. I suspect that much of the benefit of Plaiddoc comes in the particular combinations of features,

but I might be wrong. The right way to tease apart the different features is to perform another study in comparing groups of participants that are given modified Plaiddoc with just one feature or just some combinations of features.

6.3 Adaptations

6.3.1 How can we study the usability of other high level type systems?

I suggested in Section 4.7.2 that the studies in Chapters 3 and 4 can serve as a template for studying other high level type systems. I am eager to test this suggestion by applying it to access control specifications. Our research group is actively designing and building Wyvern, a programming language aimed at enforcing security properties in web and mobile applications [Nistor et al., 2013]. Wyvern therefore provides an excellent testbed for testing type-based access control specifications and their usability implications.

6.3.2 How can we support simultaneously active high level type systems?

In the pilot studies I discuss in Section 5.3 participants using Plural had trouble distinguishing the access permission and typestate annotations. This suggests that a set of high-level type systems which are usable independently, may become much less usable in combination. Combining high-level type systems effectively is likely a great challenge on its own. Thankfully, there is a lot of research in related areas which can be used as building blocks. The software architecture community has long supported multiple architectural views of the same system [Clements et al., 2002]. Similarly, there is a huge body of research in software visualization which takes advantage of sophisticated program analysis to present programmers with the right information in the context in which they need it [Price et al., 1993]. These techniques and others like them are likely to be helpful when combining high-level type systems.

Chapter 7

Conclusion

Researchers have been building tools, analyses, and languages to check API protocols for decades. However, like the human aspects of programming generally, API protocol usability has been very understudied. In other words, researchers have been building tools to address a problem they did not fully understand. This thesis addresses this limitation by directly studying programmers using APIs with protocols in the two-part qualitative study discussed in Chapter 3. These studies make the following three contributions to our understanding of API protocol usability:

- The questions asked by forum participants related to API protocols contained the following recurring problems: missing state transitions, state tests, state independence, multi-object protocols, and terminology confusion.
- In our observations of programmers using APIs with protocols, most of the programmers' time were spent performing four categories of state search. The categories were: A) What abstract state is the object in? B) What are the capabilities of an object in state X? C) In what state(s) can I do operation Z? D) How do I transition from state X to state Y?
- In our observations, when programmers fix protocol violations, the first place they look is at the documentation related to the method call occurring at the violation location.

I hope these qualitative results refocus the research communities efforts toward addressing these common barriers, solving state search problems, and providing instruction at method call sites.

In this thesis, I design, develop, and evaluate two programming interventions—the programming language Plaid and the documentation generator Plaiddoc—designed to

improve the usability of APIs with protocols. The design of Plaid and Plaiddoc are based on Java and Javadoc, modified so that abstract states are first-class. The first-class state features enable programmers to efficiently answer state search questions.

This thesis contributes the concrete language design and evaluation of the Plaid language. The Plaid language, as we demonstrate by the examples in Chapter 2, has the following benefits:

- Stateful designs (e.g. those illustrated by state machines in design documentation) are clearly reflected in Plaid code.
- State constructs are enforced by the Plaid semantics, and thus there is no need for programmers to explicitly check for protocol violations.
- Protocol violation error messages refer to explicit state constructs.
- The fact that state is a high-level concept exposes new candidates for reuse like open/closed and position within a stream.

In combination, these benefits combine to enable API designers to better understand the protocols they impose on clients, and enable client programmers to find state information more easily.

The main contribution of Plaiddoc is that it enables an empirical evaluation of the impact of state organization and state type annotations on API client state search performance. The experiment in Chapter 4 compares Plaiddoc to Javadoc in a series of state search tasks. The study found:

- Plaiddoc participants were significantly faster than Javadoc participants at answering state search questions.
- Plaiddoc participants and Javadoc participants were equally fast at answer non-state search questions.
- Plaiddoc participants answered state search significantly more correctly than Javadoc participants.

These results clearly show that Plaiddoc improves on the status quo when performing state search tasks. Since Plaiddoc embodies many of the same state search features as Plaid, it also suggests that Plaid is an improvement on the status-quo for using API protocols. Finally, since Plaiddoc's state features are essentially high-level type annotations—these empirical results suggest that high level type annotations can have substantial benefits as documentation above and beyond types' benefits as enablers of verification.

Appendix A

Plaid language semantics

A.1 Introduction

In this appendix we present the formal definition of the Plaid language and give it a precise semantics. At its core, Plaid is an object system with first-class generators[1] and functions. Individual generators can be combined and specialized using composition and operators inspired by traits [Ducasse et al., 2006][2], instantiated to create objects, or used to specify the abstract state the object should change to. We start by describing the syntax and object model of a core language, which is intended to be simpler than Plaid source code yet be capable of representing all of the major semantic elements of Plaid. Then we discuss the execution semantics of the core language.

A.2 Core Syntax

The syntax of the internal representation of Plaid is given in Figure A.1. In these definitions, x ranges over bound variables, while members of objects are represented by $f, m,$ and s, which respectively range over fields, methods, and state members. We use n to represent any kind of object members when we do not distinguish between them. Abstract states are represented using tags which are generated as needed. We will introduce each syntactic category in turn, describing its purpose and motivations.

[1] Generators are functions that return a sequence of values.
[2] A trait is a collection of methods which can serve as a building block for classes or objects.

Obj Val	ov	$::=$	$mv \mid dv \mid mv \curlyvee ov \mid dv \curlyvee ov$
Dim Val	dv	$::=$	$tag\{ov\} \mid tag\{ov\} <: dv$
Mbr Val	mv	$::=$	$\textsf{method}\ m(\overline{x})\{e\} \mid$
			$\textsf{val}\ n = v$
ObjExp	oe	$::=$	$me \curlyvee oe \mid de \curlyvee oe \mid e \curlyvee oe \mid$
			$me \mid de \mid e$
Dim Exp	de	$::=$	$dv \mid tag\{oe\} \mid tag\{oe\} <: de \mid$
			$e \mid e\{\overline{to}\}$
Mbr Exp	me	$::=$	$mv \mid \textsf{val}\ f \rhd x = e \mid$
			$\textsf{recstate}\{\overline{\textsf{val}\ s \rhd x = \textsf{proto}\ sd}\}$
State Decl	sd	$::=$	$\textsf{freshtag}\{oe\} <: de \mid$
			$\textsf{freshtag}\{oe\} \mid oe$
Trait Op	to	$::=$	$\backslash n \mid n \rightarrow n' \mid me \mid (\textsf{tagOf}\ e).me$
Val	v	$::=$	$\ell \mid ov \mid \textsf{proto}\ oe \mid \textsf{fn}(\overline{x}) \Rightarrow e$
Exp	e	$::=$	$x \mid v \mid \textsf{let}\ x = e\ \textsf{in}\ e \mid$
			$e(\overline{e}) \mid e.m(\overline{e}) \mid e.n \mid$
			$e \leftarrow e \mid e \twoheadleftarrow e \mid \textsf{new}\ e \mid$
			$\textsf{match}(e)\{\overline{c}\} \mid$
			$\textsf{freeze}\ e \mid \textsf{recstate}\{\overline{mv}\}\#\mathrm{l}$
Case	c	$::=$	$\textsf{case}(\textsf{tagOf}\ e)\ \{e\} \mid \textsf{default}\ \{e\}$

Figure A.1: Internal Syntax

A.2.1 Expression Syntax

Plaid contains the standard expressions found in object systems, including object creation through new, field selection, and method calls. Because Plaid also has first-class functions, we include standard function definition and application as well. For sequential expressions, we include let bindings and bound variable references.

The rest of the expression forms are related to Plaid's encoding of abstract states and the transitions between them:

Changing state. The Plaid core has two state change operators. $\leftarrow$ represents a state update and only removes portions of the receiving object that are mutually exclusive with the incoming states. For completeness and flexibility, Plaid also includes a state replacement operator, $\twoheadleftarrow$, which wipes the receiving object clean before adding the incoming states, much like an in-place new operation. One could imagine using this operator in a situation where an object needed to be in a particular state and no other states. This cannot be guaranteed by the state update operator because state update leaves dimensions unrelated to the updating state alone.

Unlike the source language, Plaid's core does not require the target of a state change operator to be *this*. This makes the core simpler and more flexible since the restriction can be enforced at the source level.

proto values. First class instance generators are provided by proto expressions. These are values which can be stored in fields and passed as parameters. During a well-formed execution, the target of new expressions and the right-hand side of state change expressions will evaluate to a proto value. This is because they encapsulate object expressions, oe, which are uninitialized objects. The state change and new expressions cause the initialization steps specified by the object expression wrapped in the proto to be evaluated for use in creating a new object or changing the state of an existing one.

State expressions. To allow states to be chosen dynamically at runtime, we include several expression forms that can evaluate to a proto. As they are values, standard deference or bound variables could result in proto expressions. Because most states included in protocols must be defined with (mutual) recursion, proto values represented source-declared states are wrapped into a recstate. A particular proto can be selected from the recstate as from a standard record.

The freeze expression is a more novel way to get a proto. It takes the object and wraps it up in a proto allowing new instances to be generated from it. As an example of the use of freeze, consider the myResultSet value defined in Listing 2.3. Say we wanted to do some extra initialization of the ResultSet before using it and that over the course of a program we would create the same ResultSet over and over. To avoid needing to do the same initialization repeatedly, one could freeze the object the fully initialized object and then instantiated it each time a new ResultSet of this form was needed. freeze has already been used in the Plaid compiler to more cleanly support certain initialization paradigms, such as the transformation to let-normal form, where strings of let bindings must be concatenated together.

Matching tags. Finally, the match construct allows pattern matching based on tags. Each case tests the target object against the tagOf another expression. This expression is expected to evaluate to a proto value with a single outer tag which is grabbed by tagOf and compared with the tags of the target object. If the object contains the tag, the corresponding case is executed. Cases are evaluated in order.

An example of the use of match comes from the Plaid standard library. Plaid's syntax does not include control structures. Instead, if and while are encoded as functions that make use of match. The states True and False are each defined as a case of Boolean. Thus, the if function determines whether or not to evaluate the body based on whether the object returned by the condition matches the True tag.

A.2.2 Object Value Syntax

Plaid objects are collections of tags representing the states that the object is in along with fields and methods that provide the representation and operations of those states. In order to implement the desired semantics, these object must be organized to formally encode the relationships between tags and members that the semantics depend on. In particular, we need to represent the following relationships between the abstract states that the tags represent:

1. *Superstates*: An object in state, S, which is defined to be a case of a superstate, T, must also be in state T. For instance, an object in the NotEnd state defined in figure 2.4 is also in the Position state.
2. *Or-states*: Distinct cases of a given state, such as the OpenFile and ClosedFile case of File, cannot exists together in an object.

3. *And-states*: Both objects and states can be defined as a composite of other states. For example, the Open state from Listing 2.3 is defined in terms of states Direction, Status, and Action. Objects in the composite state are considered to be in each of the component states as well.
4. *Defining states*: Members must be associated with the state that declares them so that they can be removed from an object when their defining state is removed.

To formalize these relationships, *objects values* are organized as hierarchical collections of *dimensions*, which contain tags for the state and all of its transitive super states, and *members*.

