


Verifying Smart Contract Transformations Using Bisimulations

Kegan McIlwaine ✉ 

University of Wyoming, Laramie, WY, USA

James Caldwell ✉ 

University of Wyoming, Laramie, WY, USA

Abstract

Determining whether two computational artifacts share the same behavior is fundamental. In general, smart contracts and their interactions can be modeled as the concurrent composition of processes where: (i.) contracts are processes, (ii.) parties to the contract are processes, and (iii.) the blockchain itself is a process. In this paper we describe how we apply this view of smart contracts to the verification of an optimizing transformation in the Faustus smart contract programming language. Faustus compiles to the embedded domain specific language (eDSL) Marlowe. With Marlowe as the target compilation language, the operators and semantics of Milner's value passing Calculus of Communicating Systems (CCS) inspired the design of Faustus. In CCS, unobservable transitions (τ -transitions) arise from the parallel composition of processes that share a label and a co-label (e.g. a and \bar{a}). CCS also supports a *restriction* operator ($P \setminus \mathcal{A}$) which internalizes, within P , the labels in the set \mathcal{A} . From an observer's point of view, any number of τ -transitions followed by an observable transition, say a , looks like a single transition on a . In Faustus, similarly, unobservable actions arise by adding actions to be internalized to a set \mathcal{A} . A proof that two Faustus contracts are *weakly bisimilar* ($P \approx Q$) verifies that, with respect to their observable executions, they exhibit identical behaviors. This paper describes an application of observational equivalence, witnessed by the existence of a weak bisimulation relation, to verify that a smart contract and its transformed instance preserves observable behavior. More precisely, if $\lceil P \rceil$ is the result of applying our transformation to a contract P , we prove $\forall P. P \approx \lceil P \rceil$. The smart contract transformation verified here trades time for space in smart contracts running on the Cardano blockchain. The results of this paper have been formalized in the Isabelle theorem prover, and we have formalized the small-step semantics of Faustus contracts together with the labeled transition system induced by those semantics.

2012 ACM Subject Classification Software and its engineering \rightarrow Formal methods

Keywords and phrases Smart Contracts, Bisimulation, Program Transformation

Digital Object Identifier 10.4230/OASICS.FMBC.2025.9

Supplementary Material *Software (Source Code)*: <https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2> [3], archived at `swb:1:dir:ba99bcd12d63cd2ece88fd95325e5032fcdd5637`

Funding The research presented in this paper was supported by a grant from IOG Singapore Pte. Ltd.

Acknowledgements The research presented in this paper was supported by a grant from IOG Singapore Pte. Ltd., and thanks especially to Charles Hoskinson.

1 Introduction

Smart contracts, initially proposed by Szabo [43], are self-enforcing, self-executing protocols governing interactions between several (potentially distrusting) parties. Smart contracts automate the execution of an agreement between participants. Trust is established between the participants by the use of a blockchain, which is an unforgeable distributed ledger where transactions are added in a cryptographically secure manner.



© Kegan McIlwaine and James Caldwell;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmsoler and Meng Xu; Article No. 9; pp. 9:1–9:19

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Because blockchains are massive distributed systems, the cost of running and maintaining the blockchain must be paid in some way. Typically, this cost is a transaction fee, colloquially referred to as “gas”. The cost of running a smart contracts has led to a culture where programmers apply optimizations to minimize gas costs [17, 37, 4, 14]. These optimizations may introduce errors not in the original contract; there are no guarantees that they preserve the semantics of the original smart contract.

Reports of smart contracts being hacked are legion, and the losses from hacked contracts can be astronomically large [7, 23, 2]. The potential for, and the magnitude of, financial losses is a motivation for applying the most rigorous correctness guarantees to smart contracts, *i.e.* formal verification. The work reported here was funded by Input Output Global (IOG, formerly IOHK), the developers of the Cardano Blockchain¹. The IOG development methodology is based on application of *agile* formal methods for specification and verification of the system [22, 16] and the work reported here follows that methodology.

Faustus, the smart contract language used here, compiles to the Marlowe smart contract language [26, 24] developed at IOG. Prior to our work, the Marlowe developers formalized the evaluation semantics of Marlowe in Isabelle/HOL [38]. That formalization served as the basis for a verification of the correctness of the Faustus (to Marlowe) compiler [29, 30] and for the further developments described in this paper.

1.1 Related Works

Formal verification has previously been used to prove individual smart contracts satisfy some properties. A recent survey of these efforts can be found in Tolmach et al. [44]. These techniques verify the correctness of a smart contract with respect to a formal specification. For example, tools have been developed to verify unannotated common smart contracts against their canonical forms [13], automated tools have been developed to verify liquidity in BitML [11] and Solidity contracts [8, 12, 9], and verified super-optimization techniques have been applied to Ethereum smart contracts [4].

