A User Study to Inform the Design of the Obsidian Blockchain DSL

Celeste Barnaby
Wesleyan University
cbarnaby@wesleyan.edu

Michael Coblenz
Carnegie Mellon University
mcoblenz@cs.cmu.edu

Tyler Etzel
Cornell University
tje44@cornell.edu

Eliezer Kanal
Carnegie Mellon University
ekanal@cert.org

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

Brad Myers
Carnegie Mellon University
bam@cs.cmu.edu

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

Abstract

Blockchain platforms such as Ethereum and Hyperledger facilitate transactions between parties that have not established trust. Increased interest in these platforms has motivated the design of programming languages like Solidity, which allow users to create blockchain programs. However, there have been several recent instances where Solidity programs have contained bugs that have been exploited. We are currently developing a blockchain-based programming language called Obsidian with the goal of minimizing the risk of common security vulnerabilities. We are designing this language in human-centered way, conducting exploratory user studies with a natural programming approach to inform our design choices. In this paper, we discuss our approach to the design of these user studies, as well as our preliminary findings.

Keywords keyword1, keyword2, keyword3

ACM Reference format:
Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. 2017. A User Study to Inform the Design of the Obsidian Blockchain DSL. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

We are designing a blockchain-based program called Obsidian [1] with the goal of minimizing the risk of common security vulnerabilities in blockchain programs. Blockchain programs written with current domain-specific languages such as Solidity [2] often contain exploitable bugs. In a particularly calamitious example, $50 million was stolen from a contract on the Ethereum blockchain called the DAO [4]. Obsidian contains several core features – namely, first-class typestate and linear resources – that will allow users to write programs safely and effectively. The target userbase of Obsidian is business professionals who will use the language to write smart contracts, and its design is thus oriented towards this domain: Obsidian programs consist of contracts – similar in function to classes in Java – which contain fields, states, and transactions – similar in function to methods.

1.1 Typestate

The DAO hack was accomplished using a reentrant function call. States have the ability to prohibit malicious reentrant calls by ensuring that external code is only invoked in a safe state. In light of this, Obsidian makes state first class: an object in Obsidian has a mutable state that restricts which transactions can be invoked on it. The states of a contract are defined as explicit blocks containing transactions and fields, which can only be accessed if the contract is in the corresponding state.

```
contract LibraryCard {
    state NoCard {
        transaction getCard() {
            ...
            ->HasCard
        }
    }

    state HasCard {
        transaction checkOutBook() {
            ...
        }
    }
}
```

Fig. 1. An example of states in Obsidian.

In the example in Figure 1, a `LibraryCard` is always in one of two states, `NoCard` or `HasCard`. The transaction `checkOutBook` can only be called on an instance of `LibraryCard` if it is in the

with paper note.
PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

HasCard state; otherwise, it will throw an exception. The -> operator indicates a state transition.

1.2 Linear Resources

Blockchain programs may manage some kind of resource, like a cryptocurrency, or a token indicating a more complicated right (e.g. ownership of a financial option). The static semantics for linear resources [5] enforce a safe, clean programming model: resources cannot be used more than once, but must be used before leaving the current scope (thus ensuring that a resource is not lost accidentally).

```
contract Treasury {
 // Money is a resource of Treasury
   contract Money { ... }

transaction t1(Money m) {
       spendMoney(m);
       Bond b = exchangeForBonds(m);
       // compiler error: m is used twice
       ...
   }

   transaction t2(Money m) {
     // compiler error: m is never used
     return;
   }
}
```

Fig. 2. Linear resources in Obsidian

Obsidian also supports path-dependent types [3], wherein linear values can have type members. In the above example, the type of Money is dependent upon a specific value of Treasury; put another way, each Treasury t has its own type t.Money.

Path-dependent types assist in writing secure contracts: without such types, every instance of Treasury would share the same Money type, and the programmer would have to manually check that all money deposited into a treasury – for example – is of the correct type. Incorrect or insufficient checks could leave contracts vulnerable to exploitation. With path-dependent types, all such checks are automatic: we can ensure that only money that comes from a specific treasury is used in transactions within that treasury.