Object values. The basic component of an object is an object value, ov, which is a list of dimension values, dv, and member values, mv. They are used to represent both top-level objects and the dimensions and members that define a given state (see dimension values below). The $\gimel$ operator that separates each element of the list represents composition. Object values encode and-states by allowing two dimensions to coexist together inside the definition of a state. For instance, the object value that defines a ReadStream would have two composed dimensions, one for the Position dimension, and the other for the Reader dimension.

Dimension values. Dimension values, $tag\{ov\}[<: dv]$, encode the structure of a state and its super states. They are represented by a tag, tag, which is a unique name for the most specialized state from the dimension. Associated with the tag is an object value which represents the collection of members that the state defines along with any other dimensions that make up the and-states of the state. A dimension value may optionally contain another dimension value encoding the superstate relationship.

By containing the representation of a given states' superstates, dimension values give us a way to encode the or-state relationship as well. Two states that are the case of the same superstate would be encoded as separate dimensions with the same state at the root of the dimension. Because the tags in the dimensions partially overlap, by restricting tags to appear only once in a given object value, we can ensure that no or-states can coexist in a single object.

Concretely, we would represent an instantiated Open state from Listing 2.3 as

$$Open\{Direction \gimel Status \gimel Action\} <: ResultSet.$$

Here, the most specific state is Open, which specializes the ResultSet state and is defined with the three states Direction, Status, and Action.

A dimension is also Plaid's version of a trait. Multiple inheritance is achieved by allowing multiple dimensions to be composed in an object value as well as in the object values associated with the tags of a dimension. The hierarchical nature of Plaid's dimension prevent us from using all of the trait mechanisms for solving the problems of multiple inheritance. In particular, a multiple inheritance system must deal with the case when one class inherits from two classes that share a (transitive) parent. This situation is challenging because it is non-obvious how to inherit members from the common grandparent. This problem is commonly referred to as the *diamond problem* [Malayeri and Aldrich, 2009], because of the shape of the inheritance hierarchy diagram. The original traits proposal [Ducasse et al., 2006] *flattens* [3] composed traits and forces any conflicts between method names to be explicitly resolved (field were not allowed in traits). However, as Plaid's semantics depend on members being related to the tag they are defined in, we cannot use flattening. Instead, Plaid prevents the diamond problem by preventing or-states from coexisting, thereby preventing the same tag and member definition from appearing more than once (following Malayeri's no-diamonds rule [Malayeri and Aldrich, 2009]). Plaid's solution follows recent extensions of traits including [Bergel et al., 2008; Reppy and Turon, 2007; Cutsem et al., 2009]. Like Plaid, these system support traits with fields and work in a variety of object models including those that, like Plaid, add hierarchy and do not enforce the flattening property. As with the original trait proposal, all name conflicts across dimensions must be explicitly resolved in Plaid via the trait operators described below.

Member values. A member value is either a method, with a set of arguments and a body, or a field, val f, bound to a value, v. The member is said to be defined in the state represented by its immediately enclosing tag. As a concrete example, an object in the ClosedFile state described in Listing 2.1 would be represented formally as

$$\texttt{ClosedFile}\{\textsf{method}\ \mathit{close}()\{e\}\}$$
$$<:\ \texttt{File}\{\textsf{val}\ \mathit{filename} = v\}$$

This indicates that the object is in both the ClosedFile and the File states, one of which

[3]The flattening property in Ducasse et al. [2006] states that each object member is treated equally regardless of the trait in which it was defined.

is a substate of the other, and each of which defines a single member.

Uninitialized Object Syntax

Plaid has corresponding syntax for uninitialized objects organized into *object expressions*, *dimension expressions*, and *member expressions*. When compared to their value counterparts, they share the same structure but contain expressions which are not yet values. In this section, we discuss the places where execution can occur in these forms and the motivation behind them.

Object expressions. Object expressions, oe, are made up of the composition of dimension expressions, member expressions, as well as raw expressions. The purpose of unevaluated expressions in dimension and member expressions will be explained below. Raw expressions as components of object expressions allow part of an uninitialized object to be determined at the time of initialization. These expressions evaluate to proto values which are then incorporated into the initializing object. This provides for Plaid's implementation of dynamic trait composition by allowing portions of the object to be selected at runtime.

Dimension Expressions Dimension expression can contain unexecuted expressions in the object expression associated with the most specific tag as well as in tags up the hierarchy if they exist. Dimension expressions may also have associated trait operations, to, which need to be evaluated. Trait operations allow standard manipulations such as renaming, $n \rightarrow n'$, and removal, $\backslash n$. Note that these operate on the *whole* dimension, renaming or removing all members of the specified name defined directly in tags in the hierarchy (not including nested dimensions). This allows the changes to be preserved by state change in the dimension as we will see below.

Members can also be added or replaced.[4] By default, they are (re)placed in the most specific tag of the dimension expression. However, in cases where members need to be added or replaced in a particular tag, they can be qualified by a particular tag, specified as with tags in case statements by tagOf another expression. The redefinition of Position.EndState for the ReadStream in Listing 2.5 is an example of using qualified trait operations. This mechanism is important in Plaid because of the hierarchical nature

[4]The semantics defined here do not allow fields and states in trait operations to refer to other trait operation members. The formalism could be extended to support this, mirroring the case for declarations in states.

of Plaid's object model and when and how member definitions are removed during state change.

Member expressions. Only fields can be member expressions, me, as methods do not have any initialization code. On the other hand, fields can be defined with initialization expressions that require evaluation as a part of object creation or update. In order to allow fields to refer to the initialized value of previous fields in the same state, field expressions define an internal bound variable in addition to their external name (this is a standard approach from [Pierce, 2005], chapter 8). Fields are also generated by state declarations. Since the definitions of related states, such as the OpenFile and ClosedFile from Listing 2.1, are typically recursive, the initialization of state members occurs in a recstate binding.

State members are also special in that when an uninitialized object containing state members is initialized, new tags may need to be generated. The proto expression encapsulates uninitialized objects as discussed above. Normally they contain object expressions, but when appearing in a recstate, they contain state declarations, sd which may contain the freshtag operations that generates a new tag when executed, resulting in an object expression. This feature means that new tags are generated for states defined inside states each time the outer state is instantiated. Because these tags can then be used to pattern match on objects, this allows Plaid to implement ML-style generative functors[5]. Functors have well recognized modularity benefits that we do not discuss here.

A.3 Dynamic Semantics

We now introduce the dynamic semantics of Plaid. We formalize the execution using a small step operational semantics. The basic evaluation judgment has the form $e@H \mapsto e'@H'$ and is read "expression e with heap H evaluates to expression e' in heap H'". We define a similar judgment $oe@H \mapsto oe'@H'$ for the evaluation of object expressions. In this section, we will define the form of the heap and the invariants that we maintain on it. We will also discuss the Plaid-specific evaluation rules, in particular those that use ancillary judgments for implementing state change. As state change is at the core of Plaid's design and is the most complicated we go into depth about the motivation

[5]Generative functors, in contrast to applicative functors, generate new abstract types for each application of the functor. This impacts pattern matching when using these generated types in a similar way as pattern matching on freshly generated tags in Plaid.

$$
\begin{array}{lcll}
\text{Heap} & H & ::= & [\ell \rightsquigarrow ov], H \mid \cdot \\
\text{Eval} & E & ::= & [\,] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid \\
 & & & E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid E.f \mid E \leftarrow e \mid \\
 & & & v \leftarrow E \mid v \leftarrow \mathsf{proto}\ E \mid E \twoheadleftarrow e \mid \\
 & & & v \twoheadleftarrow E \mid v \twoheadleftarrow \mathsf{proto}\ E \mid \mathsf{new}\ E \mid \\
 & & & \mathsf{new\ proto}\ E \mid \mathsf{match}(E)\{\overline{c}\} \mid \\
 & & & \mathsf{match}(v)\{\mathsf{case}(\mathsf{tagOf}\ E)\ \{e\}, \overline{c}\} \mid \\
 & & & \mathsf{freeze}\ E \mid ov \curlyvee E \mid O \curlyvee oe \mid O \\
\text{Obj} & O & ::= & \mathsf{val}\ n \rhd x = E \mid tag\{oe\} <: E \mid \\
 & & & tag\{E\} \mid tag\{E\} <: dv \mid E\{\overline{to}\} \mid \\
 & & & dv\{\overline{to}, \mathsf{val}\ n = E, \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\mathsf{tagOf}\ e).(\mathsf{val}\ n = E), \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\mathsf{tagOf}\ E).mv, \overline{to}\}
\end{array}
$$

Figure A.2: Contexts

$\boxed{e@H \mapsto e@H}$

$$
\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e@H \mapsto e[v/x]@H}\ \text{E-LET}
\qquad
\frac{|\{\overline{x}\}| = |\{\overline{v}\}|}{(\mathsf{fn}\ (\overline{x}) \Rightarrow e)(\overline{v})@H \mapsto e\overline{[v/x]}@H}\ \text{E-APP}
$$

$$
\frac{\begin{array}{c} H[\ell] = ov \\ \mathtt{lookup}(m, ov) = (\mathsf{method}\ m(\overline{x})\{e\}) \\ |\{\overline{x}\}| = |\{\overline{v}\}| \end{array}}{\ell.m(\overline{v})@H \mapsto e[\ell/\mathsf{this}]\overline{[v/x]}@H}\ \text{E-CALL}
\qquad
\frac{\begin{array}{c} H[\ell] = ov \\ \mathtt{lookup}(f, ov) = (\mathsf{val}\ f = v) \end{array}}{\ell.f@H \mapsto v@H}\ \text{E-FIELD}
$$

$$
\frac{\begin{array}{c} H[\ell] = ov_1 \qquad \mathtt{uniqueTags}(ov_2) \\ ov_1 \leftarrow ov_2 \Rightarrow ov_3 \qquad \mathtt{uniqueMembers}(ov_3) \end{array}}{\ell \leftarrow \mathsf{proto}(ov_2)@H \mapsto \mathsf{void}@H[\ell \rightsquigarrow ov_3]}\ \text{E-SU}
\qquad
\frac{\mathtt{uniqueTags}(ov) \qquad \mathtt{uniqueMembers}(ov)}{\ell \twoheadleftarrow \mathsf{proto}(ov)@H \mapsto \mathsf{void}@H[\ell \rightsquigarrow ov]}\ \text{E-REPLACE}
$$

$$
\frac{\begin{array}{c} \ell \notin H \qquad \mathtt{uniqueTags}(ov) \\ \mathtt{uniqueMembers}(ov) \end{array}}{\mathsf{new}\ (\mathsf{proto}\ ov)@H \mapsto \ell@H[\ell \rightsquigarrow ov]}\ \text{E-NEW}
$$

$$
\frac{de = tag\{oe\}[<: de'] \qquad tag \notin \mathrm{tags}(H[\ell])}{\mathsf{match}(\ell)\{\mathsf{case}\ (\mathsf{tagOf\ proto}\ de)\{e\}, \overline{C}\}@H \mapsto \mathsf{match}(\ell)\{\overline{C}\}@H}\ \text{E-CASENOMATCH}
$$

$$
\frac{de = tag\{oe\}[<: de'] \qquad tag \in \mathrm{tags}(H[\ell])}{\mathsf{match}(\ell)\{\mathsf{case}\ (\mathsf{tagOf\ proto}\ de)\{e\}, \overline{C}\}@H \mapsto e@H}\ \text{E-CASEMATCH}
$$

$$
\frac{}{\mathsf{match}(\ell)\{\mathsf{default}\{e\}, \overline{C}\}@H \mapsto e@H}\ \text{E-CASEDEFAULT}
\qquad
\frac{H[\ell] = ov}{\mathsf{freeze}\ \ell@H \mapsto \mathsf{proto}\ ov@H}\ \text{E-FREEZE}
$$

$$
\frac{\mathrm{l} = \mathrm{l}_s \qquad oe = oe_{\mathrm{l}_s}\overline{[\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#\mathrm{l}_i/s_i]}}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#\mathrm{l}@H \mapsto \mathsf{proto}\ oe@H}\ \text{E-RECSTATESELECT}
$$

Figure A.3: Expression Evaluation

and design of the rules that implement it. Finally, we describe object initialization and trait operations that may be involved.

