Contracts over Smart Contracts: Recovering from Violations Dynamically

Christian Colombo[2], Joshua Ellul[1,2], and Gordon J Pace[1,2]

[1] Centre for Distributed Ledger Technologies, University of Malta, Malta
[2] Department of Computer Science, University of Malta, Malta

Abstract. Smart contracts which enforce behaviour between parties have been hailed as a new way of regulating business, particularly on public distributed ledger technologies which ensure the immutability of smart contracts, and can do away with points of trust. Many such platforms, including Ethereum, come with a virtual machine on which smart contracts are executed, described in an imperative manner. Given the critical nature of typical smart contract applications, their bugs and vulnerabilities have proved to be particularly costly. In this paper we argue how dynamic analysis can be used not only to identify errors in the contracts, but also to support recovery from such errors. Furthermore, contract immutability means that code cannot be easily fixed upon discovering a problem. To address this issue, we also present a specification-driven approach, allowing developers to promise behavioural properties via smart contracts, but still allowing them to update the code which implements the specification in a safe manner.

1 Introduction

Smart contracts built on top of blockchain and other distributed ledger technologies (DLTs) have been hailed as a game changer in providing a formal interface through which to regulate interaction between real-world parties. Originally, Szabo [27] conceived the notion of smart contracts as means of automated agreement and regulatory enforcement to the extent that they *"make breach of contract expensive (if desired, sometimes prohibitively so) for the breacher"* — thus allowing for breaches of contract, and yet at a cost. This corresponds closely to the notion of legal contracts which include the possibility of breaches to the extent that they frequently include clauses to regulate what happens in case of violation of other clauses. In contrast, Lessig's [21] dictum of *"code is law"* saw computer code regulating behaviour in an incontrovertible way, and thus, e.g. if the code automatically reroutes 25% of your income to pay taxes, there is no way in which you may breach this 'law' and not pay your taxes.

The term *contract* has been used in different contexts with different meanings, ranging from legal contracts which talk about ideal behaviour agreed upon between the participating parties, to programming language contracts to allow for the specification to be part of the system implementation (e.g. pre- post-conditions in Eiffel [23] and behavioural interfaces [17]). In their current incarnation as adopted on distributed ledger systems such as Ethereum, smart contracts are closer to Lessig's view of code as law, with smart contracts providing executable transactions enforced implicitly by the underlying distributed ledger system and possibly changing its state. They provide an opportunity to

execute code affecting global state in a safe manner which would otherwise be impossible without the participation of trusted central authorities or resource managers.

Whether specifications should be executable [14] or not [18] is an old debate in computer science, but what is clear is that a non-executable specification may limit itself to describe *what* the resulting state should look like (or satisfy), while an executable one must also include a description of *how* to achieve such a state. The additional information required for the latter leaves more room for incorrect or mistaken specifications.

This is a crucial issue with the current incarnation of smart contracts: smart contracts do exactly what they say they do, but that might not be what the agreeing parties thought it would do — or for that matter what the developer of the contract thought it would do. This is particularly important since once deployed on the underlying blockchain, smart contracts are immutable and cannot be changed. The only way to support updates to a smart contract is to include the possibility to update the code in its own logic, which goes back to the question of trust. Whether a smart contract is written by one of the parties participating in a transaction, or by an outsider, participating parties may rightfully fear that there might be obscure ways in which others can exploit the contract to their benefit. There have been well-known instances of bugs in smart contracts, for instance, on Ethereum [1].

Although due to the immutable nature of smart contracts one would prefer to use static analysis techniques to ensure correctness at compile-time, such work is still sparse, with most of it aimed at addressing common vulnerabilities rather than business-logic specifications. For instance, Fröwis et al. [13] try to identify control-flow mutability, OYENTE [22] performs reentrancy detection, while Bhargavan et al. [2] transform Solidity into F* on which they perform analysis to identify general vulnerable patterns. Much of this work is performed at the EVM level, partly due to the fact that the semantics of Solidity being only informally described in the language documentation, and effectively pragmatically decided based on what the compiler does. In contrast, there are published formal semantics for EVM bytecode either through direct formalisation or via translation in [19, 16]. However, what these approaches do not address are application specific, business-logic properties; perhaps mainly due to issues of scalability, especially because of the data intensive aspects of many smart contracts. In such cases, one may have to resort to runtime analysis of smart contracts.