Nelaturu et al. [37] have developed a methodology for optimizing the cost of smart contracts, and synthesize bisimulation relations to verify the correctness of the optimizations. Their techniques are limited by the constraints on the inputs to the automated tools they use in their system, *e.g.* the programs must be deterministic and the kinds of loops their system can operate on are restricted.

Similar to the work described in this paper, Sergey and Hobor [42] aim to provide an analogy between smart contracts and concurrent processes. Differing from previous works of low-level verification, Sergey and Hobor argue that smart contract programmers and verifiers will work more efficiently and be less error prone by adopting the perspective of smart contracts as concurrent processes.

Building on the perspective of viewing smart contracts as concurrent processes, Qu et al. [41] presented a method for modeling and verifying smart contracts using Communicating Sequential Processes (CSP) [21]. Their technique is used to verify that a smart contract is safe against particular attacks. In their methodology, smart contracts that are written in Solidity are translated into CSP. This step introduces a gap between the actual code and the model, as they do not verify the correctness of the translation.

Additionally, a lot of work has already been done in the field of developing programming languages for specifying and implementing financial contracts. Peyton Jones et al. [40] developed a declarative language for specifying executable financial contracts in Haskell and

¹ <https://iohk.io/>

proved that two syntactically different contracts are equivalent using denotational semantics. Interestingly, this paper is one of the few that focuses on proving the equivalence of contracts. Andersen et al. [6] expanded on that work with their Contract Specification Language (CSL). Then Henglein et al. [19] provided a static analysis framework for verifying properties of CSL contracts, such as fairness and party participation. In a separate piece of work, Seijas and Thompson [26] developed Marlowe, a Domain Specific Language (DSL) for writing financial contracts on the Cardano blockchain. All Marlowe contracts are total - they are all guaranteed to terminate. They used the Isabelle/HOL theorem prover to prove that all Marlowe contracts terminate and pay out any locked funds. Seijas et al. [24, 25] also developed a static analysis tool that can verify all possible executions of a Marlowe contract never result in an error.

Another related field of work is the development of smart contract languages based on process algebra. Rholang, based on the ρ -calculus, was developed as the native smart contract language for the RChain blockchain [18]. Then Bartoletti et al. [11] developed the process calculus BitML as a DSL for writing Bitcoin smart contracts. Rholang executes natively on the RChain blockchain, while BitML is compiled to Bitcoin transactions. Finally, another process calculus, ILLUM, was recently developed by Bartoletti et al. [10] as an intermediate language for compiling higher level languages into transactions on the Bitcoin and Cardano blockchains.

1.2 Our Contribution

This paper describes the Faustus smart contract language and a methodology for verifying equivalence of Faustus contracts based on bisimulation [39, 34] from the perspective of parties interacting with the contract. Our work differs from the previous works in the following ways:

1. We have developed a methodology based on Milner's Calculus of Communicating Systems (CCS) [33, 34, 35] to prove the *observational equivalence* of contracts, *i.e.* to prove that Faustus contracts are semantically equivalent from the perspective of the parties interacting with the contract. Other work that we are aware of is: limited by the types of contracts that can be verified; verifies properties of individual contracts; or develops tools to analyze safety and liveness properties of specified contracts.
2. In Faustus, reasoning using CCS based techniques is done directly on well-formed code of the smart contract. There is no translation from the smart contract code to a separate model for verification. In many methodologies the first step is an unverified translation from a contract/program to a model suitable for formal analysis.
3. In our formal model, Faustus processes include a set of actions that, when executed, are hidden from users. Following Milner [34], we call this collection a restriction set. The behavior of a Faustus process with a nonempty restriction set corresponds to a parallel composition of processes. In the process defined by the parallel composition, actions in restriction set are unobservable.
4. Our methodology for verifying two processes are equivalent is to prove the two are observationally equivalent by exhibiting the appropriate bisimulation between the two. We describe a program (contract) transformation, and apply this methodology to formally prove that an arbitrary contract and the resulting transformed contract are observationally equivalent.
5. The work described here has been formalized in the Isabelle/HOL interactive theorem prover. We provide proofs for theorems and lemmas in this paper where possible, and give a more informal outline if the proof relies on theorems outside of the scope of this work.

1.3 Paper Organization

This paper briefly introduces the Faustus smart contract language, and then shows how to interpret Faustus contracts as CCS-style processes based on the transition relation given by the operational semantics of Faustus. Finally, we apply the methodology to verify a nontrivial transformation on Faustus contracts.