A.3.1 Heap

A heap, H, is a mapping from locations, ℓ, to object values. We place additional well-formedness requirements on all object values stored in the heap. These restrictions prevent ambiguities from multiple inheritance.

Tag uniqueness. We require that all well-formed object values have no duplicate tags. As alluded to above, this property ensures that an object is not in two cases of a single or-state at the same time. This is because the tags representing two mutually exclusive or-states must come from the same dimension and thus must have at least the dimension tag in common. It also prevents the diamond problem of multiple inheritance by ensuring that a particular member definition does not appear multiple times in a single object. This invariant is encoded in the helper judgment `uniqueTags` also defined in Figure A.7.

Member uniqueness. Even though a given definition for a member cannot appear more than once, it is still possible that multiple tags define members with the same name. To prevent ambiguities in this case we require that all members of an object are provided by exactly one dimension. Because the hierarchy of dimensions gives us a natural way to choose the visible definition (the one from the most specific tag in the dimension) we allow a single name to be defined directly in multiple tags from a single dimension. Formally, two tags are in the same dimension if one is a transitive case of the other. This relaxation of classical traits allows, for instance, a common super state to define a default behavior for a method which can be overridden by (some of) its substates. The judgment `uniqueMembers` defined in Figure A.7 captures this requirement. It uses the judgments $mv :: tag.x@ov$, which states that member value mv from tag tag defines name x in object value ov, and $tag << tag'@ov$ which asserts the property that tag tag is a transitive subtag of tag' in object value ov. Based on these helper judgments, an object value has unique members if whenever we find the same member defined in two tags, then one of these tags is a transitive subtag of the other. We prove that evaluation preserves member and tag uniqueness in Appendix B.

Member lookup. As an object can contain multiple members with the same name, we need an unambiguous way to choose which one is visible. The `lookup` function also in Figure A.7 defines this logic. When multiple definitions are found, we know by `uniqueMembers` that they all come from the same dimension. Since the tags of a dimension form a total order, we know that one of tags defining the member will be a transitive subtag of all other tags defining the member. The definition from this most specific tag is the one returned by `lookup`.

A.3.2 Expressions

The evaluation rules for expressions in Plaid are given in Figure A.3. We list only computation rules here, defining congruence rules using evaluation contexts shown in Figure A.2. In these, each expression with a subexpression that requires evaluation defines a hole, [ ], into which any expression can be placed. Evaluation proceeds by using the computation rule that evaluates the expression in the hole.

Standard rules. The computation rules for the evaluation of Plaid expressions are borrowed from standard object models and the lambda calculus. They should therefore be very familiar for readers familiar with programming language semantics. These include the rules E-LET, E-APP, E-CALL, and E-FIELD for let expressions, application, method calls and field dereferences respectively. One note is that member selection during calls and dereferences use the `lookup` judgment described above. We also use standard record evaluation rules when selecting a label from a recstate (E-RECSTATESELECT).

Match. Plaid uses a first-match semantics, so that we find the first case clause whose tag matches the target object. We find the tag to match against by grabbing the most specific tag (tagOf) from a dimension expression wrapped in a proto value. Note that in this case the dimension expression is not evaluated since we are only interested in the tag. If the tag is found in the target object, the code for this case is evaluated (E-CASEMATCH); otherwise, execution proceeds to the next case (E-CASENOTMATCH). Default cases are always executed and terminate the match if reached (E-CASEDEFAULT). Evaluation gets stuck if no matching case is found.

Freezing. To freeze a location in the heap (E-FREEZE), we simply pull the object value from the heap and wrap it in a proto expression.

$\boxed{ov \leftarrow ov \Rightarrow ov}$

$$\frac{ov_t \leftarrow ov \Rightarrow ov' \qquad ov' \leftarrow ov_u \Rightarrow ov_o}{ov_t \leftarrow ov \curlyvee ov_u \Rightarrow ov_o}\ \text{SU-LIST}$$

$$\frac{}{ov_t \leftarrow mv_u \Rightarrow ov_t \curlyvee mv_u}\ \text{SU-MV}$$

$$\frac{\text{tags}(ov_t) \cap \text{tags}(dv_u) = \emptyset \qquad \texttt{uniqueTags}(dv_u)}{ov_t \leftarrow dv_u \Rightarrow ov_t \curlyvee dv_u}\ \text{SU-ADDH}$$

$$\frac{\begin{array}{c}\text{tags}(dv) \cap \text{outerTags}(dv_u) \neq \emptyset \\ dv \leftarrow dv_u \Rightarrow dv_r \\ \text{tags}(ov) \cap \text{tags}(dv_r) = \emptyset\end{array}}{ov \curlyvee dv \leftarrow dv_u \Rightarrow ov \curlyvee dv_r}\ \text{SU-MATCHDIM}$$

$$\frac{\begin{array}{c}\text{outerTags}(dv_u) \cap \text{tags}(ov) \neq \emptyset \qquad ov \leftarrow dv_u \Rightarrow ov_r \\ [\text{tags}(dv_u) \cap \text{tags}(dv) = \emptyset] \qquad tag \notin \text{tags}(dv_u)\end{array}}{tag\{ov\}[<: dv] \leftarrow dv_u \Rightarrow tag\{ov_r\}[<: dv]}\ \text{SU-MATCHINNER}$$

$$\frac{\begin{array}{c}\text{outerTags}(dv_u) \cap \text{innerTags}(dv) \neq \emptyset \qquad dv \leftarrow dv_u \Rightarrow dv_r \\ \text{tags}(tag\{ov\}) \cap \text{tags}(dv_u) = \emptyset\end{array}}{tag\{ov\} <: dv \leftarrow dv_u \Rightarrow tag\{ov\} <: dv_r}\ \text{SU-MATCHSUPERINNER}$$

$$\frac{tag \notin \text{outerTags}(dv_u) \qquad \text{outerTags}(dv_u) \cap \text{outerTags}(dv) \neq \emptyset \qquad dv \leftarrow dv_u \Rightarrow dv_r}{tag\{ov\} <: dv \leftarrow dv_u \Rightarrow dv_r}\ \text{SU-MATCHSUPER}$$

$$\frac{dv_u = [dv_{sub}] <: tag\{ov'\} <: [dv_{sup}] \qquad [\text{tags}(dv_{sub}) \cap \text{tags}(tag\{ov\}[<: dv]) = \emptyset] \qquad \texttt{uniqueTags}(dv_{sub})}{tag\{ov\}[<: dv] \leftarrow dv_u \Rightarrow [dv_{sub}] <: tag\{ov\}[<: dv]}\ \text{SU-MATCH}$$

Figure A.4: State Update

Manipulating objects in the heap. The state change operators and new cause objects in the heap to be changed or allocated. Because we only allow object values to appear in the heap, we must first initialize the object that will be used to alter the heap by reducing it to an object value. Evaluation is mostly handled by the evaluation contexts: first the expression representing the object is reduced to a proto value and then the object expression wrapped in the proto is evaluated down to an object value. An important design decision in Plaid was to run the initializers for all members of an object expression. This happens despite the fact that not all members may end up in the object (see the explanation of state update below). In particular, any effectful initializers will always be run and update the wider context. We experimented with other possible semantics but decided that a clear and unambiguous rule for when initializers were run (always) was better than a flexible but complicated one. Furthermore, we consider it good Plaid style to avoid the use of effectful initializers and instead use other design techniques, such as factory methods, when effectful operations are required as a part of object initialization.

Once the initialization code in the proto has been run, the resulting object value can be used to update the heap. In the case of new and state replacement (↞) expressions it is clear what the object value that is inserted into the heap will be. new allocates a new location on the heap and maps it to the resulting object value. State replacement

```
val rs = Open {
    Inserted <: Inserting <: Action,
    Scrollable <: Direction,
    Updatable <: Status }
<: ResultSet
```

Listing A.1: Open, Inserted, Scrollable, Updatable ResultSet

replaces the mapping of the target location on the heap with the updating object value. Since we know the precise form of the object value that is being inserted into the heap, in order to maintain the heap invariants on object values, we can simply check that `uniqueTags` and `uniqueMembers` both hold on the new object value as done in the rules E-REPLACE and E-NEW. On the other hand, the semantics of updating an object on the heap using state update are much more complicated, and so we devote the next section to a discussion of its design and proof that they maintain the necessary invariants.

A.3.3 State Update

At the core of the rule E-SU which updates the heap with the result of a state update is the state update judgment, $ov \leftarrow ov \Rightarrow ov$, which is described in Figure A.4. The judgment takes two object values and determines the resulting object value when the *target* object on the left side of the arrow is changed to the state given by the *update* object from the right side. The semantics of this judgement are the most complicated and important part of Plaid's dynamic semantics. Thus, before describing the semantics given by the rules, we step back and give a high-level overview of the desired behavior. We then define some general properties and assumptions of the judgment before describing the rules themselves.

Design considerations. Our goal is that the design of the state change judgment should match the semantics of stateful abstractions as modeled by state charts and similar tools. Thus, a state update should transition a target object from its current set of abstract states to a possibly new set of abstract states as specified by the update object. To do this, we need to formalize this intuition in terms of object values.

Update dimensions. Our first task is to determine which abstract state the update object is changing. That is, which dimensions of the target object need to be updated?

Consider the object value (without members) of an Open ResultSet in the Inserted, Scrollable, Updatable state, stored in val rs as depicted in Listing A.1. What should happen if we update rs to the ReadOnly state?

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

While there are clearly matches between tags in the target and update objects, since the tags are nested inside the Open tag of the target object, it is not clear that they should be updated. However, if we think of the state update as an transition to a new abstract state, then we can see that the nesting in the target object should not matter. This state update specifies that the Status dimension should transition to the ReadOnly substate, and thus out of the Updatable state.

The converse question is does nesting matter in the other direction? In other words, can a nested state trigger a change in an abstract state? Concretely, would this state update

$$rs \leftarrow \texttt{Foo}\{ReadOnly <: Status\}$$

result in an object in the ReadOnly state? Based on the semantics of state charts, the answer would be “no". Our definitions of object dimension indicates that the Status dimension of the Foo state is part of the definition of Foo. Thus, it is brought along with the transition to the Foo state. The Status state is also a defining and-state of the Open state. Thus the resulting object cannot be consistent because two separate dimensions are claiming the Status state meaning there would need to be duplicate tags.