Runtime monitoring, already a special case of dynamic analysis, admits to a whole family of activities. At the most basic level, one can merely *monitor* or *observe* a system and log information about its runtime behaviour. The next level up is that of *runtime verification*, in which not only is the behaviour observed, but particular behavioural patterns are identified to be undesirable and algorithmically classified to be so. This notion can be taken further by adding on logic to support *runtime recovery* or *reparation*, triggering in the case of undesirable behaviour being observed to make up for it. One can also go another step further, using *runtime enforcement* to ensure that the undesirable behaviour is avoided in the first place, modifying the system's behaviour to ensure it works as expected[3]. In the rest of the paper, we primarily focus on runtime verification and recovery.

[3] Needless to say, this terminology has been used in a wide variety of contexts, and not all usages correspond to the neatly compartmentalised descriptions we give. In case of disagreement with

The very architecture of blockchain (and similar distributed ledger technologies) in itself provides the monitoring process for free. Each transaction and invocation to a smart contract is immutably recorded on the underlying ledger. The violation detection process itself can be addressed using techniques not too different from those already in use for other software systems. It is worth noting that the ledger architecture does provide an opportunity in injecting online runtime verification into the underlying design — one can design a monitoring-aware DLT in which verification code can be added to the architecture (the DLT implementation itself), ensuring no smart contracts are executed or data written unless verified to be correct. In the rest of this paper, however, we will simply assume that runtime verification is being performed, thus allowing for violations to be identified and captured. Whether this verification is performed in the traditional manner (e.g. injecting code in the smart contract to perform the monitoring and verification), or performed by modifying the underlying architecture is irrelevant.

Even just identifying such violations can be useful in practice — consider a (physical-world) legal contract which stipulates that the parties agree on legal liability whenever the runtime monitor identifies a violation. However, in this paper we concern ourselves primarily with going beyond the monitoring and verification process — looking at the choices and challenges in reacting upon the identification of points of violation, primarily in the form of reparation, but also, enforcement in a limited manner. When runtime verification detects a violation of the specification at runtime, the system is typically instructed on how to react to (i) make up for the violation from the point of view of the system logic (e.g. block an account for safety); and (ii) restore the system state to a sane one (e.g. revert a financial transaction to leave no pending transactions or locked resources). We discuss how one can support such reparations in the context of smart contracts, and show how these notions can be used to extend the existing Solidity runtime verification tool CONTRACTLARVA [11].

However, on normal systems, the detection of a violation also triggers offline behaviour outside of the system — when the system developers try to identify the origin of the bug which led to the violation, fix it and redeploy the updated system. With the immutability of smart contracts, this phase is severely crippled. One of the contributions of this paper is the proposal of a model-based approach incorporating runtime verification, to support updatable smart contracts in order to address violations discovered post deployment.

In order to be able to illustrate our ideas, in Section 2 we give a brief overview of the Ethereum platform[4] and the Solidity smart contract scripting language. In this section, we also show how CONTRACTLARVA specifications can be written, enabling us to propose concrete extensions supporting richer means of handling violation in smart contracts in the following sections. In Section 3, we then discuss the challenges of recovery from specification violations both to recover the internal state of the smart contract, but also to make up for the violation from the affected users' perspective. The

---

our use of terminology, kindly read the rest of the paper replacing the terms with your preferred ones.

[4] Since DLTs vary in design and in their take on smart contracts, we particularly focus on the Ethereum blockchain platform [28], even if many of the ideas presented herewith can be extended for other takes on smart contracts and other DLTs.

issue of dynamically addressing bugs discovered post-deployment in the context of smart contract immutability is discussed in Section 4. Finally, in Section 5, we discuss related work and draw some conclusions.

2 Smart Contracts on Ethereum

Smart contracts and the programming thereof, due to the inherent immutability of blockchains and the critical nature of applications they are used for, requires a different programming mindset [10]. Once deployed a smart contract is there forever. The internal code cannot be changed, and with this in mind, developers tend to use defensive programming techniques to ensure that users cannot exploit bugs or unintended functionality. Ethereum provides for the execution of a 'one world computer', the Ethereum Virtual Machine (EVM) [28], which can be seen as a single computational core which executes function code atomically. What is really happening though, is that every node is computing and storing the same values within the blockchain, and must therefore require computation to be deterministic (since the same result must be computed on every node). Calls to smart contracts are treated as atomic transactions, which often instills a sense of security in programmers since race conditions no longer appear to be an issue. It has been argued that smart contract programming still shares much with concurrent object programming [26] and issues such as reentrancy remain — occurring when calls are made to third party smart contracts that in turn call back the caller smart contract.