2 Introduction to Faustus

Faustus is based on an extension of Marlowe’s financial contract operators that makes it into a (more) fully featured programming language. The formal interpretation of a Faustus program is based on its operational semantics which have been formalized Isabelle/HOL [3]. Faustus contracts describe allowable interactions between contracts and the parties to the contract. Aside from blockchain addresses, which serve as parties, the details of the interaction between the contract and the blockchain are hidden from the Faustus programmer.

It is quite natural to use the operational semantics to define a standard kind of automata, a labeled transition system, where states of the automata are contract states, and labels are the interactions. Milner’s CCS [34] is defined over labeled transition systems. CCS processes are constructed with guarded commands, a choice operator, and parallel composition. We have implemented these operators in the Faustus language of guarded commands². Because our execution semantics prioritize certain transitions, our target is a variant of CCS with priority [27].

There are six basic things to note about the execution of Faustus contracts:

1. Faustus contracts run as a series of sequential statements that pause when waiting for external input (from an agent or the blockchain) or a specific timeout to occur.
2. Hidden (τ -actions) arise in contracts by the evaluation of: variable declarations; assignment statements; if-statements; close commands; assert statements; and pay statements. (See Section 3.1.)
3. The **when** contract denotes actions that parties are allowed to take within the contract, and a timeout contract in the case that no action is received before the specified time. **when** contracts have the form **when** $\{g_1 \rightarrow c_1, \dots, g_n \rightarrow c_n\}$ **after** $t \rightarrow c_t$. The list of guarded commands is tested in order until a guard evaluates to true in the current state with the current action. If no guard evaluates to true and the time $t < s$, where s is the start of the current time window, then the contract c_t is evaluated.
4. Faustus contracts contain internal accounts for each party. Currencies are moved between internal accounts and paid out to parties’ blockchain wallets according to the contract logic.
5. All Faustus contracts end with a **close** command or a contract variable. Recursion is not allowed to ensure termination. The **close** command ensures no funds are locked in the closed Faustus contract by paying out all internal account balances.
6. Variables can be bound for any syntactical element.

We can illustrate more interesting features of Faustus through the three-party escrow contract in Figure 1. At the beginning, two actions are declared corresponding to **yes** and **no** votes for moving the money along. Then a parameterized contract is declared, called

² Interestingly, we initially realized we could use choice and interleaving to compactly describe common combinatorial patterns used in many Marlowe programs. This led us to consider the more full-bodied implementation of CCS which this paper is based on.

```

action yes(party p) = p chooses "agree";
action no(party p) = p chooses not "agree";
contract escrow(party alice, party bob, party escrow, time start_time) {
  contract payout(int votes) {
    if votes >= 2
      then escrow pays 1000 ada to bob; close
    else escrow pays 1000 ada to alice; close
  };
  when {
    alice deposits 1000 ada into escrow -> {
      int votes = 0;
      when {
        (yes(alice) -> votes := votes + 1 <+> no(alice))
        <|> (yes(bob) -> votes := votes + 1 <+> no(bob))
        <|> (yes(escrow) -> votes := votes + 1 <+> no(escrow)) -> {
          payout(votes)
        }
      }
    } after start_time + (90 days) -> payout(votes)
  }
} after start_time + (3 days) -> close
};//  alice          bob          escrow          start_time
escrow(addr1q8zg...48g7, addr1qygn...9pl5, addr1qx2q...xlw7, 2025-05-04 00:00.00)

```

■ **Figure 1** Three party escrow contract in Faustus.

`three_party_escrow`, which takes wallet addresses for the participants and the time the contract will start. Inside `three_party_escrow` another `payout` contract is declared. Finally, a `when` clause allows defining the interactions between the participants and the contract.

First, Alice is expected to deposit 1000 ada into the escrow account. After the deposit, the 1000 ada will reside in an account that is internal to the contract for the escrow party. The contract logic will not allow the funds to be used until the `payout` contract is called.

Each participant is then allowed to take either a `yes` or `no` vote. The choice operator is `<+>`, and $P<+>Q$ only evaluates P or Q , but not both. Note that the `yes` action also increments `votes`. Then, those choices are allowed to be received in any order using the interleaving operator `<|>`³. Note that until the final continuation contract is written, guards may only be followed by variable reassignments, payments, or other guards. Both the `<+>` and `<|>` operators prioritize executing the left operand.

After all of the votes are received, or the contract times out, the `payout` contract is called. The `payout` contract checks if the votes meet a certain threshold to send the money to Bob. Otherwise, it sends the money back to Alice. In each case, the contract is then fully closed.