Therefore, we define the dimensions along which a state update occurs to be only those found at the top level of the object value that describes the update object. All other dimensions that are a part of the update object are considered definitions of these dimensions and do not induce transitions but are only added to the object with their enclosing state.

Dimension updates. Once we know which dimensions will be updated, we need to know what in those dimensions is changed. We first note that we can treat the transition in each dimension independently as dimensions are orthogonal by definition. Second, recall the file example from Listing 2.1. In this example, we stated that the filename member was shared between the OpenFile and ClosedFile states. Thus, when we transition from an ClosedFile to an OpenFile the members of the File state should

remain constant. This is the semantics behind the restricted update semantics of state change described in [Aldrich et al., 2010]. We use and extend these semantics in a natural way to account for our hierarchical object model.

Properties of object and state update. With the intuition we have for the design, we can define some terminology that is used in the judgment itself.

Inner and outer tags. In the informal description of state change, we differentiated between dimensions and tags defined at the top level of the update object and those that appear within a top-level dimension. Figure A.7 defines two judgments, $\mathrm{outerTags}$ and $\mathrm{innerTags}$, which capture this distinction. The $\mathrm{outerTags}$ of an object value, ov, are all the tags which appear as the most specific tag and any of its super tags from dimensions appearing directly in ov. For example, using rs, the ResultSet object from Listing A.1, $\mathrm{outerTags}(rs) = \{Open, ResultSet\}$. Conversely, the $\mathrm{innerTags}$ of an object value are all of the tags defined in dimensions that are recursively included in the definition of each of the outer tags. For example,

$$\begin{aligned}\mathrm{innerTags}(rs) = \{&\texttt{Inserted}, \texttt{Inserting}, \texttt{Action},\\ &\texttt{Scrollable}, \texttt{Direction}, \texttt{Updatable}, \texttt{Status}\}\end{aligned}$$

Unique dimension property. Given a dimension within which to transition the target object, we need to find the location of the matching dimension within the target object value. To do this, we look for the part of the object that has tags which overlap the outer tags of the update dimension. We ignore all super-tags of the matching tag in the update dimension under the assumption that these supertags will match the tags in the target. This assumption is based on the the Unique Dimension Property which states that a single unique tag can only ever appear in a single dimension. That is, a tag either has no super tags or always appears with the same supertag. While this property is not guaranteed by the syntax and semantics of the internal language, it is enforced by the elaboration from Plaid's source syntax so we assume it in our rules.

Maintaining the `uniqueTags` property. Rule E-SU in Figure A.3 does not check whether the object returned from the state update judgement has unique tags. Therefore the state update judegment must maintain this property. Formally: If

$$\boxed{oe@H; T \mapsto oe@H; T}$$

$$\frac{[oe' = oe[v/f']]}{\mathsf{val}\ f \rhd f' = v[\curlyvee oe]@H \mapsto \mathsf{val}\ f = v[\curlyvee oe']@H}\ \text{E-RecField}$$

$$\frac{sd = \mathsf{freshtag}\{oe\}[<: de] \qquad tag \text{ is fresh} \qquad r = \mathsf{recstate}\{\overline{\mathsf{val}\ s_d \rhd x_d = v}, \mathsf{val}\ s \rhd x = \mathsf{proto}\ (tag\{oe\}[<: de]), \overline{\mathsf{val}\ s_r \rhd x_r = \mathsf{proto}\ sd_r}\}[\curlyvee oe]}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_d \rhd x_d = v}, \mathsf{val}\ s \rhd x = \mathsf{proto}\ sd, \overline{\mathsf{val}\ s_r \rhd x_r = \mathsf{proto}\ sd_r}\}[\curlyvee oe]@H \mapsto r@H'}\ \text{E-RecstateI}$$

$$\frac{\overline{oe'_i = oe_i[\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe_i}\}\#s_i/x_i]} \qquad [oe' = oe[\overline{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe}\}\#s_i/x_i}]]}{\mathsf{recstate}\{\overline{\mathsf{val}\ s_i \rhd x_i = \mathsf{proto}\ oe_i}\}[\curlyvee oe]@H \mapsto \overline{\mathsf{val}\ s_i = \mathsf{proto}\ oe'_i}[\curlyvee oe']@H}\ \text{E-RecstateII}$$

$$\frac{}{(tag\{oe\} <: \mathsf{proto}\ de[\{\overline{to}\}])[\curlyvee oe']@H \mapsto (tag\{oe\} <: de[\{\overline{to}\}])[\curlyvee oe']@H}\ \text{E-De}$$

$$\frac{}{\mathsf{proto}[ov\ \curlyvee](\mathsf{proto}\ oe[\{\overline{to}\}])[\curlyvee oe']@H \mapsto \mathsf{proto}[ov\ \curlyvee](oe[\{\overline{to}\}])[\curlyvee oe']@H}\ \text{E-Oe}$$

$$\frac{dv\{\overline{to}\} \Rightarrow dv'}{(dv\{\overline{to}\})[\curlyvee oe]@H \mapsto dv'[\curlyvee oe]@H}\ \text{E-TraitOps}$$

Figure A.5: Object Evaluation

$\texttt{uniqueTags}(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then $\texttt{uniqueTags}(ov_3)$. A proof of this property is in Appendix B.

Inference rules. With this understanding, we can describe the rules that produce the object value after a state update operation. The rules start by breaking apart the update object ov into the individual member values and dimension values and processing the state changes for each dimension or value individually (SU-LIST). This is allowed since each dimension can be treated independently. We can assume that `uniqueTags` holds for each dimension individually since it holds for the object as a whole. For member values (SU-MV) and dimension values for which there is no overlap between the tags of the target object and update dimension (SU-ADDH), we just compose the update object with the target object. The rest of the rules assume that there is a match between the outer tags of the update object and the tags of the target object. If that is not the case, then the evaluation gets stuck.

SU-MATCHDIM covers the case where we have found a particular dimension of the target object that contains the tags that are changing. By the unique dimension property explained above, we know that the $\mathrm{outerTags}(dv_u)$ will not appear in ov, so it suffices to calculate the state update on just the matched dimension. To ensure that we maintain the unique tags property, we can assume that both the result of the state update and the unmatched portion of the object have unique tags, and so it suffices to check that the tags of these two portions of the object do not intersect.

SU-MATCHINNER handles the case where there is overlap between the $\mathrm{innerTags}$ of the current tag and the $\mathrm{outerTags}$ of the update dimension. We recursively find the state update on just this matching portion and then check that the tags from the resulting object value do not intersect with the tags of the super tag, if it exists, to maintain the `uniqueTags` invariant.

In SU-MATCHSUPERINNER, we find that the matching dimension is defined somewhere inside of a super tag. Thus, we run state update on just the supertags. We then verify that the tags of the result are distinct from the tags of the subtag and its $\mathrm{innerTags}$ to maintain the `uniqueTags` invariant.

SU-MATCHSUPER represents the case where we have found the right dimension, but have not reached the level of the dimension where the tags overlap. The current tag of this dimension is not in the outer tags of the update dimension, but there is overlap somewhere in its super tags and so we find the updated state from that portion of the dimension. In this case, we know that the current tag will be removed with any of its nested tags, which means that we do not need to check if these tags would conflict with tags that enter the object with the update dimension to preserve `uniqueTags`.

The base case SU-MATCH handles the actual alteration of the target dimension. The current tag matches a specific tag in the outer tags of the update dimension, which indicates that the state update only affects states below this point in the dimension. In particular the tags below this one in the dimension in the target object are discarded, as already occurred through the SU-MATCHSUPER rule. In their place are put all the subtags of the matched tag from the incoming dimension. To make sure that we do not have duplicate tags anywhere, we only need to check that the tags added from the update dimension do not intersect with the tags that are in its new supertags.

Example. To give a specific example, consider evaluating the following state update on the object defined in Listing A.1:

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

The state updates proceeds first by finding that there is tag overlap between the incoming and target objects and a match for the Status tag of the incoming state nested inside the Open state with the SU-MATCHINNER. Next it finds the correct dimension `Updateable` $<:$ `Status` using the SU-MATCHDIM rule. It discards the $\mathrm{Updateable}$ tag and recurses up the dimension in the SU-MATCHSUPER rule and finally adds the $\mathrm{ReadOnly}$ tag in its place with the SU-MATCH rule.

Reduction rule The E-SU reduction rule uses the state update judgement to determine what object value to update the target object to. The state update judgement incrementally checked that `uniqueTags` was maintained. It does not guarantee that `uniqueMembers` is satisfied and so the rule checks that the resulting object value has unique member declarations.

A.3.4 Object Evaluation

The final class of reductions that we must model is that of state expressions, including the initialization of object expressions within a proto. These rules are defined in Figure A.5. Congruence rules are again taken care of by evaluation contexts from Figure A.2.

- E-RECFIELD: When field members have been evaluated down to values, we propagate them forward into the rest of the declarations that need to be initialized by substituting the value for the bound variable on the right of the $\triangleright$. This allows subsequent fields to use the values of previously declared fields during their initialization. After this propagation, we do not need to keep track of the bound variable any longer and so do not record it in the member value. Note that these semantics force us to be strict about the order in which portions of the object are initialized. In particular, member declarations are initialized from left to right as specified by the evaluation contexts.
- E-RECSTATE1: If there are freshtag directives in the state declarations of a recstate, new tags are generates by picking a fresh tag not previously mentioned.
- E-RECSTATE2: After assigning new tags to all of the state declarations inside a recstate, we need to remove the recstate construct and convert it into a list of val declarations. This is done in a manner similar to the fix construct in the lambda calculus. Since our recstate is modeled as a record, we replace all references to the inner bound variable of each of the nested state vals with selections of the external name from the recstate. We do this both inside the object expressions of each proto as well as in subsequent declarations. Note again that after propagation we can remove the bound variable from the val declaration.
- E-DE and E-OE: These rules state that it is possible to unwrap a proto that is nested inside another proto. This can occur when a proto is part of an object expression inside another proto (E-OE), or when a proto is in a dimension expression, which only appear in proto expressions (E-DE). In either case, if trait operations are associated with this proto, then they are retained. Execution

$$\boxed{ov\{\overline{to}\} \Rightarrow ov}$$

$$\frac{dv\{to\} \Rightarrow dv' \qquad dv'\{\overline{to}\} \Rightarrow dv''}{dv\{to, \overline{to}\} \Rightarrow dv''}\text{T-GENERAL} \qquad \frac{ov = ov'[\curlyvee mv'] \qquad [\texttt{name}(mv') = x = \texttt{name}(mv)]}{(tag\{ov\}[<: dv])\{mv\} \Rightarrow tag\{ov' \curlyvee mv\}[<: dv]}\text{T-MEMBER}$$

$$\frac{ov\{\backslash n\} \Rightarrow ov' \qquad [dv\{\backslash n\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-REMOVEDV} \qquad \frac{dv\{\backslash n\} \Rightarrow dv' \qquad ov\{\backslash n\} \Rightarrow ov'}{(dv \curlyvee ov)\{\backslash n\} \Rightarrow dv' \curlyvee ov'}\text{T-REMOVEOV1}$$

$$\frac{\texttt{name}(mv) = n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlyvee ov)\{\backslash n\} \Rightarrow ov'}\text{T-REMOVEOV2} \qquad \frac{\texttt{name}(mv) \neq n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlyvee ov)\{\backslash n\} \Rightarrow mv \curlyvee ov'}\text{T-REMOVEOV3}$$