The Ethereum platform allows executable smart contracts to be written using the EVM's assembly instruction set, but also provides high-level languages, with the predominant one being Solidity. Once deployed on Ethereum, a smart contract has an associated unique identifier, corresponding to its address and can intrinsically own ether (Ethereum's internal currency) and transfer ether to other addresses (which could be contracts or user accounts). The EVM instruction set is Turing complete, and in order to deal with smart contract functionality which may not terminate or take inordinately long, uses the notion of gas — effectively payment (in ether) for the execution of each instruction step. When the gas allocated to a particular transaction is exhausted, execution stops and the altered state is reverted to the original one upon initiation of the transaction, thus effectively ensuring that (i) all functionality is terminating; and (ii) computationally more expensive functionality is also financially more expensive, thus avoiding possible attempts to overload the Ethereum platform with complex computation.

In the rest of the paper, we will use a running example of a smart contract to implement a casino which provides a single game that allows for guessing the outcome of a coin toss. The legal contract which we will be using as a running example is shown in Figure 1. This can be implemented as a smart contract on a platform like Ethereum (with part of the code in Solidity shown in Listing 1), where each party's possible actions are encoded as functions which the respective parties may invoke. The shown `closeBet` function is used by the casino owner to reveal the coin tossed after a player has made a guess, corresponding to clause 6. It is worth remarking on some aspects used in the code which will be used in the rest of the paper.

The `require` function provides a mechanism to ensure that a predicate holds before proceeding with the code. If the predicate does not hold, the whole transaction and

1. *The casino owner may deposit or withdraw money from the casino's bank, with the bank's balance never falling below zero.*
2. *As long as no game is in progress, the owner of the casino may make available a new game by tossing a coin and hiding its outcome. The owner must also set a participation cost of choice for the game.*
3. *Clauses 1 and 2 are constrained in that as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out.*
4. *The win-out for a game is set to be 80% of the participating cost.*
5. *If a game is available, any user may choose to pay the participation fee and guess the outcome of a coin toss to join the game. After that, the game will no longer be available to other users.*
6. *The owner of the casino is obliged to reveal the coin tossed upon creating the game within half an hour of a player participating. If the coin matches the guess, the player's participation fee and the game win-out is to be paid to the player from the casino's bank. Either way, the game then terminates.*
7. *If the casino owner does not adhere to clause 6, the player has the right to declare a default win and be paid the participation fee and the game win-out from the casino's bank. At this stage, the game also terminates.*

Fig. 1. A legal contract regulating a coin-tossing casino

execution of the code is abandoned, and the variables are reverted to their original values. This mechanism can also be triggered directly through the Solidity `revert` instruction (which Solidity's `require` uses internally). Reverts are bubbled up to functions calling the failing one, and the only way to stop such a revert chain is through contract communication. Contracts on Ethereum may invoke functions in other contracts through the `call` and `delegatecall` functions which stop the bubbling up of a revert. In addition, delegate calls run the called code from within the caller (i.e. giving access to variables defined in the caller function).

A function call is viewed as a message passed to the contract, accessible through the `msg` variable, and allowing access to information such as the message sender's address: `msg.sender`. A smart contract can transfer any ether owned by the smart contract to an Ethereum address through the `address.transfer(amount)` instruction.

Finally, it is worth noting that `private` variables and functions in Solidity (as opposed to `public` ones), only prevents other contracts from accessing the data directly. However, the data is still visible to anyone outside since it is publicly written on the Ethereum blockchain, so the `hiddenCoin` would have to be encrypted and not simply written to a private variable. One commonly used way is to encode the hidden coin toss by submitting and storing the hash of an odd number if it was heads, and even if it was tails. Upon revealing the actual number, it is easy to confirm that the coin was not changed and whether it was heads or tails (achieved using the function `sameAs`, the implementation of which is not shown).

It is worth noting that contracts may not only call and execute functions in the same contract, but may also have calls to other smart contracts. Solidity provides `call` and `delegatecall` functions as means to execute named functions at a given contract address, with the main difference (of interest to this paper) being that `delegatecall`

gives the called contract access to the state of the contract from where the call is made. This allows for delegation of control of state to external contracts.

When an exception is raised within the callee, the `call` function will return a `false` value (and if it was successful, a `true` value). A similar function `delegatecall` allows for calls to external contract functions which execute the external contract function code within the context of the caller's contract and caller's transaction, which will maintain the same values for the `msg.sender`, `msg.value`, and other contract context including the storage used. This can be seen to be the same as though the contract was calling another internal function, although in actual fact the code is stored in an external function.