1.3 Usability

It is crucial that our language is usable – that is, real programmers are able to write correct Obsidian code easily and efficiently. However, even with seemingly intuitive features, it is not always clear which approach to design is most effective for programmers. For instance, consider the following simple Obsidian contract:

```
contract C {
   state Start {
       int x;

       transaction a() returns int {
           ->S1{x1 = x};
           return b(x1);
       }
   }

   state S1 {
       int x1;

       transaction b(int y) returns int {
           return y;
       }
   }
}
```

In transaction a, we transition to state S1 and then call transaction b. However, does it make sense to call b(x1) after the transition? While we have just transitioned to S1, lexically we are still in the Start state, and it is not immediately apparent which variables and transactions are in scope. Obviously this example is extremely simple, but in a program with many different contracts, transactions, and states, a situation like this could cause serious confusion. Failure to encode state machines properly has been shown to be a significant source of errors in smart-contract programming [6]. It is critical that users are able to understand and use states easily and effectively – if not, there is potential to create the same security weaknesses as in a language without states.

Similar questions of usability arise with path dependent types. An initial approach we have taken is to use nested contracts to indicate a dependency relationship; however, nesting has different implications in different languages, and we are not certain that users will be able to recognize the presence and utility of path dependency.

A related issue is whether users will want to use these features if they are made available. Are these logical, sensible solutions to real problems that programmers face? If people do not understand or choose to use the language's core features, then any of the potential benefits of those features are lost. We are conducting a user study to investigate these questions of usability in Obsidian [7]. The key research questions we address in the study are as follows:

- Are states and path dependent types a natural way of approaching the challenges that arise in blockchain programming?
- How do people naturally express states, state transitions, and path dependent types?
- Which (if any) of our proposed ways of presenting states, state transitions, and path dependent types is most understandable and usable by programmers?
- Are people able to effectively use and understand path dependent types as they are implemented in Obsidian?

2 User Study Design

This study is explorative rather than evaluative: its purpose is to gain information about the usability of state transitions and path dependent types so we can make smart choices about the design of our language. We chose these two features as the focus of our study because they are both important safety features that we expect users will use frequently. In addition, they both have several distinct options for how to be represented syntactically, and the results of the study will factor into which option we choose to implement.

Participants are asked to complete two programming exercises, both of which are divided into parts that gradually introduce the participant to Obsidian and one of its core features. Participants are instructed to think aloud throughout the exercise, and are permitted to ask questions.

2.1 Voter Registration Exercise

Participants are offered a description of a voter registration system for a hypothetical democratic nation. The system has certain stipulations that make a state machine a logical means of representing the required behaviors. For instance, the system has specific conditions under which a citizen either becomes registered to vote, or remains unregistered.

The exercise is divided into five parts. In part one, participants are asked to implement the system using pseudocode. They are encouraged to invent any language features that they may want in order to solve this problem. We want to see how people naturally solve a problem in this domain: what ideas do they have, and what assumptions do they make?

In part two, participants are offered a state diagram that models the voter registration system, and are asked to modify their pseudocode to include the states and state transitions shown in the diagram. Again, we want to observe people's natural ideas about how to represent states and state transitions in a program.

In part three, participants are offered a two-page Obsidian tutorial detailing the key components of the language. The tutorial explains how state blocks work, but does not give any information about how transitions should be written. Participants are then given an Obsidian program that implements the voter registration system, but is missing state transitions. Participants are asked to add state transitions to the code, inventing the syntax themselves.

In part four, participants are offered three unique options for the syntax and functionality of state transitions, each accompanied by a short code example. Participants are presented the options in a random order; the order given here is arbitrary.

In option 1, users are allowed to use any transaction in any state, given they have already transitioned to that state. For instance, in Figure 3, it is legal to use the `toS2` transaction inside the `Start` state, even though that transaction is defined with `S1`. This is because there is a transition to `S1` in the previous line.

```
contract C {
    state Start {
        transaction t(int x) {
            ->S1{x1 = x};
            toS2();
        }
    }

    state S1 {
        int x1;

        transaction toS2() {
            ->S2{x2 = x1};
        }
    }

    state S2 {
        int x2;
    }
}
```

Fig. 3. One option for state transitions

In option 2, each state has a constructor that is invoked when the user transitions to that state. With this option, there cannot be any code following a state transition, so a transition must thus be the final line of a transaction. In option 3, there are conditional `if in {state}` blocks which allow the user to use the transactions and fields of that state while lexically in a different state.

Participants are asked to complete a short Obsidian contract once for each option. The contract is designed to be a simple yet non-trivial use of state transitions that illustrates the benefits and drawbacks of each option. The goal of this part is to assess participants' reactions and see whether they are able to implement the contract successfully with each option.

In part five, participants are asked to modify the Obsidian program from part three to use one of the three given options for state transitions. They are then asked to explain their reasoning and elaborate on if there was anything confusing about any of the options.