$$\frac{ov\{n \to n'\} \Rightarrow ov' \qquad [dv\{n \to n'\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-RENAMEDV}$$

$$\frac{dv\{n \to n'\} \Rightarrow dv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(dv \curlyvee ov)\{n \to n'\} \Rightarrow dv' \curlyvee ov'}\text{T-RENAMEOV1}$$

$$\frac{\texttt{name}(mv) = n \qquad \texttt{rename}(n', mv) = mv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlyvee ov)\{n \to n'\} \Rightarrow mv' \curlyvee ov'}\text{T-RENAMEOV2}$$

$$\frac{\texttt{name}(mv) \neq n \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlyvee ov)\{n \to n'\} \Rightarrow mv \curlyvee ov'}\text{T-RENAMEOV3}$$

$$\frac{de = tag\{oe\}[<: de'] \qquad ov\{tag.mv\} \Rightarrow ov'}{(ov)\{(\mathsf{tagOf\ proto}\ de).mv\} \Rightarrow ov'}\text{T-STATEMEMBER}$$

$$\frac{tag \notin \text{tags}(dv) \qquad ov\{tag.mv\} \Rightarrow ov'}{(dv \curlyvee ov)\{tag.mv\} \Rightarrow dv \curlyvee ov'}\text{T-STATEMEMBEROV1}$$

$$\frac{tag \in \text{tags}(dv) \qquad dv\{tag.mv\} \Rightarrow dv'}{(dv[\curlyvee ov])\{tag.mv\} \Rightarrow dv'[\curlyvee ov]}\text{T-STATEMEMBEROV2}$$

$$\frac{tag \neq tag' \qquad ov\{tag'.mv\} \Rightarrow ov' \qquad [dv\{tag.mv\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{tag'.mv\} \Rightarrow tag\{ov'\}[<: dv']}\text{T-STATEMEMBERDV1}$$

$$\frac{(tag\{ov\}[<: dv])\{mv\} \Rightarrow dv'}{(tag\{ov\}[<: dv])\{tag.mv\} \Rightarrow dv'}\text{T-STATEMEMBERDV2}$$

Figure A.6: Trait Operations

will continue by evaluating the wrapped object expression if needed.

- E-TRAITOPS: This rule applies only once the all of the trait operations have been fully reduced and proceeds using the trait operations judgment defined below to produce a new dimension value.

A.3.5 Trait Operations

As with state change, we define a separate judgement for trait operations that applies once all trait operations have been fully initialized, meaning that they can all be applied atomically without reduction. The rules for initialization of trait operations are all congruence rules handled by evaluation contexts (see Figure A.2). Thus, the judgement,

$\texttt{uniqueTags}(ov)$ $\quad$ $\texttt{uniqueMembers}(ov)$ $\quad$ $\texttt{lookup}(x, ov) = mv$ $\quad$ $dv \in ov$ $\quad$ $mv :: tag.x@ov$
$tag <<: tag@ov$ $\quad$ $\texttt{validTagMembers}(ov)$ $\quad$ $\texttt{rename}(n, mv) = mv$ $\quad$ $\texttt{name}(mv) = n$
$\text{tags}(ov)$ $\quad$ $\text{outerTags}(ov)$ $\quad$ $\text{innerTags}(ov)$

$$\frac{tag \notin \text{tags}(ov)\ [\ \cup \text{tags}(dv)] \qquad [\text{tags}(ov) \cap \text{tags}(dv) = \emptyset] \qquad \texttt{uniqueTags}(ov) \qquad [\texttt{uniqueTags}(dv)]}{\texttt{uniqueTags}(tag\{ov\}[<: dv])}\ \textsc{UniqueTagsDv}$$

$$\frac{\text{tags}(dv) \cap \text{tags}(ov) = \emptyset \qquad \texttt{uniqueTags}(dv) \qquad \texttt{uniqueTags}(ov)}{\texttt{uniqueTags}(dv \curlyvee ov)}\ \textsc{UniqueTagsOv1}$$

$$\frac{[\texttt{uniqueTags}(ov)]}{\texttt{uniqueTags}(mv[\curlyvee ov])}\ \textsc{UniqueTagsOv2} \qquad \frac{mv_1 :: tag_1.x@ov \ldots mv_n :: tag_n.x@ov \qquad tag_i <<: tag_1@ov \ldots tag_i <<: tag_n@ov}{\texttt{lookup}(x, ov) = mv_i}\ \textsc{Lookup}$$

$$\frac{\texttt{validTagMembers}(ov) \qquad \exists n(\exists tag\ mv :: tag.n@ov \wedge \exists tag'\ mv' :: tag'.n@ov) \Longrightarrow (tag <<: tag'@ov \vee tag' <<: tag@ov)}{\texttt{uniqueMembers}(ov)}\ \textsc{UniqueMembers}$$

$$\frac{}{dv \in dv}\ \textsc{Leaf1} \qquad \frac{dv \in ov'}{dv \in mv \curlyvee ov'}\ \textsc{Leaf2} \qquad \frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in dv'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\curlyvee ov'']}\ \textsc{Leaf3}$$

$$\frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in ov'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\curlyvee ov'']}\ \textsc{Leaf4} \qquad \frac{tag\{[ov_1 \curlyvee]mv[\curlyvee ov_1]\} <: dv \in ov}{mv :: tag@ov}\ \textsc{MbrInTag}$$

$$\frac{tag\{ov'\} <: dv \in ov \qquad tag' \in \text{outerTags}(tag\{ov'\} <: dv)}{tag <<: tag'@ov}\ \textsc{CaseOf} \qquad \frac{name(mv) \notin names \qquad [\texttt{validTagMembers}(names \cup name(mv), ov')]}{\texttt{validTagMembers}(names, mv[\curlyvee ov'])}\ \text{VTM1}$$

$$\frac{\texttt{validTagMembers}(\emptyset, ov) \qquad [\texttt{validTagMembers}(\emptyset, dv)] \qquad [\texttt{validTagMembers}(names, ov')]}{\texttt{validTagMembers}(names, (tag\{ov\}[<: dv])[\curlyvee ov'])}\ \text{VTM2}$$

$$\frac{}{n = \texttt{name}(\mathsf{val}\ n = v)}\ \textsc{Name1} \qquad \frac{}{m = \texttt{name}(\mathsf{method}\ m(\overline{x})\{e\})}\ \textsc{Name2}$$

$$\frac{}{\texttt{rename}(a, \mathsf{val}\ n = v) = \mathsf{val}\ a = v}\ \textsc{Rename1} \qquad \frac{}{\texttt{rename}(n, \mathsf{method}\ m(\overline{x})\{e\}) = \mathsf{method}\ n(\overline{x})\{e\}}\ \textsc{Rename2}$$

$$\frac{}{\text{tags}(ov) = \text{innerTags}(ov) \cup \text{outerTags}(ov)}\ \textsc{Tags}$$

$$\frac{}{\text{outerTags}(tag\{ov\}[<: dv]) = \{tag\}\ [\cup \text{outerTags}(dv)]}\ \textsc{OuterDv}$$

$$\frac{}{\text{outerTags}(dv[\curlyvee ov]) = \text{outerTags}(dv)\ [\cup \text{outerTags}(ov)]}\ \textsc{OuterOv1}$$

$$\frac{}{\text{outerTags}(mv[\curlyvee ov]) = \emptyset\ [\cup \text{outerTags}(ov)]}\ \textsc{OuterOv2}$$

$$\frac{}{\text{innerTags}(tag\{ov\}[<: dv]) = \text{tags}(ov)\ [\cup \text{innerTags}(dv)]}\ \textsc{InnerDv}$$

$$\frac{}{\text{innerTags}(dv[\curlyvee ov]) = \text{innerTags}(dv)\ [\cup \text{innerTags}(ov)]}\ \textsc{InnerOv1}$$

$$\frac{}{\text{innerTags}(mv[\curlyvee ov]) = \emptyset\ [\cup \text{innerTags}(ov)]}\ \textsc{InnerOv2}$$

Figure A.7: Helper Judgements

$ov\{\overline{sp}\} \Rightarrow ov$, does not require a heap. In general, trait operations follows previous work on traits. However, Plaid's object model, unlike traditional traits models, is hierarchical. Hence, trait operations other than the local member addition must take this hierarchy into account.

Local member updates are agnostic to whether the added member is already a member of the tag and simply add the new member, replacing the existing member if one exists (T-MEMBER). Updates of members in specific tags act the same, but first must recurse through the object value looking for the specified tag before performing the member update. The computation will get stuck if the tag is not found. Because each of these trait operations, as well as member renaming described below, may potentially add new members, there is the danger that the object value might no longer satisfy the `uniqueMembers` invariant. However, since the specialization must be occurring as part of object instantiation, it will be checked at the point that the object is created, so we do not make the check here.

Member removal and renaming operate on the whole object, removing or renaming instances of members with the given name throughout. This is in contrast to `lookup`, which stops at the first declaration of the member. These semantics are required in order to allow trait composition, which includes the ability to remove members from a trait and instead provide them in another trait. This would result in a conflict if some members were left in the old dimension.

A.4 Elaboration

The core language defined in the previous section shares much in common with the full Plaid programming language, but there are still differences. The source syntax is defined in figure A.8. The semantics of the full Plaid language are defined as an elaboration into the core language defined in Section A.5.

For most expressions, the elaboration proceeds structurally, without changing the construct itself. For field bindings, we add the internal variable referred to above, and replace references to the field in later field initializers with the fresh variable. Sequences of state declarations are transformed into recstate blocks. Each state declaration is transformed into a val declaration which binds to a proto representing the uninitialized state, with a freshtag expression for generating the state's tag when the declaration is executed.

Our formal semantics defines all of the Plaid language except for module linking and

Declarations	D	$::=$	$SD \mid \textsf{method}\ m(\overline{x})\{SE\} \mid$
			$\textsf{val}\ f = SE$
State Decl.	SD	$::=$	$\textsf{val}\ s = S \mid \textsf{state}\ s = S \mid$
			$\textsf{state}\ s\ \textsf{case of}\ s\{\overline{TO}\} = S$
States	S	$::=$	$\textsf{freeze}(SE) \mid \{\overline{D}\} \mid s\{\overline{T}\} \mid$
			$S\ \textsf{with}\ S \mid SE.s \mid s$
Trait Ops	TO	$::=$	$\backslash n \mid n \rightarrow n' \mid$
			$\textsf{val}\ f = SE \mid \textsf{val}\ s = S \mid$
			$\textsf{val}\ s.f = SE \mid \textsf{val}\ s.t = S \mid$
			$\textsf{method}\ m(\overline{x})\{SE\} \mid$
			$\textsf{method}\ s.m(\overline{x})\{SE\}$
Expression	SE	$::=$	$x \mid \textsf{let}\ x = SE\ \textsf{in}\ SE \mid SE.f \mid$
			$SE(\overline{SE}) \mid SE.m(\overline{SE}) \mid$
			$SE \leftarrow S \mid SE \twoheadleftarrow S \mid \textsf{new}\ S \mid$
			$\textsf{match}(SE)\{\overline{C}\} \mid$
Case	C	$:=$	$\textsf{case}\ SE.s\ \{SE\} \mid$
			$\textsf{case}\ s\ \{SE\} \mid \textsf{default}\ \{SE\}$
Compil. Unit	CU	$::=$	$\overline{D}$

Figure A.8: Source Syntax

cross language binding. Module linking currently follows the Java standard, including packages, imports, and a classpath for loading elements. Plaid primitives are defined using Java classes and methods, which can be directly accessed in Plaid via their fully-qualified Java names. Details of both of these aspects of Plaid are discussed in more detail in the Plaid language definition [Aldrich et al., 2012].