The code in Listing 1, thus ensures that (i) it is being invoked by the casino owner; (ii) the revealed coin matches the originally given (encrypted) hidden one; and (iii) a player has participated in the game. If all three conditions hold, then the player is given a reward in case of a guess (clause 6). The game terminates after that.

```
contract Casino {
  .
  .
  address private hiddenCoin;
  .
  .
  function closeBet(uint _shownCoin) public {
     require(msg.sender == casinoOwner);
     require(sameAs(_shownCoin, hiddenCoin));
     require(gameStatus == PLAYER_PARTICIPATED);

     if (matches(_shownCoin, guessedCoin)) {
       player.transfer(participationCost + winout);
     }
     gameStatus = GAME_OVER;
  }
  .
  .
}
```

Listing 1: Part of the smart contract implementing the casino table

Consider a property which states that the casino owner may not withdraw from the casino's bank leaving less than the required player payout as long as there is an active bet. This property can be expressed as a dynamic event automaton (DEA), the specification language used by CONTRACTLARVA [11] (a runtime verification tool for Solidity contracts), as shown in Figure 2. DEAs are effectively automata whose transitions are tagged by a triple $e \mid c \mapsto a$, where e is an event, c is a Solidity condition (which has to be satisfied to take the transition) and a is a Solidity action (essentially code which is executed upon taking the transition). Both the condition and the action can be left out if not required. The events are of the form: $o :: m : f$, where f is a Solidity function name and parameters, m is the modality which will trigger it, and o is the agent who must call the function for the event to trigger. In turn, the modality can be `start` which triggers as soon as the function is called, or `end` which triggers when the function terminates successfully i.e. without a revert. DEAs also allow a `fails` modality (which will be used

later in the paper) which triggers if the function is called but is reverted for any reason other than lack of gas. DEAs are deterministic automata and include identified bad states (marked in black in the figure) which flag a violation if reached at runtime.

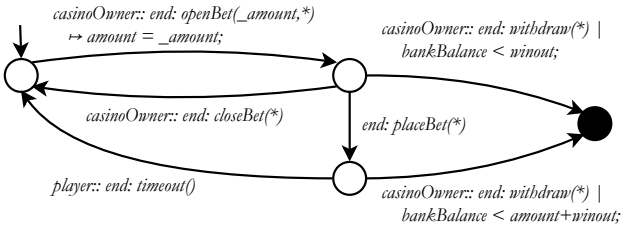

Fig. 2. Property specifying that enough funds must remain in the bank throughout the bet's lifetime.

3 Smart Contract Recovery

Detecting situations where a contract has violated applicable correctness properties is desirable, but dealing with the aftermath of such a discovery might not be straightforward. In the context of software which is not blockchain-based, one approach may simply block the execution of the whole system or of part thereof (e.g. block the users or functionality affected by the issue) until the problem is resolved. In the case of smart contracts though, naïvely blocking the contract from proceeding further would mean locking funds held within it forever, implying the need for more sophisticated recovery code.

Using a custom recovery action to manage such violations allows for comprehensive and customisable handling. For instance, in the casino property example, one might consider an escrow arrangement, in which the casino owner initially pays into the contract an amount which is paid off to the player in case of a violation to make up for the malfunctioning contract. At the most coarse grained level, recovery actions may be generic (any violation fires this recovery), but can be made more specific for particular properties or particular parts of code which trigger the violation (effectively acting similar to typical exception handling). This approach has been adopted by several runtime verification tools e.g. Larva [9] and Java-MOP [5].

While such custom recovery arrangements are convenient in that they provide a specific case-by-case solution to violations, they have the downside of being hard to automate, i.e. procedures have to be customised and coded manually, increasing the complexity of the smart contract. Taking once more the escrow arrangement example, if a contract involves a number of different stakeholders, who has to pay the escrow and how to divide it for each violation becomes substantially more complicated. We now look at a number of alternative approaches to specifying recovery in more compositional ways.

3.1 Checkpointing

One standard way of automating recovery from failure is through the use of *checkpointing* [25], i.e. to save the state of the contract at important points of execution in order to allow reverting back to them when the monitor detects a deviation from the expected behaviour. In the casino example, this would mean that money placed on a bet would automatically be returned to the player. Through the `revert` mechanism, the EVM already provides an underlying notion of checkpointing for its atomic transactions: if a transaction fails half way through, its effects are discarded by returning to the state of the blockchain before the start of the transaction, and this can be used to ensure that calls to a smart contract which cause a property to fail are completely undone, thus guaranteeing that the state is returned to its previous (assumed to be sane) state.