2.2 Lottery Ticket Exercise

Participants are offered a description of a program that allows users to create and participate in lotteries. Every lottery sells lottery tickets, but a user should only be able to redeem a winning ticket from the lottery where they bought it – thus motivating the use of path dependent types.

The exercise is divided into two parts. Part one mirrors the voter registration exercises in that participants are asked to implement this program using pseudocode. Again, we want to see how people naturally go about solving this problem. Will using path dependent types – or some feature similar to that – occur to anyone?

In part two, participants are given an explanation of path dependent types and offered an Obsidian contract that implements the lottery program, but has two transactions left unwritten. Participants are asked to write those transactions. We want to observe whether people are able to understand path dependent types and write correct code using them after only a brief introduction.

3 Discussion of Design

An important part of designing the user study was ensuring that the programming exercises have an appropriate level of difficulty. The scenarios must to be simple enough that participants can comprehend and implement them in the little time they have; however, they must also to be complex enough that implementing them is a non-trivial problem that actually motivates the use of Obsidian's features. Our first several pilot studies revealed that the initial design of our exercises was too complicated, and participants took much longer to read and understand the instructions than we had anticipated. Additionally, there were parts of the exercises that people were continually confused about, which made it difficult to assess their ability to use the language. We could not tell if people were confused about aspects of Obsidian, or about the instructions.

As we revised the programming exercises, we trended towards simplifying and condensing. For instance, the state diagram in the voter registration exercise originally had six states and five transitions, but was modified to have three states and three transitions; the tutorial was cut from three pages to one and a half; and two parts were removed from the lottery ticket exercise. We also made sure that each exercise targeted exactly one feature: the voter registration exercise is focused only on state transitions, the lottery ticket exercise on path dependent types.

We found that simplifying the exercises allowed us to collect better data. Participants who completed the simplified exercises spoke their thoughts aloud more consistently and stated their preferences and opinions more confidently. They were able come up with better and more interesting solutions to the problems, and write Obsidian code more effectively. But simplifying our programming exercises also created certain limitations. Since the exercises were short and not very complex, participants' opinions may have been based only on a cursory understanding of the language. There may be an option for state transitions, for instance, whose utility only becomes apparent in a large, complicated contract. We are currently missing out on that potential insight.

4 Preliminary Findings

We used a convenience sample of 12 participants. They had varying levels of programming experience: some were beginners, some experts. None had any knowledge about Obsidian prior to completing the study. The majority of the participants were undergraduates studying computer science, though one was a computer science Ph.D student, and two were working in a business-related fields. Most participants only completed one of the two exercises, and some only completed several parts of one exercise due to time constraints. The exercises were revised and modified across the 12 pilots we did, though their central objectives remained consistent.

4.1 Voter Registration Exercise

Seven participants were given the voter registration exercise. When asked to write pseudocode for the voter registration exercise, the general approach every participant took was to create a globally accessible list that stored the registration status (either registered or unregistered) of each citizen. While this is a logical implementation of the problem, it was not complete secure. For example, some participants created separate lists for registered and unregistered citizens, meaning that it would be possible for a citizen to erroneously appear on both lists.

Six out of these seven participants were shown a state diagram and asked to modify their pseudocode to use states. Of these six participants, two created explicit state blocks with functions and variables inside. The rest either maintained a global state variable that changed based on the status of a citizen, gave each citizen a state field that changed based on their status (e.g. with syntax such as "Citizen.state = CANVOTE()"), or created empty, immutable states at the top of the program. Several participants did not check whether a citizen was unregistered before processing their application, meaning it would be possible for an already registered citizen to register again – which is something we expressly prohibited in the instructions.

When looking at and writing Obsidian code with states, participants asked a lot of questions about what should be allowed to happen during and after a state transition – that is, what variable are and are not in scope, what the keyword "this" refers to, and what transaction can be used. Several participants asked if there was any way to check which state you are in currently. One participant noted that he felt it should never be allowed to call transitions from one state while lexically in another, saying "I'm calling S1's transaction from code for Start." Another participant said that she felt that state transitions were like return statements, and once you make a transition there should not be any more code in that transaction. Several participants expressed a preference for the state transition option that includes state constructors, maintaining that this option was easier to understand.

4.2 Lottery Ticket Exercise

Six participants were given the lottery ticket exercise. When asked to implement the program using pseudocode, four out of the six defined a class for Lottery. The instructions specified that users of the program should be able to buy a ticket from any lottery, but must only be able to redeem a winning ticket from the lottery where they bought the ticket. Four out

of the six implemented with program without immediately recognizing or forming a solution to this problem. When the study facilitator pointed out this issue, the approach all four participants took to resolve it was to give every lottery a fixed ID. They then made sure that the function that redeems a ticket must check that the ticket's ID is equal to the lottery ID. In each case, the participant was able to understand the need to have lottery tickets be tied to lotteries in some way, but four out of the six participants had trouble executing this easily and effectively.

