User-Centered Design of Permissions, Typestate, and Ownership in the Obsidian Blockchain Language

Michael Coblenz
Jonathan Aldrich
Joshua Sunshine
Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217, USA
mcoblenz@cs.cmu.edu
jonathan.aldrich@cs.cmu.edu
sunshine@cs.cmu.edu
bam@cs.cmu.edu

About the authors: Michael Coblenz is a fourth-year PhD student in the Computer Science Department at Carnegie Mellon University. After working as a software engineer at Apple for eight years, he returned to academia to research new methods for creating and evaluating programming languages to make software engineers more effective. He is advised by Jonathan Aldrich, a Professor in the Institute for Software Research, and Brad Myers, a Professor at the Human-Computer Interaction Institute. Joshua Sunshine is a Systems Scientist at the Institute for Software Research.

Abstract

Blockchains have been proposed to support transactions on distributed, shared state, but hackers have exploited security vulnerabilities in existing programs. In this paper, we describe how we are applying user-centered design in the creation of Obsidian, a new language that uses typestate and linearity to support stronger safety guarantees than current approaches for programming blockchain systems. We show how an iterative, user-centered design process can be used to elicit design feedback and show how we incorporated that feedback into the language. We found that formative user studies, even with a small number of participants, can lead to useful insights in language design. The study results motivated important language changes, such as adding explicit ownership transfer syntax and changing the structure of state initialization in state transitions.

Introduction

Blockchains have been proposed to address a variety of security and reliability objectives for computing systems. By recording all transactions in a tamper-resistant *ledger*, blockchains attempt to facilitate secure, trusted computation on a network of untrusted peers. Programs, called *smart contracts* [17], can be deployed in the ledger; once deployed, they can maintain state. For example, a smart contract might represent a bank account and store a quantity of virtual currency. Clients could conduct transactions with bank accounts by invoking the appropriate interfaces on the corresponding smart contracts.

Proponents have suggested that blockchains be used for a plethora of applications, such as finance, healthcare [14], supply chain management [5], and solar energy production [6]. Unfortunately, prominent blockchain applications have included security vulnerabilities, through which over $80 million worth of virtual currency was stolen [4, 3]. Because platforms require that contracts are immutable, bugs cannot be fixed easily. Furthermore, the application domain for blockchain is typically correctness-critical: bugs can have serious financial (consider banking) or safety (consider the food supply chain or medical records) consequences. If organizations are to adopt blockchain environments for business-critical applications, there needs to be a more reliable way of writing smart contracts.

Programming languages are interfaces that programmers use to implement software. The HCI community has a long history of developing tools for software engineers and programmers, building IDEs and studying development practices [9], creating debuggers [8], and designing programming languages for novices [13]. However, human-centered design has not been adopted by the programming language design community (with a small number of exceptions) [16]. By observing that programmers are people too [10] and adopting a user-centered approach in the design of programming languages, we hope to obtain languages that are more *usable*: ones in which it is easier for programmers to achieve their goals.

In the context of blockchain software development, we seek a language that is less error-prone: a language that facilitates development of software that achieves users' objectives while avoiding serious bugs or security vulnerabilities. However, it would not achieve our goals if we created a language that only a few experts in software verification or programming languages could use, since a much broader audience will be writing blockchain applications. Instead, we seek a language that strikes a balance between *ease of use* and *safety* so that a broad base of programmers can use it effectively.

Human-centered programming language design

The goal of using human-centered design for programming languages is particularly challenging because the results of a laboratory study of a language design inevitably depend on the tasks given and the skills and experiences of the participants. It would seem likely that the language that would evaluate best is the one that a participant already knows! Furthermore, if it can be shown that language A is better than language B for some tasks and programmers, we must ask:

1. If there were several differences between A and B, which were responsible for the difference?
2. Do the results generalize to real-world tasks, which may take much longer than the small tasks in the lab, and pertain to much larger codebases than the participants can understand in a short lab session?
3. Do the results generalize to real-world programmers, given that the experiment was done with one population (perhaps undergraduates) and not, for example, practicing software engineers?

Finally, how will we evaluate non-trivial changes to programming languages, which might require significant training or experience to use effectively?