In using reverts to undo execution of a failed transaction on Ethereum, particular care has to be taken due to calls and delegate calls which stop a revert from being bubbled up to the caller.

However, using EVM reverts to handle system state recovery comes with a number of caveats:

Normal vs. exceptional reverts: Since reverts are typically used in the normal logic of the smart contracts (e.g. the assertions in the code shown in Listing 1 may trigger reverts), reverts now play two roles — that of normal exits from the system logic, and that of exceptions due to behaviour which was not expected from the smart contract. Care has to be taken to avoid these from interacting together, particularly since smart contracts may use calls to capture normal (expected) reverts to follow up its behaviour.

Finer grained checkpointing: The basic checkpointing mechanism provided on the EVM does not provide the possibility of fine-grained checkpointing; the checkpoint can only be (implicitly) placed at the start of the transaction. Ideally, one should be able to allow for marking checkpoints and allow for reverting to particular ones. For instance, consider if the casino smart contract were developed by someone other than the casino owner who would benefit from a transaction fee with every attempted withdrawal from the casino bank. In such a case, one may want to ensure that violation of the smart contract property from Figure 2 should revert the withdrawal from the bank, but still keep the transaction fee. One way of achieving this would be to use named checkpoints (see Listing 2) and reverting to the named checkpoint `BEFORE_WITHDRAWAL` when that violation occurs. Such a mechanism can be easily implemented using code transformation on the smart contract with the help of explicit calls. The downside of such an approach is that there is even more complex interwinding between the forward and the recovery logic, with checkpoint tags which may have been created purely for recovery appearing in the main code thus violating the often held principle of separation-of-concerns (keeping the normal logic and the verification specification separate). In [7], we had proposed an alternative to this approach in that checkpoints relevant only to recovery are also identified as part of the dynamic analysis. By adding appropriate tagging (e.g. adding a checkpoint tag after the transfer to the developer is specified on the DEA using an action or a checkpoint tagging state), one can still keep checkpoint tags relevant to reparation separate in the specification.

Forward recovery: Whilst reverting to a previous state provides a straightforward way of restoring the state of the smart contract, sometimes one still needs to perform a recovery action *after* recovering the state. For instance, in the casino smart contract, one may want to not only disallow the withdrawal, but also allow the player a default win. Such forward recovery logic can be placed in the smart contract itself, but as argued before, makes more sense in the specification (for instance by tagging the bad state with the code). Figure 3 shows how the specification can be extended with this information, adding another DEA to keep track of the relevant checkpoint upon matching a particular sequence of events.

```
function withdraw(uint _amount) public {
    require(msg.sender == owner);
    ...
    // Pay transaction fee
    developer.transfer(transactionFee);
    // Withdraw specified amount
    checkpoint(BEFORE_WITHDRAWAL);
    casinoOwner.transfer(_amount);
}
```

Listing 2: Named checkpointing for partial reverts

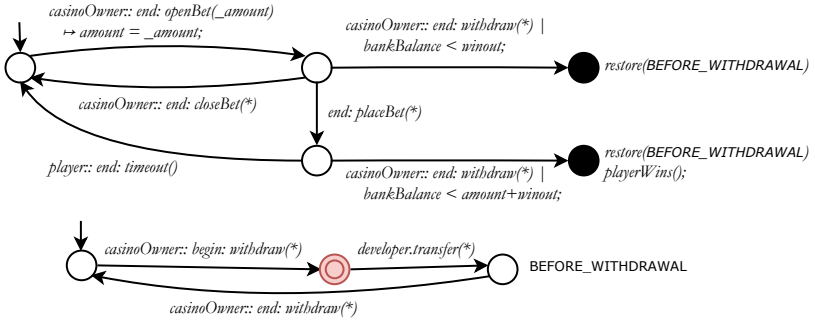

Fig. 3. Extending the property to allow for tagged checkpointing followed by forward recovery — checkpoints being saved upon entering the red state.

3.2 Compensations

The notion of forward recovery after restoring to a checkpoint, as discussed in the previous section, is typically used to make up for earlier behaviour e.g. allowing the player to win is done to compensate for the fact that the player has already (in a previous transaction) committed him or herself to betting in the casino. Although each function

call to the EVM is seen as a full transaction, from a higher level of abstraction, sequences of function calls can be seen as long-lived transactions [25, 15, 6]. Just as in long-lived transactions, previous function calls to the EVM may not always be fully reversible[5], in which case compensation for such functions cannot be feasibly done via checkpointing.