3 Equivalent Contracts

A central question that arises when thinking about computational systems is, when do two systems have equivalent behavior? Quoting Milner [36], “*Until we know what constitutes similarity or difference of behavior we cannot claim to know what ‘behavior’ means - and if that is the case then we have no precise way of explaining what our systems do.*”

³ In general, if there are k interleaved guarded commands there are $k!$ (factorial) possible execution paths induced by them.

Thinking from first principles to answer what it might mean for a pair of processes, P and Q , to be equivalent⁴, if we are very clever, we might come up with something like:

Processes P and Q are observationally equivalent if for every sequence of actions (say s), applying s to P results in state P' , then there is a state Q' that results from applying s to Q and P' and Q' are themselves equivalent; and vice versa.

This idea has served as the basis of a whole host of formalisms for reasoning about processes described as labeled transition systems.

It is convenient to use some notation for this relation. We write $P \approx Q$ to mean that P and Q are observationally equivalent. To make the idea of a sequence of actions (say s) being *applied* to a process P yielding P' , we write $P \xRightarrow{s} P'$. With this notation we can make the definition more precise as follows:

$P \approx Q$ if, and only if, for all sequences of actions s

- i.) Whenever $P \xRightarrow{s} P'$ then, for some Q' , $Q \xRightarrow{s} Q'$ and $P' \approx Q'$.
- ii.) Whenever $Q \xRightarrow{s} Q'$ then, for some P' , $P \xRightarrow{s} P'$ and $P' \approx Q'$.

In the rest of this section, we describe the formalization of this idea in a form suitable for proving equivalence of Faustus processes.

3.1 Faustus Processes

The operational semantics of Faustus is formalized in Isabelle/HOL [3]. The small step semantics of Faustus are given as an inductive relation over configurations which consist of a Faustus contract together with the Faustus state. The small step semantics define an evaluator for Faustus. Configurations correspond to CCS processes. Transitions in the small step semantics are either observable, and labeled with an action; or unobservable, and labeled with a τ . Additionally, Faustus processes contain a set of actions that become restricted and unobservable to the user. This formalization serves as the basis of our development of observational equivalence.

► **Definition 1** (Blockchain Primitives). ***String** is the set of all strings. **Party** is the set of all blockchain addresses. **Curr** is the set of all blockchain currencies or kinds of tokens.*

► **Definition 2** (Labeled Transition System). *A labeled transition system (LTS) is a triple $\langle \mathcal{S}, \mathcal{L}, \Sigma \rangle$ where \mathcal{S} is a set of states, \mathcal{L} is a set of labels, and $\Sigma \subseteq (\mathcal{S} \times \mathcal{L} \times \mathcal{S})$ is a transition relation (see e.g. [34]).*

We now provide definitions with the end result of defining an LTS for Faustus processes. The states of the Faustus LTS will be the contract, its execution state, and a restriction set. The transition relation is defined by the operational semantics of Faustus. The labels will be actions paired with a time window when they are generated.

► **Definition 3** (Faustus Execution *State*).

$$\text{State} = \langle \text{Context}, \text{Env}, \text{Acct}, \text{Choice}, (\mathbb{Z}, \mathbb{Z}), [\text{Payment}], [\text{Assertion}] \rangle$$

Where **Context** is a typing context (a map of variable names to types); **Env** is an environment (a map of variable names to their meanings); **Acct** are account balances (a map of elements of $(\text{Party}, \text{Curr})$ pairs to integers); **Choice** are the choices that have been

⁴ This is exactly what Milner [31, 32], Hoare [20], and Park [39] did.

made (a map of choice name and party pairs to integers); the **start** and **end** times of the current transaction window are given by a pair of integers interpreted as *POSIX Times*; [**Payment**] is the history of payments; and [**Assertion**] is the history of failed assertions. We use variables $\{\sigma, \sigma_1, \sigma_2, \dots\}$ to denote arbitrary execution states.

In practice, we use a small step semantics to define the transition relation used to evaluate Faustus processes (see Definition 8). The small step semantics manage the actions in the restriction set to make them unobservable to parties observing the progress of a running contract.

There are three ways to interact with a contract running on the blockchain. They are the choice, deposit, and notify actions.

A choice action allows a party to provide an integer value for a specific choice, labeled by a string. For each choice action $\langle \text{choice}, p, c, x \rangle$ encountered during the evaluation of a contract, the choice-party pair is mapped to the integer value x in the execution state (see Definition 3). This data can be referred to later, both as part of the logic of the contract itself and by other parties, to guide future behavior. The party and choice in the user action must match a party and choice specified in a *choice guard* of the current **when**. In a choice guard, choice values are specified to be in a list of range values, $[[y_1, z_1], \dots [y_k, z_k]]$. A guard is triggered if, for the particular choice value x , if $y_i \leq x \leq z_i$ for some $i \in \{1..k\}$.

