Obsidian in the Rough: A Case Study Evaluation of a New Blockchain Programming Language

Paulette Koronkevich
Carnegie Mellon University Indiana University
pkoronke@iu.edu

1 INTRODUCTION
Obsidian [2] is a typestate-oriented blockchain programming language, designed in a user-centered way. A blockchain is a decentralized, distributed ledger managed by a peer-to-peer network [7]. Programs installed on the blockchain, called smart contracts, maintain state which can be mutated through transactions.

Blockchain programs are often oriented around states. Obsidian uses typestate [1] to lift dynamic state to static types, enabling the programmer to reason about protocols at compile time. For example, an insurance Policy may be in an Offered, Active, or Expired state, where attempting to purchase an active or expired policy results in a type error.

To enforce these protocols in the presence of aliasing, Obsidian uses permissions [4] to reason about references and their possible aliases. Owned references are guaranteed to be the sole mutable reference to an object, with all aliases being Unowned, or read-only, references. A reference with a state specification is Owned to ensure the specification is not violated. Owned references can be split into Shared references, which are mutable but cannot guarantee typestate or that there are no other Shared references. Shared references follow the standard semantics of references in OO languages. Unowned references cannot mutate the object.

This paper describes a case study to evaluate Obsidian’s design. We implemented ParamSure, a blockchain application for parametric insurance, to address the following questions:
• Can Obsidian express the protocol required using its current features?
• Can Obsidian express the operation of the program in a domain appropriate way?

Parametric insurance is a type of insurance that insures against events outside the policyholder’s control. For example, a farmer could buy parametric insurance to insure crops against weather events that could damage them, such as excessive rain. The farmer receives compensation if the event occurs, whether or not any damage occurred. In contrast, traditional insurance insures against pure loss, which relies on trust between the insurer and the insured since an insurance adjuster assesses the loss.

We also implemented a smaller prototype of ParamSure in Solidity [5], another blockchain language, as a comparison.

2 RELATED WORK

3 METHODOLOGY
Our criteria for choosing the case study were as follows: (1) the application should benefit from being deployed on the blockchain, (2) there should be an unbiased client to enforce realism in the design, and (3) it should include features found in other blockchain applications. We chose ParamSure because it meets all of these criteria, as we will now describe.

The World Bank is deploying a blockchain-based parametric insurance platform to provide agricultural insurance in developing countries. ParamSure is our prototype of this platform. We met with their team to elicit requirements for our case study. We iteratively implemented a prototype of ParamSure, confirming that we were following the desired structure. Since our client is responsible for the design, we cannot handpick a structure that is a perfect fit for Obsidian, which increases the realism of the case study.

Parametric insurance can benefit from deployment on the blockchain. Traditionally, the insured would need to trust
contract Policy {
    enum States { Offered, Active, Expired }
    States public currentState;
    int public cost , expirationTime;
    constructor (int c, int expTime) public {
        cost = c;
        expirationTime = expTime;
        currentState = States.Offered;
    }
    function activate () public {
        require ( currentState == States.Offered );
        currentState = States.Active;
        cost = -1;
        expirationTime = -1;
    }
    function expire () public {
        require ( currentState == States.Offered );
        currentState = States.Expired;
        cost = -1;
        expirationTime = -1;
    }
}

Figure 2: Solidity implementation (left) and Obsidian implementation (right) of a Policy contract.

that the insurer will pay claims when appropriate. By being deployed on the blockchain, any terms such as compensation can be automatically executed.

ParamSure also includes realistic features of blockchain programs, such as several entities interacting with each other and deploying contracts as shown in the simplified architecture in Figure 1.

4 RESULTS

We found that Obsidian’s typestate system can enforce certain protocols well at compile time. Many contracts used references with typestate specification, instead of using Shared references and dynamic checks.

Figure 2 shows the implementation of a Policy contract in Solidity and Obsidian. This contract models the insurance policy presented to the client in an Offered state. If the Policy is purchased, it transitions to the Active state using the activate() transaction. Otherwise, it eventually transitions to the Expired state using the expire() transaction.

The Solidity implementation requires dynamic checks to ensure that the Policy is in the proper state, whereas Obsidian simply requires that this must be in the Offered state to invoke either activate() or expire(). Obsidian guarantees that after calling activate(), the Policy transitions to the Active state, whereas Solidity cannot guarantee this. If a user forgets to update the current state in Solidity, the bug cannot be caught at compile time. Given a Policy, Obsidian guarantees that it is Active after calling activate(), whereas Solidity must repeatedly check, even after calling activate(). Furthermore, Obsidian directly enforces that fields are only available in certain states, whereas Solidity assigns these fields to -1 as a sentinel value.

Obsidian’s permission system, though simple, can also express powerful protocols. The PolicyRecord contract has an instance of a Policy. In Solidity, certain measures should be taken to ensure the PolicyRecord cannot be manipulated to maliciously update the state of the Policy, but in Obsidian we simply make the reference Unowned.

However, we discovered that the Owned permission makes accessing fields from contracts difficult. Obsidian currently allows fields to be incompatible with their declared type during transactions, as long as the transaction restores the declared type. Once accessed, the field is no longer Owned, and thus incompatible with its declared type. We plan to address this limitation in future designs of Obsidian.

We plan to create a similar implementation of ParamSure in SILL [9] to provide further comparison. Our hypothesis is that Obsidian is a better fit for the domain, based on the general structure desired by the clients. The implementation in SILL would require reasoning about processes and channels that may not be clear to the client, whereas typestate can model clearer protocols such as an expired policy.

5 CONCLUSION

Obsidian combines typestate and permissions in a novel way, which we hypothesize can help programmers reason about protocols at compile time. Obsidian seeks to provide these guarantees in a usable way. We evaluated these features through a case study. We found the case study useful for uncovering difficulties with the current features.
REFERENCES