A.5 Source Translation Rules

The rules in figures A.9 and A.10 below describe how to translate a program written in the Plaid source language given in figure A.8 to the internal language defined in figure A.1.

$$\boxed{CU \rightsquigarrow e \qquad \overline{D} \rightsquigarrow oe \qquad SD \rightsquigarrow \mathsf{val}\ s \rhd s' = se \qquad S \rightsquigarrow oe \qquad \overline{TO} \rightsquigarrow \overline{to} \qquad \texttt{groupStates}(\overline{D})}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{CU \rightsquigarrow \mathsf{let}\ top = \mathsf{new}\ (\mathsf{proto}\ oe)\ \mathsf{in}\ top.main()}\ \text{TR-CU}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE_f \rightsquigarrow e_f \qquad f' = freshname \qquad oe'_d = oe_d[f'/f]}{\mathsf{val}\ f = SE_f, \overline{D} \rightsquigarrow \mathsf{val}\ f \rhd f' = e_f \curlyvee oe'_d}\ \text{TR-DeclField}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE \rightsquigarrow e}{\mathsf{method}\ m(\overline{x})\{SE\}, \overline{D} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\} \curlyvee oe_d}\ \text{TR-DeclMethod}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad \overline{SD \rightsquigarrow \mathsf{val}\ s \rhd s' = sd}}{\{\overline{SD}\}, \overline{D} \rightsquigarrow (\mathsf{recstate}\{\overline{\mathsf{val}\ s \rhd s' = \mathsf{proto}\ sd}\} \curlyvee oe_d)[\overline{s'/s}]}\ \text{TR-DeclStates}$$

$$\frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto}\ \mathsf{freshtag}\{oe\})}\ \text{TR-StateTag} \qquad \frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{val}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = oe}\ \text{TR-StateVal}$$

$$\frac{s_s\{\overline{TO}\} \rightsquigarrow de \qquad S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s\ \mathsf{case\ of}\ s_s\{\overline{TO}\} = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto}\ \mathsf{freshtag}\{oe\} <: de)}\ \text{TR-StateCase}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{\{\overline{D}\} \rightsquigarrow oe}\ \text{TR-StateDecl} \qquad \frac{S_1 \rightsquigarrow oe_1 \qquad S_2 \rightsquigarrow oe_2}{S_1\ \mathsf{with}\ S_2 \rightsquigarrow oe_1 \curlyvee oe_2}\ \text{TR-StateWith}$$

$$\frac{SE \rightsquigarrow e}{\mathsf{freeze}(SE) \rightsquigarrow \mathsf{freeze}(e')}\ \text{TR-StateFreeze} \qquad \frac{SE \rightsquigarrow e}{SE.s \rightsquigarrow e.s}\ \text{TR-StateSelect} \qquad \frac{}{s \rightsquigarrow s}\ \text{TR-StateName}$$

$$\frac{\overline{TO} \rightsquigarrow \{\overline{to}\}}{s\{\overline{TO}\} \rightsquigarrow s\{\overline{to}\}}\ \text{TR-StateInit} \qquad \frac{TO \rightsquigarrow to \qquad \overline{TO} \rightsquigarrow \overline{to}}{TO, \overline{TO} \rightsquigarrow to, \overline{to}}\ \text{TR-SpecGeneral}$$

$$\frac{\mathsf{val}\ f = SE \rightsquigarrow \mathsf{val}\ f \rhd f' = e}{\mathsf{val}\ [s.]f = SE \rightsquigarrow [s.](\mathsf{val}\ f = e)}\ \text{TR-SpecField}$$

$$\frac{\mathsf{method}\ m(\overline{x})\{E\} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\}}{\mathsf{method}\ [s.]m(\overline{x})\{E\} \rightsquigarrow [s.](\mathsf{method}\ m(\overline{x})\{e\})}\ \text{TR-SpecMethod}$$

$$\frac{S \rightsquigarrow oe}{\mathsf{val}\ [s.]t = S \rightsquigarrow [s.](\mathsf{val}\ t = oe)}\ \text{TR-SpecState} \qquad \frac{}{\backslash n \rightsquigarrow \backslash n}\ \text{TR-SpecRemove}$$

$$\frac{}{n \rightarrow n' \rightsquigarrow n \rightarrow n'}\ \text{TR-SpecRename}$$

$$\frac{\texttt{groupStates}(SD', \overline{D}) = \{\overline{sd_i = SD_i}\}, \overline{D'} \qquad s = \texttt{name}(SD)}{\texttt{groupStates}(SD, SD', \overline{D}) = \{s = SD, \overline{sd_i = SD_i}\}, \overline{D'}}\ \text{TR-GStAdd}$$

$$\frac{s = \texttt{name}(SD) \qquad \overline{D} = \cdot \vee (\overline{D} = (D, \overline{D'}) \wedge D \neq SD)}{\texttt{groupStates}(SD, \overline{D}) = \{s = SD\}, \texttt{groupStates}(\overline{D})}\ \text{TR-GStStart}$$

$$\frac{D \neq SD}{\texttt{groupStates}(D, \overline{D}) = D, \texttt{groupStates}(\overline{D})}\ \text{TR-GMember}$$

Figure A.9: Translate Declarations

$\boxed{SE \rightsquigarrow e \qquad C \rightsquigarrow c}$

$$\frac{S \rightsquigarrow oe}{\textsf{new}\ S \rightsquigarrow \textsf{new}\ oe}\ \text{TR-NEW} \qquad \frac{}{x \rightsquigarrow x}\ \text{TR-VAR} \qquad \frac{SE_f \rightsquigarrow e_f \qquad \overline{SE_a \rightsquigarrow e_a}}{SE_f(\overline{SE_a}) \rightsquigarrow e_f(\overline{e_a})}\ \text{TR-APP}$$

$$\frac{SE_r \rightsquigarrow e_r \qquad \overline{SE_a \rightsquigarrow e_a}}{SE_r.m(\overline{SE_a}) \rightsquigarrow e_r.m(\overline{e_a})}\ \text{TR-CALL} \qquad \frac{SE \rightsquigarrow e}{SE.f \rightsquigarrow e.f}\ \text{TR-FIELD} \qquad \frac{SE \rightsquigarrow e \qquad S \rightsquigarrow oe}{SE \leftarrow S \rightsquigarrow e \leftarrow oe}\ \text{TR-SU}$$

$$\frac{SE \rightsquigarrow e \qquad S \rightsquigarrow oe}{SE \twoheadleftarrow S \rightsquigarrow e \twoheadleftarrow oe}\ \text{TR-REPLACE} \qquad \frac{SE_x \rightsquigarrow e_x \qquad SE_b \rightsquigarrow e_b}{\textsf{let}\ x = SE_x\ \textsf{in}\ SE_b \rightsquigarrow \textsf{let}\ x = e_x\ \textsf{in}\ e_b}\ \text{TR-LET}$$

$$\frac{SE \rightsquigarrow e \qquad \overline{C \rightsquigarrow c}}{\textsf{match}(SE)\{\overline{C}\} \rightsquigarrow \textsf{match}(e)\{\overline{c}\}}\ \text{TR-MATCH} \qquad \frac{SE \rightsquigarrow e}{\textsf{case}\ s\ \{SE\} \rightsquigarrow \textsf{case}\ s\ \{e\}}\ \text{TR-CASE1}$$

$$\frac{SE_c \rightsquigarrow e_c \qquad SE \rightsquigarrow e}{\textsf{case}\ SE_c.s\ \{SE\} \rightsquigarrow \textsf{case}\ e_c.s\ \{e\}}\ \text{TR-CASE2} \qquad \frac{SE \rightsquigarrow e}{\textsf{default}\ \{SE\} \rightsquigarrow \textsf{default}\ \{e\}}\ \text{TR-DEFAULT}$$

Figure A.10: Translate Expressions

Appendix B

Unique members proof

Theorem 1: If $\texttt{uniqueTags}(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then $\texttt{uniqueTags}(ov_3)$.

Proof: By induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$.

Case SU-List:

$\texttt{uniqueTags}(ov_o)$ by the induction hypothesis.

Case SU-MV:

$\texttt{uniqueTags}(ov_t)$ by assumption.

$\texttt{uniqueTags}(ov_t \curlyvee mv_u)$ by rule UniqueTagsOV2.

Case SU-AddH:

$\texttt{uniqueTags}(ov_t)$ by assumption.

$\texttt{uniqueTags}(ov_t \curlyvee mv_u)$ by rule UniqueTagsOV1.

Case SU-MatchDim:

$\texttt{uniqueTags}(ov_t)$ by assumption.

$\texttt{uniqueTags}(dv_r)$ by the induction hypothesis.

$\texttt{uniqueTags}(ov_t \curlyvee dv_r)$ by rule UniqueTagsOV1.

Case SU-MatchInner:

$\texttt{uniqueTags}(tag\{ov\} <: dv)$ by assumption.

$\texttt{uniqueTags}(ov_r)$ by the induction hypothesis.

$tag \notin \text{tags}(dv) \wedge tag \notin \text{tags}(ov) \wedge \texttt{uniqueTags}(dv) \wedge$
$\text{tags}(ov) \cap \text{tags}(dv) = \emptyset$ by inversion of
$\texttt{uniqueTags}(tag\{ov\} <: dv)$.

$tag \notin \text{tags}(ov_r)$ by Lemma 3.

$\text{tags}(ov_r) \cap \text{tags}(dv) = \emptyset$ by Lemma 2.

$\texttt{uniqueTags}(tag\{ov_r\} <: dv)$ by rule UniqueTagsDv.

Case SU-MatchSuperInner:

$tag \notin \text{tags}(dv_r)$ by Lemma 3.

$\text{tags}(ov) \cap \text{tags}(dv_r) = \emptyset$ by Lemma 2.

$\texttt{uniqueTags}(tag\{ov\} <: dv_r)$ by UniqueTagsDv.

Case SU-MatchSuper:

$\texttt{uniqueTags}(dv_r)$ by the induction hypothesis.

Case SU-Match:

$\texttt{uniqueTags}(dv_{sub} <: tag\{ov\} <: dv)$ by rule UniqueTagsDv.

□

Lemma 1: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3$ then
$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$

Proof: By easy induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$

□

Lemma 2: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge$
$\text{tags}(ov) \cap \text{tags}(ov_1) = \emptyset \wedge \text{tags}(ov) \cap \text{tags}(ov_2) = \emptyset$ then $\text{tags}(ov) \cap \text{tags}(ov_3) = \emptyset$

Proof:

$\text{tags}(ov) \cap (\text{tags}(ov_1) \cup \text{tags}(ov_2)) = \emptyset$

$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$ by Lemma 1.