A deposit action $\langle \text{deposit}, p_1, x, c, p_2 \rangle$ specifies that party p_1 (recall, strictly speaking, parties are addresses on the blockchain⁵) deposits x tokens of currency c into party p_2 's account in the contract. During the execution of a contract, the contract itself holds the deposited funds. These funds may be paid back out of the contract to a party (an address on the blockchain) as specified by the logic of the contract. Deposit actions of the form $\langle \text{deposit}, p_1, x, c, p_2 \rangle$ are only successful if the parties, amount of the deposit and the currency match the deposit guard exactly. Funds internal to a contract are tracked in the **Acct** map of the execution state.

The notify action allows parties to *notify* a contract that it can proceed with execution. Notify guards are of the form $\text{notify } b \rightarrow \{C\}$ where b is a Boolean expression and C is the next contract to evaluate if the expression b evaluates to **true** in the current execution state. For a notify action to trigger a notify, the expression b must evaluate to true.

► **Definition 4** (User Actions). *There are three types of user actions; choice, deposit, and notify. We define the sets **Choice**, the set of all choice actions; **Dep**, the set of all deposit actions; and **Notif**, the set of all notify actions as follows:*

$$\begin{aligned} \mathbf{Choice} &\stackrel{\text{def}}{=} \{\langle \text{choice}, p, c, x \rangle \mid p \in \mathbf{Party}, c \in \mathbf{String}, x \in \mathbb{Z}\} \\ \mathbf{Dep} &\stackrel{\text{def}}{=} \{\langle \text{deposit}, p_1, x, c, p_2 \rangle \mid p_1 \in \mathbf{Party}, x \in \mathbb{Z}, c \in \mathbf{Curr}, p_2 \in \mathbf{Party}\} \\ \mathbf{Notif} &\stackrel{\text{def}}{=} \{\text{notify}\} \end{aligned}$$

We define $\mathbf{Act} \stackrel{\text{def}}{=} \mathbf{Choice} \cup \mathbf{Dep} \cup \mathbf{Notif}$.

Faustus processes are triples containing the code of the contract that is executing, the state of the executing contract, and a *restriction set*, a collection of actions that are *unobservable*⁶.

⁵ Cardano is based on an extended UTXO (Unspent Transaction Outputs) [15] model. Faustus uses an account based model built on top of the EUTXO model of the Cardano blockchain.

⁶ For readers familiar with Milner and Hoare style process algebras this is similar to restriction.

► **Definition 5** (Faustus Processes). Let *Contract* be the set of all well-formed Faustus contracts. Then we define the set of Faustus processes, *Proc*, to be the set of all pairs of Faustus contracts and execution states restricted by an action set *A*:

$$\mathbf{Proc} \stackrel{\text{def}}{=} \{ \langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \mid \mathcal{C} \in \mathbf{Contract}, \sigma \in \mathbf{State}, \mathcal{A} \subseteq \mathbf{Act} \}$$

Note that we write $\langle \mathcal{C}, \sigma \rangle$ when \mathcal{A} is empty instead of $\langle \mathcal{C}, \sigma \rangle \backslash \{\}$, also, we write P (or P_1, P', Q, \dots etc.) to denote arbitrary elements of $\mathbf{Contract} \times \mathbf{State}$.

Full transition labels include an action and a time window indicated by a start t_1 and end time t_2 , we use $\# = \langle t_1, t_2 \rangle$ to denote an arbitrary time window $\langle t_1, t_2 \rangle$. Due to the latency between a transaction being submitted and evaluated on the blockchain, actions are expected to be processed within the provided time window $\#$, and are rejected otherwise. The times in $\#$ are generated externally and provided in the stream of input to a running process. Typically, the time window provided allows more than enough time to be evaluated by the blockchain. These times are used to update the start and end time components of the *State* according to the Faustus semantics.

► **Definition 6** (Labels). When a Faustus process makes a transition, that transition is labeled to indicate the external/user action, a , and a time window, $\#$, during which the transaction containing the action is processed. The ϵ action allows a transition where time changes without an explicit user action. The τ label indicates an unobservable transition; a transition that does not require an external action. We define the set of all labels, *Label*, as follows:

$$\mathbf{Label} \stackrel{\text{def}}{=} \{ \langle a, \# \rangle \mid a \in \mathbf{Act} \cup \{\epsilon\}, \# \in \mathbb{Z} \times \mathbb{Z} \} \cup \{\tau\}$$

► **Definition 7** (Composition of Transition Relations). If $\mathcal{R} \subseteq A \times C$ and $\mathcal{S} \subseteq C \times B$ we write $\mathcal{R} \cdot \mathcal{S}$ to denote the relation on $A \times B$ constructed by (ordinary) composition of relations.