We are using several techniques to address these difficulties. First, rather than attempt to teach participants an entire language in a brief time, we select specific features of the language that we wish to evaluate, and *back-port* those to a language that is commonly known, such as Java. We take advantage of the orthogonality design criterion for programming languages [12, 15] to argue that this is likely to produce evidence that is relevant to the language we are designing. Second, we use *triangulation* to obtain evidence of usability and utility. Rather than relying solely on laboratory studies, we use a collection of additional methods, such as case studies in which we apply our language to real systems, and interviews with experts, in which we can leverage their experience to inform the design. The laboratory studies are limited to aspects of the language that we can teach effectively in a short amount of time, but this limitation forces us to design a language that is easy to learn quickly.

Through user studies, we seek to show that users who use *our* language can write programs effectively (in a comparable or shorter amount of time as those using other languages), and their programs are more likely (or are guaranteed) to have particular correctness properties.

One example of our approach can be found in the development of our Glacier system, which provides *immutability* Java [2]. We showed how to design immutability restrictions for Java that people can use effectively to prevent bugs that they would otherwise be likely to create.

Design of Obsidian

Many techniques promote program correctness, but our focus is on programming language design so that we can prevent bugs as early as possible. We are developing *Obsidian*, a domain-specific language for blockchains that provides strong compile-time features to prevent bugs while enabling the kinds of applications that proponents of blockchain systems have advocated. Obsidian is a *transactional*, *typestate-oriented* language that supports *linear resources* for safe transactions in programs that have high-level state and track resources that should not be accidentally lost.

It does not suffice to endow the language with sophisticated features if they cannot or will not be used in practice. Our focus is on *usable* features: ones that we can show that people can use effectively. Although techniques for developing programming languages in a user-centered way are not yet mature, one of our research goals is to identify, refine, evaluate and, when necessary, create such techniques. We have adapted methods from the human-computer interaction literature to make it more likely that Obsidian will be a practical, effective language for programmers to use. We show techniques that can be used to help refine existing ideas from the PL community to increase their adoptability.

Typestate and Permissions

Our analysis of proposed blockchain applications and existing smart contracts showed that a large portion maintain *high-level state*. The available operations depend on the current state. For example, a Bond contract might be either `Offered` or `Purchased`. A `Offered` bond includes a `buy` transaction, but a `Purchased` bond does not. Aldrich et al. investigated use of typestate [1] to provide static (i.e. compile-time) guarantees that operations are only invoked when they are safe. For example, the `Bond.Offered` type guarantees that the bond it references is in the `Offered` state and therefore `buy` is available. Attempting to invoke `buy` on a `Bond` that is not statically known to be of type `Bond.Offered` will cause the compiler to emit an error.

If there are multiple references (*aliases*) to a given object with mutable typestate, they cannot all have typestate guarantees. Instead, we use *permissions* to specify which references provide which static guarantees and allow which operations.

Is there a permission system that users will understand and use effectively? If so, what can we learn from users about how to design it? We conducted the first studies (of which we are aware) in which people other than the designers were asked to use a typestate system.

In order to study permissions, we extracted the permission system from Obsidian and re-cast it in Java as a set of annotations. Rather than implementing a permission system in Java, we conducted a Wizard-of-Oz study [7] where participants received documentation on an extension to Java and the experimenter provided simulated compiler error messages. The training materials explained the Java annotations: `@Resource` (on classes); and `@Owned`, `@Shared` (for default permissions on objects), and `@ReadOnlyState` (on references). Our goal was not to gather quantitative information about the frequency of usability problems; instead, we assume that any problem we see occurring with multiple participants is worth fixing if possible.

We sought both to evaluate our proposed design and to elicit specific ways in which we could improve our design. For some questions, we used the natural programming technique [11], in which we asked participants to give us design ideas in as unbiased a manner as possible.

User Studies

In order to achieve our goals of helping people write smart contracts effectively, we wanted to make design decisions in a human-centered manner. We used *formative user studies* to inform our initial design rather than waiting until the end to evaluate a finished design. Our studies took place in a laboratory, where we invited participants to complete programming-related tasks. We recruited six participants, all of whom had experience with Java, on campus. Participants were assigned to a portion of the study according to the investigator doing the recruitment. For practical reasons of participant recruitment, sessions per participant were limited in length to between one and two hours. Due to space constraints, we describe here only a small fragment of the study.