$\text{tags}(ov) \cap \text{tags}(ov_3) = \emptyset$

□

Lemma 3: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge tag \notin \text{tags}(ov_1) \wedge$
$tag \notin \text{tags}(ov_2)$ then $tag \notin \text{tags}(ov_3)$

Proof:

$tag \notin (\text{tags}(ov_1) \cup \text{tags}(ov_2)) = \emptyset$

$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$ by Lemma 1.

$tag \notin \text{tags}(ov_3)$

□

Theorem 2: If we $e@H \mapsto e'@H'$ and $\forall \ell \in H.\texttt{uniqueTags}(H[\ell])$, then $\forall \ell \in H'.\texttt{uniqueTags}(H'[\ell])$

Proof: By induction on $e@H \mapsto e'@H'$

Case E-New and E-Replace:

$\forall \ell' \in H[\ell \rightsquigarrow ov].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis and rule premise.

Case E-Su:

$\texttt{uniqueTags}(ov_3)$ by Theorem 1.

$\forall \ell' \in H[\ell \rightsquigarrow ov_3].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis.

All Other Rules:

Heap does not change.

Appendix C

Plaiddoc experiment study materials

This appendix contains the study materials for the Plaiddoc experiment discussed in chapter 4. It contains:

- The experimental protocol.
- The screening survey.
- The training script.
- The Javadoc documentation for the Car API used in training.
- The Plaiddoc documentation for the Car API used in training.
- The state glossary given to all participants during the study.
- The post-experiment interview script.

Chapter 4 contains several other pieces of the study materials:

- The UML state machine for the CAR API used in training is in Figure 4.2.
- The tasks themselves are in Table 4.1.

1. Screen participant with the screening survey.
2. If the participant is eligible for the study, assign the participant to one of the four conditions.
3. Seat participants at a table and have them read and sign the consent form.
4. Hand participants Car API documentation in three forms (state diagram, Javadoc, Plaiddoc).
5. Read the training script aloud and answer any questions asked.
6. Ask the participant, "Do you have any more questions or are you ready to begin the move on?"
7. Reseat participant in front of the computer.
8. Instruct participant, "We now begin the main part of the study. I will ask you a series of questions about three APIs and you will answer them using the [Java/Plaid]doc documentation opened for you in a browser window. I will also hand you a glossary of states for each API. Find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it. Some questions will require you to name a state or states. Your answer will always be one of the states in the glossary. After you answer each question I will ask if it is your final answer. If it is, we will move on to the next question.
9. Ask the participant, "Are you ready to begin?"
10. Start recording with Camtasia.
11. Open the Timer and TimerTask or URLConnection documentation. Hand the participant the related glossary
12. Ask the first batch of questions.
13. Close the first batch of documentation and take the first glossary.
14. Open the ResultSet documentation and hand the participant the ResultSet glossary.
15. Ask the second batch of questions.
16. Close the ResultSet documentation and take the ResultSet glossary.
17. Open the Timer and TimerTask or URLConnection documentation and the third glossary.
18. Ask the third batch of questions.
19. Close the third batch of documentation and take the third glossary.
20. Conduct the post experiment interview.
21. Thank, pay, and escort the participant from the lab.
22. Stop Camtasia recording.

Are you a student?

What kind of student, undergraduate, masters or a PhD student?

Which program are you in?

Have you ever programmed in Java or C#?

Have you used Java or C# API documentation?

Describe your professional programing experience. Please include summer internships.

Criteria: Accepted any student that had at least one summer of professional programming experience and had used Java or C# and their associated API documentation.

Participant Number: _____

Condition: _____

We'll begin with a brief training session today. Feel free to interrupt with any questions you have during this training.

In the study today, you'll be using APIs with protocols. APIs with protocols have a finite number of states and in each state a different subset of method calls are valid. Protocols also specify transitions between states that occur as part of some method calls.

To make this concrete, let's look at a fictional Car class, which is a simple programmatic representation of a real-world car. Look at the Car class' state machine diagram. You can see that Car state machine is divided into three sections by a dashed line.

First, look inside the bottom left section. You should see a rounded rectangle with the label "Brakes" written at the top. This is the "Brakes" state. In this state machine, all states are written as rounded rectangles.

The "Brakes" state has two children "Braking" and "Not Braking." A car can be either braking or not braking, but not both simultaneously. We call states like this that cannot be simultaneously active "or-states." We also say that the "Braking" and "Not Braking" states are "or-children" of the "Brakes" state.

You can transition a car object from the "NotBraking" to the "Braking" state by calling its "putFootDown" method. This is indicated in the diagram by the arrow that starts at "NotBraking" and ends at "Braking" and is labeled "putFootDown."

> *Question 1: How do you transition a car object from the "Braking" to the "NotBraking" state? (Answer: "liftFoot")*

Moving to the right section, you should see the "Gear" state. You can transition the gear to "First" gear only from "Neutral."

Question 2: What state does the car have to be in to call the "toFourth" method? (Answer: Neutral)

A car is in both a "Gear" state and a "Braking" state at the same time. We call states like this that are simultaneously active "and-states." We also say that the "Brakes" and "Gears" states are "and-children" of the top-level Car state. Changing the gear of the car has no impact on the state of the brakes and vice versa. For example, if the car is in the "Braking" state and the "toSecond" method is called the car will still be in the "Braking" state after the method returns. All "and-states" act independently in this way.

Question 3: If the car is in the "Braking" state and the "Second" gear and the "liftFoot" method is called and returns successfully. What states is the car in now? (Answer: "NotBraking" and "Second")

Methods that are available in a state are available for all of that state's children. For example, the "toNeutral" method which transitions from the "Gear" state to the "Neutral" state is available from any Gear.

Question 4: What methods can I call in the "Fourth" gear? (Answer: "toNeutral")

Let's now take a look at the Javadoc documentation for the Car API. Like all Javadoc, this page has a "Method Summary" table which is an alphabetized list of the methods available on a car object. It shows the return type and a short description of each method. Further along in the page you should see "Method Detail" section which lists the same methods but provides more detail. Look at the "toSecond" method in this section. Notice that it mentions in the description that it changes the gear "from neutral." Also note the fact that this method throws a "java.lang.IllegalStateException" if the car is not in neutral. In this case, both the method description and the exception description document the protocol. The last method in the class, "openMethod" only works if the "car is in neutral gear." This method does not throw an exception, instead it returns false if the method is not in the neutral state. Therefore, the return value description documents the protocol.

Question 5: What state does the car need to be in to call the liftFoot method? (Answer: Braking)

Question 6: What happens when you call the "liftFoot" method if the NotBraking state? (Answer: IllegalStateException)

Finally, let's look at the Plaiddoc documentation for the Car API. This document looks mostly like Javadoc, but the method summary tables are divided by the state in which the methods are available. For example, the "liftFoot" method appears inside the a section marked "Braking." At the top of each state table is a state diagram which describes the relationships between states. It shows that "Brakes" and "Gear" are "and-children" of the top-level "Car" state and that "Braking" and "NotBraking" are "or-children" of "Brakes."

All of the method summaries in Plaiddoc include two columns, precondition and postcondition, that are not in Javadoc. The postcondition of the "liftFoot" method is "NotBraking," which indicates that it transitions the car to the "NotBraking" state.

Question 7: What is the postcondition of the "toSecond" method? (Answer: Second)

Most of the preconditions listed in the Car API are "null," but that doesn't mean you can call those methods in any state. Instead, the state in which the method appears is an implicit precondition. For example, the "liftFoot" method can only be called in the "Braking" state since it is in the braking section of the Plaiddoc. However, some methods have additional prectonditions. The "toFifth" method, which is appears in the Turbo section also lists Neutral as its precondition. Therefore, the car must be in bot the "Turbo" and "Neutral" states to call the "toFifth" method.

Question 8: What state must an object with be in to call the "foo" method be in, if the "foo" method is listed in the "Bar" section and has the precondition "Baz"? (Answer: Bar and Baz)

Package Class Use Tree Deprecated Index Help

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

sunshine.josh.thesis.training

Class Car

java.lang.Object
sunshine.josh.thesis.training.Car

```
public class Car
extends java.lang.Object
```

A programmatic representation of a real-world car.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	Car.Option

Constructor Summary

Constructors

Constructor and Description
Car(Car.Option opt)

Method Summary

Methods

Modifier and Type	Method and Description
void	liftFoot() Deactivates the brakes.
boolean	openTrunk() Opens the car's trunk, if the car is in the neutral gear.
void	putFootDown() Activates the brakes.
void	toFifth() Changes the gear to fifth gear from neutral, if the car has the turbo option.
void	toFirst() Changes the gear to first gear from neutral.

void	**toFourth**() Changes the gear to fourth gear from neutral.
void	**toNeutral**() Changes the gear to neutral from any gear.
void	**toSecond**() Changes the gear to second gear from neutral.
void	**toThird**() Changes the gear to third gear from neutral.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Car

```
public Car(Car.Option opt)
```

Method Detail

liftFoot

```
public void liftFoot()
```

Deactivates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already deactivated.

putFootDown

```
public void putFootDown()
```

Activates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already active.

toNeutral

```
public void toNeutral()
```

Changes the gear to neutral from any gear.

toFirst

```
public void toFirst()
```

Changes the gear to first gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toSecond

```
public void toSecond()
```

Changes the gear to second gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toThird

```
public void toThird()
```

Changes the gear to third gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFourth

```
public void toFourth()
```

Changes the gear to fourth gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFifth

```
public void toFifth()
```

Changes the gear to fifth gear from neutral, if the car has the turbo option.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

`java.lang.UnsupportedOperationException` - if the car does not have the turbo option.

openTrunk

```
public boolean openTrunk()
```

Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.

Returns:

true if the trunk is in the neutral gear and successfully opens; false otherwise.

Package Class Use Tree Deprecated Index Help

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Package | Class | Use | Tree | Deprecated | Index | Help

Prev Class | **Next Class** | Frames | No Frames | All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

sunshine.josh.thesis.training

Class Car

java.lang.Object
  sunshine.josh.thesis.training.Car

```
public class Car
extends java.lang.Object
```

A programmatic representation of a real-world car.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**Car.Option**

Constructor Summary

Constructors

Constructor and Description
Car(**Car.Option** opt)

Car

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|             |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```

Option

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Standard

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Turbo

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
void	Neutral	Fifth	**toFifth**() Changes the gear to fifth gear from neutral.

Brakes

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Braking

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
void	null	NotBraking	**liftFoot()** Deactivates the brakes.

NotBraking

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
void	null	Braking	**putFootDown**() Activates the brakes.

Gear

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
void	null	Neutral	**toNeutral**() Changes the gear to neutral from any gear.

Neutral

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
boolean	null	null	**openTrunk**() Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.
void	null	First	**toFirst**() Changes the gear to first gear from neutral.
			toFourth()

void	null	Fourth	Changes the gear to second gear from neutral.
void	null	Second	**toSecond**() Changes the gear to second gear from neutral.
void	null	Third	**toThird**() Changes the gear to second gear from neutral.