$$\mathcal{R} \cdot \mathcal{S} \stackrel{\text{def}}{=} \{ \langle x, y \rangle \in (A \times B) \mid \exists z \in C. x \mathcal{R} z \wedge z \mathcal{S} y \}$$

► **Definition 8** (Faustus Transition Relation). We write $\langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle \backslash \mathcal{A}'$ to denote transitions restricted by \mathcal{A} where $\mathcal{A}, \mathcal{A}' \subseteq \mathbf{Act}$. Under the small step semantics of Faustus⁷, contract \mathcal{C} in state σ with restricted actions \mathcal{A} transitions on input (label) λ to contract \mathcal{C}' , with state σ' , and restriction set \mathcal{A}' . Labels $\langle a, \# \rangle \in \mathbf{Act} \times \mathbb{Z}^2$ are hidden in the sense that transitions made on labels containing actions in \mathcal{A} are not observable⁸. When $\mathcal{A} = \mathcal{A}'$ (and it always will be in this paper) we write $\langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle \backslash \mathcal{A}$. In the case where $P \backslash \mathcal{A} \xrightarrow{\lambda_1} R_1 \backslash \mathcal{A}$, $R_1 \backslash \mathcal{A} \xrightarrow{\lambda_2} R_2 \backslash \mathcal{A}$, and so on, up to $R_{k-1} \backslash \mathcal{A} \xrightarrow{\lambda_k} Q \backslash \mathcal{A}$, we write $P \backslash \mathcal{A} \xrightarrow{\lambda_1 \dots \lambda_k} Q \backslash \mathcal{A}$ (instead of $P \backslash \mathcal{A} (\xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_k}) Q \backslash \mathcal{A}$) to denote a transition in the relation formed by the composition of $\xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_k}$. In the case where there is a vector of actions, $\vec{\alpha} = \langle \alpha_1, \dots, \alpha_k \rangle$, occurring sequentially with the same time window, we write $P \backslash \mathcal{A} \xrightarrow{\langle \vec{\alpha}, \# \rangle} Q \backslash \mathcal{A}$ instead of $P \backslash \mathcal{A} \xrightarrow{\langle \alpha_1, \# \rangle \dots \langle \alpha_k, \# \rangle} Q \backslash \mathcal{A}$. The full transition relation is defined by the following equations:

- 1.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_0 Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \langle b, \# \rangle, Q \backslash \mathcal{A} \rangle \mid b \notin \mathcal{A} \wedge P \xrightarrow{\lambda} Q \}$
- 2.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_k Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \langle b, \# \rangle, Q \backslash \mathcal{A} \rangle \mid k > 0 \wedge b \neq \epsilon \wedge b \notin \mathcal{A} \wedge \exists \vec{\alpha} \in \mathcal{A}^k. P \xrightarrow{\langle \vec{\alpha}, b, \# \rangle} Q \}$
- 3.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle} Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}} \{ P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_k Q \backslash \mathcal{A} \}$
- 4.) $P \backslash \mathcal{A} \xrightarrow{\tau} Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \tau, Q \backslash \mathcal{A} \rangle \mid P \xrightarrow{\tau} Q \}$

⁷ Formally, in the Faustus semantics, $\langle \mathcal{C}, \sigma \rangle \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle$ is defined by $\llbracket \Gamma, \mathcal{A} \vdash \mathcal{C} : \text{contract} \rrbracket \sigma \lambda = \langle \mathcal{C}', \sigma' \rangle$.

⁸ In CCS [34] unobservable τ -transitions arise by the synchronization of a name (a label) say λ and its co-name $\bar{\lambda}$.

Note that $\xrightarrow{\lambda}^*$ denotes the reflexive transitive closure of $\xrightarrow{\lambda}$, i.e. it is the relation defined by the composition of any number of $\xrightarrow{\lambda}$ relations including 0.

The Faustus transition relation allows an arbitrary number of hidden actions from the restriction set to occur before an observable transition. In this way, the restriction set not only prevents the users from sending actions in the set, but it also corresponds to a process running concurrently with the Faustus smart contract that can send any of the actions in the restriction set. Regular τ -transitions and timeout transitions remain unchanged by the restriction set.

► **Definition 9** (Faustus Labeled Transition System). *The Faustus LTS is defined as the triple $\langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$. The states of the Faustus transition relation are elements of \mathbf{Proc} . The labels of the Faustus transition system are elements of \mathbf{Label} . The transition relation \rightarrow is defined in Definition 8, and given by the small step semantics of Faustus; the details of which have been formalized in Isabelle/HOL [3].*

Typically, users interact with contracts using an empty set of restricted actions. Non-empty sets of restricted actions allow for traversing different contract structures through multiple transactions without requiring additional user interaction. This will be explained in more detail in Section 4.