When a global compensation is applied (as in the case of giving a default win to the player), compensations can be easily handled, but when in more complex situations, one usually has compensations gathering as the long-lived transaction advances. The appealing aspect of a compositional compensation mechanism is that each individual action can be assigned a default compensation, i.e. an action which manages the effects of the action being compensated for, and unless specifically changed, the compensation of a sequence of actions results in the execution of the individual actions' compensations in reverse order. Such a mechanism is frequently used on, for instance, payment transaction systems to ensure that the participating entities are compensated for the failing transaction, also in the context of runtime verification [8, 7].

For instance, consider a casino scenario in which a player may join either a roulette or a coin-tossing table, where they may place multiple bets. A monitor can be used to ensure that if a player performs an illicit action (e.g. placing more bets than legally permitted), they will be refunded any bets they have placed (less charges, which may depend on the game they are betting on) and their account will be disabled. Figure 4 shows how this can be handled using simplified compensation automata [7] — extending the notation used earlier for transitions to add a compensation u: $e \mid c \mapsto a/u$, where u can either be an action which will be added to the compensation stack or an instruction to clear the compensation stack. When a violation is identified (by a separate monitor), actions are individually removed from the compensation stack and executed.

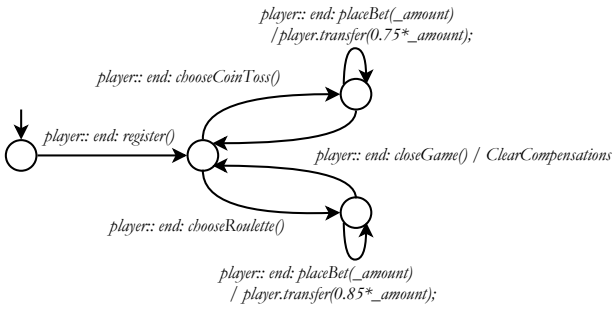

Fig. 4. Monitor-based compensation synthesis.

[5] Atomic transactions rely on locking to isolate themselves from external observation — which is impractical with transactions which have a long lifespan. If the environment reacts to intermediate results after which the transaction fails, then the transaction cannot simply be wiped out. Rather, the effects it had had on the environment in its lifetime need to be managed. This is done through compensations.

Since compensations depend on the history of execution, and can very easily lead to substantial increase in space and time execution resources required, this comes with an additional problem when used on platforms such as Ethereum, due to substantial gas consumption increase. A standing challenge is how to constrain the notion of compensations in order to avoid or mitigate this issue.

4 Updating Code

In most software systems, when a violation to a specification is discovered, one important action is to report the problem back to the development team to assess its severity and accordingly report the issue to be eventually addressed in a patch or future release. In hardware verification circles, it has always been recognised that bugs are more serious and costly, as the 1994 Intel FDIV bug [24] had shown, since one cannot cheaply update a chip post-production. In a manner, despite the software nature of smart contracts, their intrinsic immutability shares much with hardware systems. Once deployed, there is no simple manner in which one may update the code.

In order to deal with problems identified post-deployment, the industry has developed a family of design patterns in order to support code updates through having the code of the smart contract refer to updatable references to secondary smart contracts or through means of migrating users from a smart contract to an updated one. Listing 3 shows a code snippet of how this is typically done using the *proxy* or *hub-spoke* pattern. The approach involves the use of an interface contract (with no internal implementation but) with a reference to the current version of the actual contract implementation. Any function calls to the contract are simply passed on as calls or delegate calls (if the data is also stored in the interface contract) to the actual implementation contract. The primary issue with this approach is that each such contract must choose what policy to adopt in order to decide how a version update can be accepted. For instance, in the example shown in Listing 3, the casino owner would be able to unilaterally update the code, but one may adopt more sophisticated approaches, e.g. requiring updates to be decided by a majority vote amongst the current users of the contract.

```
contract Casino {
  address currentVersionOfContract;
  address owner;

  function updateVersion(address _newVersionOfContract) public {
    require(msg.sender==owner);
    currentVersionOfContract = _newVersionOfContract;
  }

  function openTable() public {
    currentVersionOfContract.call(bytes4(sha3("openTable()")));
  }
  ...
}
```

Listing 3: Enabling versioning of smart contracts

In this section we identify a solution to this challenge of enabling smart contract updates in a safe manner, building on ideas from behavioural interfaces [17], monitoring-oriented programming [4] and using dynamic analysis to ensure safety.