However, this left some room for exploitation: two participants made the lottery's ID a randomly generated number, meaning it would be possible for two lotteries to have the same ID, and one participant had the ticket owner input the lottery ID themselves upon redeeming the ticket, meaning that if a ticket owner somehow found the ID of a different lottery, they could redeem their ticket from there.

When offered an explanation of path-dependent types and given an Obsidian program to complete, all participants were able to write correct (albeit very simple) code. Four participants were asked to identify the types of two lottery tickets that had been purchased from different lotteries. Since lotteries and lottery tickets had an established dependency relationship, the correct answer was that they should have different types, even though they were both lottery tickets. Of these five participants, two were able to offer the correct answer with accurate reasoning. One vaguely said that it "seems" like they should have different types, but was not sure since it was not indicated explicitly in the code.

Three participants noted that the use of nested classes was confusing or unclear – one said, "it's usually bad practice to use nested classes in Java."

5 Discussion

The results of the pseudocode portions of both exercises indicate that states and path dependent types are not necessarily the most obvious or natural ways of solving the problems we presented. This makes sense given the backgrounds of our participants: most of them noted that they were writing pseudocode resembling the language they were most comfortable with, and thus may not have thought about inventing new, unfamiliar language features. Moreover, the simplicity of both exercises – the voter registration exercise in particular – may have made the need or these features somewhat opaque. Still, we found in many cases that the approaches participants did take to implementing the programs was insufficient or unsafe, and the weaknesses in their programs could have been solved by using states or path dependent types. This result, coupled with our strong background evidence in the validity of these features, motivates us to continue developing these features in Obsidian, while further investigating the best ways to implement them and present them in exercises.

The results of the voter registration exercise indicate that a majority of participants prefer to write state transitions using state constructors. The fact that participants preferred this option after a short coding exercise is not conclusive prove that this is the best option or the one we should implement; however, it does indicate that Obsidian users want it to be simple and easy to tell which variables are in scope and determine the current state of an object. Our results offer evidence that encapsulating all the actions of a state within that state will allow users to understand clearly which state an object is currently in and which transactions and fields they are allowed to use – thus enabling them to write better code.

The responses we received from participants in the lottery ticket exercise reveal that nesting contracts is likely not the most understandable way to express a dependency relationship. Three participants commented made comments about this, and those who did not were not able to identify path-dependent types correctly. An alternative to this approach would be to prohibit nesting and instead use a keyword to denote this relationship (e.g. "`resource contract LotteryTicket of Lottery`").

6 Future Work

In the future, we will conduct more user studies with our current exercises and introduce new programming exercises.

- We will target people in the business domain (e.g. business students and business analysts) with limited programming experience, in order to collect data from the intended user base of Obsidian.
- We will design programming exercises that further address the usability of linear resources. One question of interest, for example, is how to enforce linearity for field accesses and state transitions: what should happen when you attempt to access an owned field that has already been read once, and how can you transition between states with different owned fields?
- We will design programming exercises that require participants to read and write longer, more complicated Obsidian contracts that actually compile. This will offer us more evidence about whether people are able to write correct Obsidian code. It will also allow participants to gain a deeper, less superficial understand of Obsidian's features and thus offer more constructive feedback about Obsidian's usability.
- Finally, we will implement the Obsidian language using the results of these studies, and evaluate the final design in a formal study that will test its effectiveness.

7 Conclusion

...

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions

or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Coblenz, "Obsidian: A Safer Blockchain Programming Language," in *Proceedings of the 39th International Conference on Software Engineering - ICSE '17,* 2017.

[2] ——, "Solidity," https://solidity.readthedocs.io/en/develop/. Accessed Aug. 3, 2017.

[3] N. Amin, T. Rompf, and M. Odersky. "Foundations of Path-Dependent Types." In OOPSLA, 2014.

[4] E. Gün Sirer, "Thoughts on the DAO hack," 2016. [Online]. Available: http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

[5] P. Wadler, "Linear Types Can Change the World," IFIP TC, vol. 2, pp. 347 - 359, 1990.

[6] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab." IACR Cryptology ePrint Archive, vol. 2015, p. 460, 2015.

[7] B. Myers, A. Ko, T. LaToza, and Y. Yoon, "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," IEEE Computer, Special issue on UI Design, 49, issue 7, July, 2016, pp. 44-52.