With the definition of the Faustus transition relation, we also define the experiment relation between Faustus processes on sequences of labels.

► **Definition 10** (Experiment Relation). *Let $s = \lambda_1 \cdots \lambda_n \in \mathbf{Label}^*$. We define the experiment relation \xRightarrow{s} as follows:*

$$\begin{aligned} \xRightarrow{s} &\stackrel{\text{def}}{=} \xrightarrow{\tau}^* \\ \xRightarrow{s} &\stackrel{\text{def}}{=} \Rightarrow \cdot \xrightarrow{\lambda_1} \cdot \Rightarrow \cdots \Rightarrow \cdot \xrightarrow{\lambda_n} \cdot \Rightarrow \end{aligned}$$

Thus, \xRightarrow{s} is the behavior that allows an arbitrary number of τ actions before, between, and after the observable actions in s .

3.2 Observational Equivalence

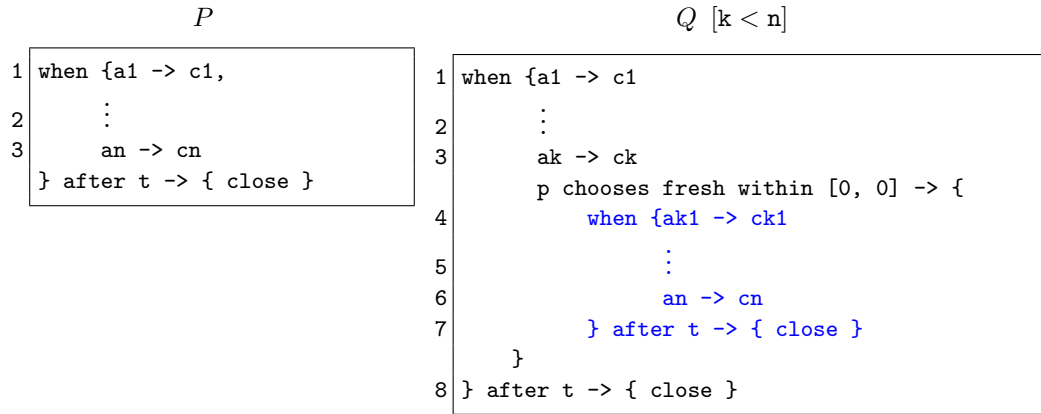
A *bisimulation* is a binary relation between states of processes. It identifies states that cannot be distinguished from one another by any sequence of observable actions. From the perspective of a user interacting with two bisimilar processes, the same inputs result in the same outputs for both processes. The user cannot tell the difference between the two processes by interacting with them. If two processes, P and Q , are bisimilar, we write $P \approx Q$. It turns out that \approx is an equivalence relation [35].

► **Definition 11** (Weak Simulation). *Let $\mathcal{F} = \langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$ be the Faustus LTS, and let $\mathcal{R} \subseteq (\mathbf{Proc} \times \mathbf{Proc})$ be a binary relation, and $s \in \mathbf{Label}^*$. \mathcal{R} is a weak simulation over \mathcal{F} when the following property holds:*

If $(P_1, Q_1) \in \mathcal{R}$ and $P_1 \xRightarrow{s} P_2$ then there exists $Q_2 \in \mathcal{S}$ such that $Q_1 \xRightarrow{s} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$

This definition leads to the definitions of weak bisimulation and observational equivalence.

► **Definition 12** (Weak Bisimulation, Observational Equivalence). *Let $\mathcal{F} = \langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$ be the Faustus LTS, and let $\mathcal{R} \subseteq (\mathbf{Proc} \times \mathbf{Proc})$ be a binary relation. Then \mathcal{R} is a weak bisimulation over \mathcal{F} , if it and its converse are both weak simulations. We say P and Q are observationally equivalent, written $P \approx Q$, if there exists a weak bisimulation \mathcal{R} such that $(P, Q) \in \mathcal{R}$.*



■ **Figure 2** A contract (P) and the result (Q) of recursively applying the transformation.

Processes are called observationally equivalent when there is a bisimulation between them. The number of unobservable transitions may differ when evaluating two observationally equivalent processes on the same observable transitions. We have proved in Isabelle/HOL that bisimulations on Faustus processes are equivalence relations, and that simulations are closed under union. The proofs of these properties are well understood, and can be found in multiple works by Milner [33, 34, 35, 36].