Limitations of the user study include the limited set of participants, short duration and artificial tasks, extraction of features to a different language, and influence of documentation. However, the purpose of the study was to find usability problems that may occur with real users, not to compare the language in a conclusive way to another approach.

The study included five parts; we summarize only one here due to space constraints. Since our goal was to identify as many usability problems as possible and not to obtain quantitative results, we revised the instructions after each participant to maximize effectiveness and make the best use of limited participant time.

We were interested in whether people could program effectively with ownership, and whether there were any changes we could make to the language to make using ownership easier for programmers. We gave participants a tutorial on ownership and told them we had chosen (*no annotation*, `@ReadOnly`) (P14) or (`@Owned`, *no annotation*) (later participants) to denote ownership. We asked them to modify some provided code to fix a bug in which a `Prescription` could be deposited in more than one `Pharmacy`, resulting in allowing too many prescription refills. The intent was for participants to require that `Prescriptions` deposited in a `Pharmacy` be owned objects and that the `Pharmacy` take ownership; thus, a deposited `Prescription` could not subsequently be deposited in a different `Pharmacy`.

We were surprised that many of the participants found this task very difficult. We expanded the tutorial to include a practice section for later participants. In general, participants were not prepared to use a type system to address a bug that they thought of in a dynamic way. Several of the participants wanted to fix the problem by introducing a global registry of prescriptions rather than storing them per-pharmacy, or by making prescriptions mutable so the number of refills used could be stored there. We asked them explicitly to use the language feature instead. P14 commented "I haven't seen. . . types that complex in an actual language . . . enforced at compile time."

P14 and P17 described thinking about the problem in a *dynamic* way rather than a static way, which was a problem when using a static tool. For example, P17 wrote `if (@Owned prescription)`, attempting to indicate a dynamic check of ownership. P18 had trouble guessing the compiler's behavior, expecting a sophisticated interprocedural analysis rather than typechecking. In a case where an owned object was being consumed twice, he expected the compiler to give an error on the second `spend` invocation rather than on the invocation of a method that took an owned argument and invoked the second `spend`.

Using ownership requires determining which variables should be annotated `@Owned`, a problem that three participants had difficulty with. In one case, a lookup method took an object to search for, but P18 specified that it should take an owned reference. Then he was stuck after invoking it: "How can I get the annotation back?" Likewise P18 was confused by accessors: should they return an owned reference? In P20's case, making a class that was contained in a collection `@Owned` unnecessarily was a costly mistake because then he had a problem iterating through the collection. He made the loop index `@Owned`, which would require removing each item from the collection when iterating over it in code that was not supposed to modify the container at all.

This suggests that ownership alone, as part of the type system rather than as part of the dynamic semantics, can pose substantial usability problems. We are evaluating integrating ownership with typestate for a unified, simpler approach that might be more understandable. We are also evaluating adding syntax for static assertions regarding typestate and ownership to make it clearer what type each variable has at a given program point.

The results here revealed concerns about the design of the typestate system. First, typestate inference may be important to reduce the annotation burden. Second, users may have difficulty adding typestate specifications on variables when appropriate. With a real compiler, this would result in a lot of error messages when users attempt to invoke methods that are not guaranteed to be available; again, typestate inference may help. Finally, guaranteeing typestate requires understanding and using ownership, so priority should be on helping people use ownership effectively.

Understanding the limitations of the type system and compiler may be an obstacle for some people. Thinking about using static features rather than dynamic tests proved unnatural for some. Users will need training to reason about what typestate can do, but tools could mitigate the problem by providing sophisticated static analyses rather than taking a traditional typechecking approach, and by providing detailed, explanatory error messages.