The major challenge faced is that unless somehow limited, code updates can be arbitrary and users of the contract have no guarantees that the new contract code will continue to implement the same logic (except for new features or fixed bugs) as the original one they signed up to. We propose a *specification-oriented approach,* in which users initially agree on a specification of how the smart contract is to behave, and set up a smart contract which (i) implements the interface of the contract; (ii) passes on any calls to the public interface to the current version of the implementation available as an external contract; (iii) enables the developer to update the version of the code arbitrarily; but (iv) instruments a monitor to ensure that the specification is adhered to by the current version of the contract. The first three are identical to the design pattern shown in Listing 3, but the fourth is what ensures user confidence in the implementation. No matter how the developer updates their code, the users are guaranteed that any violations to the specification will be captured and acted upon.

Consider, for example, a specification which a user may want to be sure holds in order to trust a casino implementation as shown in Figure 5 in terms of a DEA. The specification identifies three forms of casino implementation misbehaviour — once a bet is opened by the casino owner and a bet is placed by the user, the three violations identified are if (i) the casino reveals the number which matches the user's guess but insufficient funds are transferred on to the user; (ii) the user calls the timeout after an appropriate amount of time without the number being revealed but not enough funds are transferred to the user; and (iii) the user tries to call a timeout but is stopped from doing so by a revert.

The choice as to whether the proxy should use calls or delegate calls depends on a number of issues, including ones related to monitoring. For instance, if some properties depend on the data stored in the smart contract (e.g. the `openBet` function cannot be called when the balance stored in the state of the smart contract is negative), keeping these parts of the state on the proxy and using delegate calls may be required.

In order to instrument the specification monitor, we can use ContractLarva on the interface contract and the specification to obtain a safely encapsulated behavioural interface as shown in Figure 6. This will be able to identify any violation at runtime, ensuring we can react accordingly as discussed in the previous sections. In this manner, trust — despite versioning — can be addressed through an immutable behavioural interface, although it remains a major challenge to have a sufficiently detailed behavioural interface which disallows *all* undesirable behaviour.

This approach borrows much from behavioural interfaces, in that we automatically create a safe, trusted and immutable interface which accesses an untrusted backend and mutable implementation. In a way, the approach also borrows from monitoring-oriented programming [4] in that we are programming the safe interface using monitoring techniques.

