Usability Hypotheses in the Design of Plaid

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Joshua Sunshine
Carnegie Mellon University
sunshine@cs.cmu.edu

Abstract

Plaid is a research programming language with a focus on typestate, permissions, and concurrency. Typestate describes ordering constraints on method calls to an object; Plaid incorporates typestate into both its object model and its type system. Permissions, incorporated into Plaid's type system and runtime, describe whether a reference can be aliased and whether aliases can change that reference. Permissions support static typestate checking, but they also allow Plaid's compiler to automatically parallelize Plaid code.

In this paper, we describe the usability-related hypotheses that drove the design of Plaid. We describe the evidence, both informal and scientific, that inspired and (in some cases) validated these hypotheses, and reflect on our experience designing and validating the language.

1. Typestate in Plaid

Typestate is an abstraction that divides an object's lifetime into a sequence of abstract states, describing in which state(s) each method can be invoked, and the state transition caused by each method. For example, a file may start out in the `Closed` state, transition to the `Open` state when the `open()` method is invoked, accept several calls to the `read()` method while in this state, and finally transition to the `Closed` state when the `close()` method is invoked.

1.1. Background Hypotheses

Our work on Plaid was motivated in part by the following two background hypotheses, which developed out of our prior work on object protocols:

PLATEAU 14, October 21, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

H1: *Many components define protocols of interaction that clients must follow*

H2: *When developers are not aware of a component's protocol, they often make mistakes using it, and fixing these mistakes is hard*

In prior work, we performed an empirical study on a corpus of Java code to evaluate the first hypothesis, finding that in Java approximately 7% of all types define protocols [1]. That may not initially seem like much, but it is more than twice as many types as define generic type parameters.

We initially investigated hypothesis H2 in an empirical study of Spring and ASP.NET developer forums [5]. In our study, we observed that protocol problems with developers took hours or days to solve, even with expert help. Later, we mined protocol-related programming problems from the Stack Overflow forum and carried out observational studies in the laboratory. We observed that programmers working on these tasks spent 71% of their time searching the Java library documentation for information about the relevant protocols—and found that many of their searches were poorly supported by the standard Javadoc library documentation [8].

In case studies of Plural, a Java-annotation typestate specification and checking tool, we found few defects in source code repositories [2]. Since we did observe programmers making mistakes with protocols in our laboratory studies, we hypothesize that:

H3: *Protocol errors do not often make it into production, perhaps because they are easily caught by testing.*

If true, this hypothesis implies that tools and languages that support typestate should not have their primary aim be to correct errors in production code, but rather to help developers be more productive. This could be accomplished by helping developers learn about typestate constraints so they can write code more quickly and avoid introducing errors in the first place, or it could be accomplished by helping developers find errors more quickly than they could through testing.

1.2. A Typestate-Based Object Model

Plaid provided a novel object model that incorporated typestate in a first-class way [9]. For example, one might declare a stateful `File` abstraction as follows:

```
state File {
  val filename;
}

state Closed case of File {
  method open() {
    this <- Open { val filePtr =  };
  }
}

state Open case of File {
  val filePtr;
  method read() {  }
  method close() { this <- Closed; }
}
```

In this example, the `state` abstraction generalizes the `class` abstraction from typical object-oriented languages to describe an abstract state. The different states of `File` are represented as cases (similar to subclasses). The `open()` method is provided only in the `Closed` state. It is implemented with a primitive state transition operation, written `this <- Open {  }`, which changes the state of the current object `this` from the `Closed` state to the `Open` state.

We based this design on several hypotheses which bear on the usability of the language to represent concepts that incorporate state:

H4: *Providing state in the language's object model will enable the code to more closely reflect the programmer's intended design.*

H5: *Making states explicit will make state constraints more salient to developers who need to be aware of them.*

While our papers contain examples like the code above that intuitively seem to represent state more directly than approaches such as the State design pattern [3], it would be nice to evaluate this hypothesis more rigorously. Plaid code is still arguably less explicit than a statechart [4], but our `state` abstraction has the advantage that individual states can be reused as part of different stateful abstractions [9].

We also had hypotheses about how state support might help developers more directly:

H6: *Making states explicit will make the invariants of objects easier to understand and will help programmers avoid errors.*

H7: *Making states explicit will enable the runtime system to give developers better error messages when they misuse typestate dynamically.*

Hypothesis H6 was again not evaluated directly, but the intuition behind it is shown in the example code above. Because the `filePtr` field only exists in the `Open` state, there is no need to set the field to `null` or some other sentinel value in the `Closed` state, in which there is no meaningful value for this field. As for Hypothesis H7, Plaid's runtime is indeed able to tell the programmer when a method that does not exist in the current state is invoked. Intuitively, this seems better than allowing the call to proceed in a meaningless state and relying on the library writer to provide defensive code to check that the object is in the proper state. However, as yet we do not have concrete evidence in support of this hypothesis.