Conclusions

Obsidian represents a promising approach toward a safer, more usable way of programming for blockchains. Though the computational environment for code running in block-

Acknowledgments: This material is based upon work supported by NSF grant CNS-1734138, NSF grant CNS-1423054, by NSA lablet contract H98230-14-C-0140, by the Software Engineering Institute, and by AFRL and DARPA under agreement #FA8750-16-2-0042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

Language Modifications
We modified the language as a result of the user studies. The example in Fig. 1 shows a version after the changes:

- Transactions (the externally-invokable APIs) are lexically outside states, but the IDE automatically inserts declarations in states (lines 4-5, 8)
- Destination states can be configured before transitions: `S::x = r1; ->S` (15)
- Transitions can return resources: `r1 = ->S` (28)
- Explicit `owned` syntax for ownership transfer at assignment/invocation (15, 34)
- Keyword for dropping ownership: `disown` (20)
- Compiler infers typestate via a control flow analysis when possible (36)

Figure 1: Obsidian Language Example

```
1  contract Wallet {
2    state HasMoney {
3      owned Money m;
4      transaction forgetMoney1 ();
5      transaction forgetMoney2 ();
6    }
7    state NoMoney {
8      transaction getMoney (owned Money m);
9    };
10   Wallet() ends in NoMoney {
11     ->NoMoney;
12   }
13   transaction getMoney (owned Money m)
14     available in NoMoney ends in HasMoney {
15     HasMoney::m = owned m;
16     ->HasMoney;
17   }
18   transaction forgetMoney1 ()
19     available in HasMoney ends in NoMoney {
20     disown m;
21     // transition returns no resources
22     // because they have been disowned
23     ->NoMoney;
24   }
25   transaction forgetMoney2 ()
26     available in HasMoney ends in NoMoney {
27     // resources returned from transition
28     owned Money oldMoney = ->NoMoney;
29     disown oldMoney;
30   }
31 }
32 contract Test {
33   transaction putAndGetMoney ()
34     available in NoMoney {
35     owned Money m = ...
36     Wallet w = new Wallet();
37     w.putMoney(owned m);
38   }
39 }
```

chain is mostly traditional (e.g. serial, shared-state), the high-stakes and other aspects of the application domains offer promising opportunities for a much better programming language than is currently in use. Our user-centered design approach is a novel way of designing programming languages and has provided useful insight into how to make sophisticated safety-related features more usable and effective for programmers.

REFERENCES

1. Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 1015–1022. DOI: http://dx.doi.org/10.1145/1639950.1640073
2. Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*.
3. Luke Graham. 2017. $32 million worth of digital currency ether stolen by hackers. (2017). Retrieved November 2, 2017 from http://cnb.cx/2DVcWDu
4. Emin Gün Sirer. 2016. Thoughts on The DAO Hack. (2016). http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/
5. IBM. 2017. Blockchain for supply chain. (2017). Retrieved October 31, 2017 from https://www.ibm.com/blockchain/supply-chain/
6. MIT Digital Currency Initiative. 2017. Blockchain Applications to Solar Panel Energy: Landscape

Analysis. (2017). Retrieved October 31, 2017 from `http://dci.mit.edu/assets/papers/15.998_solar.pdf`

7. J. F. Kelley. 1983. An Empirical Methodology for Writing User-friendly Natural Language Computer Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '83)*. ACM, New York, NY, USA, 193–196. DOI: `http://dx.doi.org/10.1145/800045.801609`

8. Andrew J Ko and Brad A Myers. 2004. Designing the Whyline: a Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 151–158.

9. Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987. DOI: `http://dx.doi.org/10.1109/TSE.2006.116`

10. B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. DOI: `http://dx.doi.org/10.1109/MC.2016.200`

11. Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47 (2004), 47–52. Issue 9.

12. Terrence W. Pratt and Marvin V. Zelkowitz. 1996. *Programming Languages: Design and Implementation*. Pearson.

13. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

14. Harvard Business Review. 2017. The Potential for Blockchain to Transform Electronic Health Records. (2017). `https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records`.

15. Robert W. Sebesta. 2006. *Concepts of Programming Languages, Seventh Edition*. Addison Wesley.

16. Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 283–299. DOI: `http://dx.doi.org/10.1145/2661136.2661156`

17. Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). DOI: `http://dx.doi.org/10.5210/fm.v2i9.548`