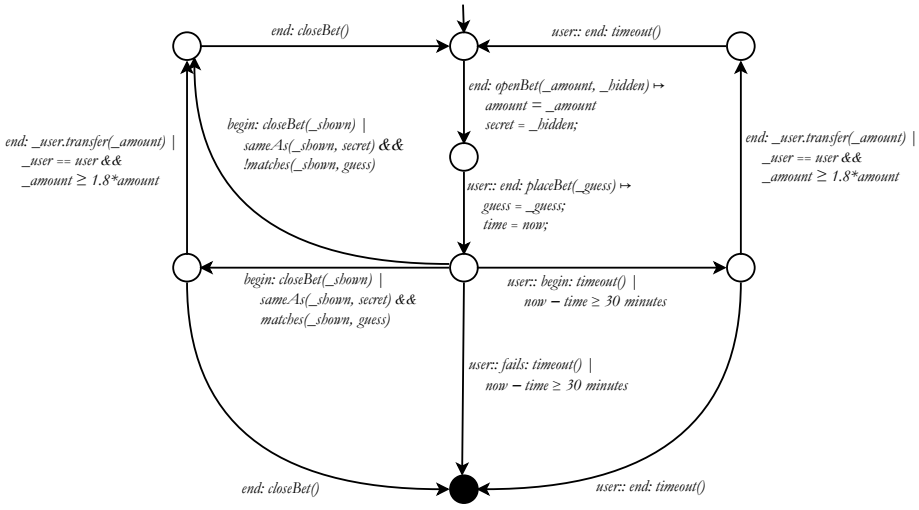

Fig. 5. User-centric casino specification

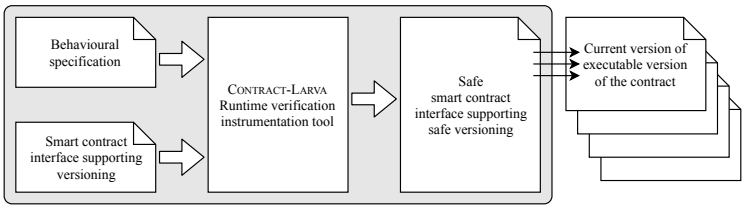

Fig. 6. Building safe behavioural interfaces for smart contracts

5 Conclusions

In this paper we have examined a spectrum of dynamic analysis techniques for making smart contracts safer and more dependable. Although at the surface level smart contracts appear to be normal software, however, there are a number of issues which result in standard runtime techniques to have to be adapted in order to be useful in this context. Clearly, the domain makes static, compile-time analysis even more attractive (or desirable) than for standard systems. However, the sparse literature applying such techniques for smart contracts e.g. [3, 22], particularly for business-logic specifications indicates that, at least for the time being, we have to depend on the lower hanging fruit dynamic analysis provides. This brings its own challenges — perhaps most pertinent is that of recovery from violations discovered at runtime. In the domain of runtime verification of general systems, the notion of *healing* has recently been explored in [12], in which the authors classify the solutions into three similar classes as found in our proposal: rollback, preventing further failures and compensation. How these can be adapted for smart contracts is, however, the challenge we have addressed in this paper.

By enriching smart contract programming languages with notions such as checkpointing and compensations, we believe that one could alleviate handling of such violations. Another major challenge is that of the immutability of smart contracts, and the solution we are proposing in order to ensure that the system works correctly but still allow the implementation to be modified follows the conclusions of other work [20], which argued for declarative as opposed to imperative and operational approaches currently used on DLTs such as Ethereum.

We are currently looking at identifying means of deploying many of the ideas presented in this paper on real-world systems. Our tool CONTRACTLARVA has already been applied on a number of smart contracts in order to deploy runtime verification and recovery, but there are still various challenges left to be addressed. It can be argued that our solution to resolve the immutability of smart contracts by making them mutable while ensuring immutability of specifications is nothing but pushing the problem one level up. However, we believe that moving one level of abstraction up, ignoring most implementation details results in lower risk of error. Furthermore, one can consider other solutions currently at the implementation level to support versioning of specifications (e.g. allowing for a specification to be updated by consensus or a majority vote). It will be interesting to see how such an approach would fare on large real-world smart contracts.

References

1. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.
2. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, New York, NY, USA, 2016. ACM.
3. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts. In *The 11th Workshop on Programming Languages and Analysis for Security (PLAS'16)*, 2016.
4. Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.*, 89(2):108–127, 2003.
5. Feng Chen and Grigore Rosu. Java-mop: A monitoring oriented programming environment for java. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 546–550, 2005.
6. Christian Colombo and Gordon J. Pace. Recovery within long-running transactions. *ACM Comput. Surv.*, 45(3):28:1–28:35, 2013.
7. Christian Colombo and Gordon J. Pace. Comprehensive monitor-oriented compensation programming. In *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2014, Grenoble, France, 12th April 2014.*, pages 47–61, 2014.

8. Christian Colombo, Gordon J. Pace, and Patrick Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
9. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, 2009.
10. Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography and Data Security 2016*, pages 79–94, 2016.
11. Joshua Ellul and Gordon J. Pace. Runtime verification of ethereum smart contracts. In *Workshop on Blockchain Dependability (WBD), in conjunction with 14th European Dependable Computing Conference (EDCC)*, 2018.
12. Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, pages 103–134. 2018.
13. Michael Fröwis and Rainer Böhme. In code we trust? — measuring the control flow immutability of all smart contracts deployed on ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2017*, pages 357–372, 2017.
14. Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.
15. Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *IEEE Data Eng. Bull.*, 14(1):14–18, 1991.
16. Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *POST*, volume 10804 of *Lecture Notes in Computer Science*, pages 243–269. Springer, 2018.
17. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.
18. Ian Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, November 1989.
19. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017.
20. Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *Rule Technologies. Research, Tools, and Applications - 10th International Symposium, RuleML 2016, Stony Brook, NY, USA, July 6-9, 2016. Proceedings*, pages 167–183, 2016.
21. Lawrence Lessig. *Code 2.0*. CreateSpace, Paramount, CA, 2nd edition, 2009.
22. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269, 2016.
23. Bertrand Meyer. Design by contract: The eiffel method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.
24. Vaughan R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 97–107, 1995.
25. Brian Randell, P. A. Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.
26. Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security 2017*, pages 478–493, 2017.
27. Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, (16), 1996.
28. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.