1.3. Checking Typestate Statically

We also worked on a type system that would allow us to check that clients use stateful libraries correctly—e.g. that they do not invoke `read()` on a `File` that is `Closed` [11]. Because of our earlier experience with Plural, we did not expect this type system to eliminate many defects from production code. Instead, we hypothesized that:

H8: *A significant benefit of types (including protocol types) is that they provide correct and easily accessible (e.g. via the IDE) documentation for programmers, helping programmers write code more quickly and correctly.*

Recently, we were able to validate this hypothesis indirectly. We looked at the protocol-related programming tasks from the qualitative study mentioned above [8] and isolated questions that programmers had to answer about protocols in order to carry out these tasks. One benefit of static typestate information is that Javadoc-like documentation can be generated that is organized by the state the object is in, and makes state transitions explicit. In a laboratory experiment, we showed that developers were able to answer protocol-related questions in half the time and with fewer errors when they used typestate-enhanced documentation compared to plain Javadoc [10]. Our work reinforces earlier result that also found that types provide documentation benefits [6].

2. Permissions in Plaid

In order to support static typestate checking in Plaid, programmers must declare a *permission* for each variable, describing whether the variable is aliased and whether the aliases could change the state of the object. For example, a `unique` variable is unaliased, and so the type system can easily track changes to its typestate through that variable. On the other hand, a `shared` variable may be aliased by other variables, and so the static type system must conservatively assume that the

typestate of the object it points to could be changed through other references.

Our earlier work in the Plural system also leveraged permissions. However, we hypothesized that:

H9: *We can provide an easier-to-use type system by building permissions into the language's type system, compared to layering permissions on top of Java.*

We did not evaluate this hypothesis empirically; however, the types in Plaid are quite obviously more succinct and anecdotally seem simpler to us than the previous systems we designed for Java. Building permissions into the language also allowed us to explore run-time checking of permissions, supporting casts and/or a gradual type system [11]. This leads to another hypothesis:

H10: *A type system that provides run-time checking in the form of casts or similar constructs can be more usable than a system without run-time checks, because the latter may require complex static constructions in places where the former uses a cast.*

A concrete example of H10 is Java's original type system, which did not support generics. Generic collections were supported by using casts to get the proper type of object out of a collection. This resulted in many run-time checks that could fail, but it did keep Java's original type system very simple.

Another hypothesis regarding permissions was:

H11: *Permission assertions are useful in their own right for design documentation or encapsulation.*

While we did not investigate this hypothesis in the Plaid project, other research has applied permissions similar to the ones we used in Plaid to a variety of problems including both design documentation and encapsulation. We believe that identifying a core set of permissions that provides a wide variety of benefits is a fruitful direction for future work in language design.

3. Parallelism in Plaid

Plaid was designed not just to support typestate, but also parallel programming. The central idea is that Plaid's permissions provide additional information about aliasing that can be used to parallelize the program automatically. We hypothesized that:

H11: *By leveraging the same permissions for both typestate and concurrency, we can provide programmers with more benefit per unit cost compared to using separate permission and type systems for these features.*

In the Plaid project, we did indeed show that it was possible to leverage the same core set of permissions for both purposes [7], but did not explicitly evaluate the comparative costs and benefits. A core aspect of Plaid's approach to parallelism (we used the name Æminium to capture Plaid's concurrency-related features) was that programmers can focus on the dependencies within the program rather than multiple threads of control, which we hypothesized would provide usability benefits:

H12: *It is easier for programmers to think correctly about dependencies rather than multiple threads of control.*

H13: *Programmers using Æminium's Parallel by Default model will expose more concurrency than is typically exposed in explicit concurrency models.*

We hope these hypotheses can be evaluated empirically in the future.

4. Discussion and Conclusions

The Plaid language design was based on a number of hypotheses that touch on the usability of programming languages, especially with respect to typestate, type systems, permissions, and concurrency. We were able to validate a few of the hypotheses, especially those that motivated our focus on typestate in the first place, and examined the documentation benefits provided by typestate. At the same time, many hypotheses that were validated at best anecdotally. Providing empirical validation for hypotheses such as these is a challenging task: while each of the papers on Plaid's language design was based (explicitly or implicitly) on several hypotheses, the few hypotheses we did validate each took at least one paper to validate, and more work could be done even on these.

Most of the hypotheses listed in this paper deal with the usability of particular features of Plaid as independent entities. We have some evidence, however, that some features work less in combination than they do independently. In particular, in unpublished pilot studies participants confused access permissions with typestate annotations. For example, in one task all three participants thought that the pure permission 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. We would like to investigate these combinations further.

While we would have liked to validate all the hypotheses on which the design of Plaid is based, in practice we are happy that some progress has been made toward this goal. The process of science involves the generation of hypotheses, the generation of artifacts to test them (concrete programming language designs, in this case), and evaluation of the hypotheses. Each of these has a role to play and a scientific paper may provide value in any of the three areas. In particular, there is value in designers of languages making their hypotheses about usability more explicit, and we thank PLATEAU for giving us a chance to do that for Plaid. Ultimately,

we hope that researchers (both in our group and outside it) will be able to evaluate some of these hypotheses more fully in the future.

Acknowledgements

This research was supported in part by the National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages." This document was prepared using Madoko.

References

[1] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*, 2011.

[2] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API Protocol Checking with Access Permissions. In *Proc. European Conference on Object-Oriented Programming*, 2009.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8 (3): 231–274, June 1987.

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

[6] 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 *Object-Oriented Programming Systems, Languages, and Applications*, 2012.

[7] 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*, 36 (1): 2:1–2:42, March 2014.

[8] Joshua Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, 2013.

[9] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.

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

[11] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming (ECOOP)*, 2011.