First

State relationships

```
         |¯Option---OR-|¯Standard
         |              |_Turbo
         | Brakes---OR-|¯Braking
         |              |_NotBraking
Car--AND-|              |¯Neutral
         |              | First
         |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```

Second

State relationships

```
         |¯Option---OR-|¯Standard
         |              |_Turbo
         | Brakes---OR-|¯Braking
         |              |_NotBraking
Car--AND-|              |¯Neutral
         |              | First
         |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```

Third

State relationships

```
         |¯Option---OR-|¯Standard
         |              |_Turbo
         | Brakes---OR-|¯Braking
         |              |_NotBraking
Car--AND-|              |¯Neutral
         |              | First
         |_Gear-----OR-| Second
                        | Third
```

```
                        | Fourth
                        |_Fifth
```

Fourth

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|              |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | ***Fourth***
                        |_Fifth
```

Fifth

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|              |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_***Fifth***
```

Constructor Detail

Car

```
public Car(Car.Option opt)
```

Method Detail

liftFoot

```
public void liftFoot()

State: Braking
Postconditions: NotBraking
```

Deactivates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already deactivated.

putFootDown

```
public void putFootDown()

State: NotBraking
Postconditions: Braking
```

Activates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already active.

toNeutral

```
public void toNeutral()

State: Gear
Postconditions: Neutral
```

Changes the gear to neutral from any gear.

toFirst

```
public void toFirst()

State: Neutral
Postconditions: First
```

Changes the gear to first gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toSecond

```
public void toSecond()

State: Neutral
Postconditions: Second
```

Changes the gear to second gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toThird

```
public void toThird()
```

```
State: Neutral
Postconditions: Third
```

Changes the gear to third gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFourth

```
public void toFourth()
```

```
State: Neutral
Postconditions: Fourth
```

Changes the gear to fourth gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFifth

```
public void toFifth()
```

```
State: Turbo
Preconditions: Neutral
Postconditions: Fifth
```

Changes the gear to fifth gear from neutral, if the car has the turbo option.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

`java.lang.UnsupportedOperationException` - if the car does not have the turbo option.

openTrunk

```
public boolean openTrunk()
```

```
State: Neutral
```

Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.

Returns:

true if the trunk is in the neutral gear and successfully opens; false otherwise.

Package Class Use Tree Deprecated Index Help

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Timer states:
- **Virgin**: A new timer.
- **Canceled**: A timer that has been cancelled.

TimerTasks states:
- **Virgin**: A new TimerTask.
- **Scheduled**: A TimerTask that has been scheduled by a Timer.
- **Executed**: A TimerTask whose action has been performed.
- **Canceled**: A TimerTask that has been canceled.

ResultSet states:
- **Closed**: A result set whose table of data is no longer available.
- **ReadOnly**: A result set that is not updatable.
- **Updatable**: A result set that is updatable.
- **Scrollable**: A result set whose cursor can move either forward or backward.
- **ForwardOnly**: A result set whose cursor can only move forward.
- **InvalidRow**: A result set whose cursor is on an invalid row.
- **Inserting**: A result set for which a new row is being created.
- **Inserted**: A result set in which a full new row has been inserted.
- **NotYetRead**: A result set whose cursor is on a row of data that is unread.
- **Read**: A result set whose cursor is on a row of data that has been read.

UrlConnection states:
- **Disconnected**: A URLConnection that has not connected.
- **Connected**: A URLConnection whose connection has been established.

Condition _____

1) Can a TimerTask be both Scheduled and Executed simultaneously?

2) What is an example of two or-states in ResultSet?

3) What is an example of two and-states in ResultSet?

4) What is the top-level state for a UrlConnection?

5) Did you like [Java/Plaid]doc?

__ Strongly disliked
__ Disliked
__ Neutral
__ Liked
__ Strongly Liked

6) Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?

__ State diagram
__ Plaiddoc
__ Javadoc

Further questions can be added at this point at experimenter's discretion.

Bibliography

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. Onward*, 2009. 1.6, 2, 2.1, 2.2.1, 2.4.1

Jonathan Aldrich, Karl Naden, and Éric Tanter. Modular composition and state update in Plaid. In *Proc. MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI, 2010. 2.1, A.3.3

Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. Permission-based programming languages (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 828–831, New York, NY, USA, 2011. ACM. 2.4.1

Jonathan Aldrich, Nels E. Beckman, Robert Bocchino, Karl Naden, Darpan Saini, Sven Stork, and Joshua Sunshine. The Plaid language: Typed core specification. Technical Report CMU-ISR-12-103, Institute for Software Research, School of Computer Science, Carnegie Mellon University, March 2012. 1.6, A.4

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. 3.2.4

Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 211–221, New York, NY, USA, 2011. ACM. 1.2, 4.1

Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 227–244, New York, NY, USA, 2008. ACM. 1.6

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. 1.1, 2.2.3, 3.2.2, 3.4

Andi Bejleri, Jonathan Aldrich, and Kevin Bierhoff. Ego: Controlling the power of simplicity. In *Proceedings of Foundations of Object-Oriented Languages*, 2006. 2.1

Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34 (2):83–108, 2008. 2.1, A.2.2

Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. A mechanism for flexible dynamic trait replacement. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 9:1–9:7, New York, NY, USA, 2009. ACM. 2.1

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. 1.1, 1.2, 1.6, 2.1, 2.2.3

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM. 1.6, 5.3

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. 1.2, 1.6, 3.2.2, 4.1, 4.5.6

Joshua Bloch. *Effective Java*. Addison-Wesley Professional, second edition, 2008. 3.2.4

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009a. ACM. 2.2.6

Bard Bloom, Paul Keyser, Ian Simmonds, and Mark Wegman. Ferret: Programming language support for multiple dynamic classification. *Computer Languages, Systems*

and Structures, 35(3):306–321, 2009b. 2.1

Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47. ACM, 2008. 2.1

Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM. 1.1

Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 50–59, New York, NY, USA, 2000. ACM. 2.1

Paul Chandler and John Sweller. Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8(4):293–332, 1991. 4.2

Michelene T.H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145 – 182, 1989. 4.2

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM. 4.7

Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002. 6.3.2

Tom Van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang De Meuter. Adding state and visibility control to traits using lexical nesting. In *Proc. European Conference on Object-Oriented Programming*, 2009. 2.1, A.2.2

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. 3.1.2

Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, New York, NY, USA, 2011. ACM. 1.2, 4.1

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. 3.3.2, 4.2

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. 1.2, 2.1

Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 435–446, New York, NY, USA, 2011. ACM. 2.1

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, 2001. Springer-Verlag. 2.1

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006. 2, 2.1, 2.2.4, A.1, A.2.2, 3

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. 1.2, 2.1, 4.1

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. 3.1.2, 4.2

David M Fetterman. Ibsen's baths: Reactivity and insensitivity. (a misapplication of the treatment-control design in a national evaluation). *Educational Evaluation and Policy Analysis*, 4(3):261–279, 1982. 4.6.1

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. 1.2, 4.1

Richard Gabriel. The rise of "worse is better". *Lisp: Good News, Bad News, How to Win Big*, 1991. 2.4.5

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. 2.1, 2.2.1, 3.1.2

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM. 1.1

Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. 4.2

Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM. 4.7.1

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, pages 1–48, 2013. 4.7.1

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. 3.2.1

David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. 1.2, 2, 2.1, 2.2.3

Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006. 9

Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism

for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973. 2.1

Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. 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 237–240, New York, NY, USA, 2005. ACM. 4.2

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, volume 1381, pages 122–138, 1998. 2.1

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. 2.1

Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM. 4.7.1

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001. 2.4.2

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. 3.2.4

Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP 2009 – Object-Oriented Programming*, pages 27–51. Springer Berlin Heidelberg, 2009. 3.2.4

Ciera N.C. Jaspan. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116. 1.1

Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4): 320–351, December 1996. 4.2

Alan C. Kay. The early history of Smalltalk. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of programming languages—II*, pages 511–598. ACM, New York, NY, USA, 1996. 1.1, 2.1

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. 3.1.1

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. 3.1.1, 3.3.1

George Kuk. Strategic interaction and knowledge sharing in the kde developer mailing list. *Management Science*, 52(7):1031–1042, 2006. 3.2.1

Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987. 4.2

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. 3.1.1

Donna Malayeri and Jonathan Aldrich. CZ: multiple inheritance without diamonds. *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2009. A.2.2

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. 3.2.1

Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 683–702. ACM,

2012. 4.7.1

Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998. 3.1.2

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, January 2012. 1.6, 2.4.1, 5.3

Lisa Rubin Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, pages 63–68, New York, NY, USA, 1989. ACM. 4.2

Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Mechanisms for Specialization, Generalization, and Inheritance*, MASPEGHI, 2013. 6.3.1

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the scala programming language: Second edition. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland, 2006. 5.2

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. 3.2.1

Barbara Pernici. Objects with Roles. In *Proc. Conference on Office Information Systems*, 1990. 2.1

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005. A.2.2

Blaine A Price, Ronald M Baecker, and Ian S Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993. 6.3.2

John Reppy and Aaron Turon. Metaprogramming with traits. In *Proc. European Conference on Object-Oriented Programming*, 2007. 2.1, A.2.2

Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011. 3.1.2

Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratch-

ford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013. 1.2, 4.1

Robert Rosenthal and Ralph L. Rosnow. *Essential of Behavioiural Research: Methods and Data Analysis*. McGraw-Hill Higher Education, New York, NY, USA, third edition, 2008. 4.6.1

Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, September 1996. 4.2

Sukyoung Ryu, Changhee Park, and Guy L Steele Jr. Adding pattern matching to existing object-oriented languages. In *ACM SIGPLAN Foundations of Object-Oriented Languages Workshop*, 2010. 5.2

A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003. 4.7

Darpan Saini, Joshua Sunshine, and Jonathan Aldrich. A theory of typestate-oriented programming. In *Proceedings of Fundamental Techniques for Java-like Programs (FTfJP '10)*, 2010. 1.6, 5.3

William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Cengage Learning, 2002. 4.6.1

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. 3.1.1

Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking SAML: Be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium, Security*, volume 12, pages 21–21, 2012. 1.1

Asher Sterkin. State[chart]-oriented programming. In *Proceedings of Multiparadigm Programming with Object-Oriented Languages*, 2008. 2.1

Sven Stork. *ÆMINIUM Freeing Programmers from the Shackles of Sequentiality*. PhD thesis, Carnegie Mellon University Institute for Software Research, Pittsburgh, PA, March 2013. 2.4.1

Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems*

languages and applications, pages 933–940. ACM, 2009. 2.2.5, 2.4.1

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. ÆMINIUM: A permission based concurrent-by-default programming language approach. *Transactions on Programming Languages and Systems*, To appear 2014. 2, 2.4.1

Anselm L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, June 1987. 3.3.2

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. 1.2, 2, 2.1

Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM. 4.7.1

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. 3.1.2

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. 3.1.2, 4.2

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. 1.6, 2.4.1

Dean F. Sutherland and William L. Scherlis. Composable thread coloring. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 233–244, New York, NY, USA, 2010. ACM. 4.7

Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object Oriented Programming*, 6(3):25–32, 1993. 2.1

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. 3.2.1

Brygg Ullmer and Hiroshi Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3.4):915–931, 2000. 4.2

David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. 1.1, 2.1

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. 3.2.1

Barnabas J. Walther and Abraham S. Ross. The effect on behavior of being in a control group. *Basic and Applied Social Psychology*, 3(4):259–266, 1982. 4.6.1

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. 1.2, 3.2.2, 4.1

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming 2011*, 2011. 1.6, 2.1, 2.4.1, 5.3

Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 60–68, New York, NY, USA, 2000. ACM. 4.2

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. 3.2.1

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009. 4.2