With the definition of observational equivalence, we can now prove transformations of Faustus contracts do not modify their observed behaviors. The next section describes a common transformation that is applied to Faustus contracts, and proves the result of the transformation is observationally equivalent to the original.

4 Contract Transformations

In each round of computation on the blockchain, Faustus programs run until they pause for external input at a `when` contract, or are closed. The computation time between each pause may take longer than the maximum computation time allowed per transaction on the blockchain, leading to out of gas errors [1]. If an out of gas error occurs, the contract will revert back to its state prior to that round of computation. Reverting to the prior state can result in code that never progresses while still generating gas costs.

Consider the contract on the left side of Figure 2. Assume that in each round of computation there is only enough time to check $k + 1$ guards before an out of gas error occurs. Then there are $n - (k + 1)$ guard continuations that are unreachable when running the contract on the blockchain. On the right side of Figure 2, the sequence of n guards has been replaced by a length $k + 1$ sequence of guards, the last of which is a fresh choice guard that cascades to a `when` containing the rest of the transformed contract. We have colored this section blue to indicate it will be recursively transformed.

To allow every guarded command to be reachable, programmers apply this transformation where the list of guarded commands is truncated, and moved into a sub-contract that runs after a “fresh” choice action is taken. The key aspect is selecting a choice name that does not appear in the original contract, *i.e.* it is fresh, hence $(p \text{ chooses } \textit{fresh} \text{ within } [0, 0])^9$ is a choice guard that does not occur in the original contract. The result of the transformation

⁹ The range $[0, 0]$ simply means p must choose 0.

```

(* sg - split guards *)
function sg :: "ChoiceName ⇒ FContract ⇒ FContract" where
"sg cn (When guards t cont) = (
  let
    newGuards = map (λ(Case g c) ⇒ (Case g (sg cn c))) guards;
    newCont = sg cn cont
  in
    if (k > 1 ∧ length newGuards > k)
    then (When (take k newGuards @
      [Case (ActionGuard (Choice (FChoiceId cn p) [(0, 0)]))
        (sg cn (When (drop k guards) t cont))]))
      t newCont)
    else (When newGuards t newCont))" |
"sg cn (Close) = Close" |
"sg cn (StatementCont s c) = (StatementCont s (sg cn c))" |
"sg cn (If e c1 c2) = (If e (sg cn c1) (sg cn c2))" |
"sg cn (Let i v c) = (Let i v (sg cn c))" |
"sg cn (LetObservation i v c) = (LetObservation i v (sg cn c))" |
"sg cn (LetPubKey i v c) = (LetPubKey i v (sg cn c))" |
"sg cn (LetC i p b c) = (LetC i p (sg cn b) (sg cn c))" |
"sg cn (UseC i a) = (UseC i a)"

```

■ **Figure 3** Isabelle implementation of the transformation algorithm.

applied to P in Figure 2 is Q in the same figure. As we will see below, the “fresh” choice allows for a transition that leads directly to a **when** contract. Since **when** contracts pause execution until the next round of blockchain computation, the out of gas error is avoided, and the rest of the guard continuations can be reached in the next round of computation.

This type of transformation has been informally described by Bush [14] as a best practice for Marlowe contracts with complex logic. The Isabelle function in Figure 3, which acts on the Faustus abstract syntax tree from Appendix A, formally describes the transformation.

► **Definition 13** (Transformation Constants). *The transformation function in Figure 3 has two constants associated with it, p and k . The first one, p , is an arbitrary **Party** that will be allowed to send signals containing the $\langle \text{choice}, p, c, 0 \rangle$ action that traverses the cascading **when** contracts; and k is a natural number representing the maximum number of guarded commands before a split into cascading **when** contracts should be performed. These constants can take on any values of the appropriate type.*

Evaluation of different guard expressions may take different amounts of time; thus, experimentation may be required to determine the safe value for k for a specific contract.

► **Definition 14** (Transformed Contracts). *Given a contract C , and choice name c , we use the notation $\lceil C \rceil_c$ to denote the result of applying the transformation $(\text{sg } c)$ to C . We also overload the notation for $\lceil \langle C, \sigma \rangle \rceil_c$ to denote application of the transformation to C as well as all contracts mentioned in the environment in σ . See Definition 3 for the definition of the environment in σ .*

In the example from Figure 2, $Q = \lceil P \rceil_{\text{fresh}}$. To verify this transformation, we must show that there is a weak bisimulation between them for an arbitrary restriction set \mathcal{A} , i.e. that $\langle P, \sigma \rangle \setminus \mathcal{A} \approx \langle \lceil P, \sigma \rceil_{\text{fresh}}, \sigma \setminus (\mathcal{A} \cup \{\langle \text{choice}, p, \text{fresh}, 0 \rangle\}) \rangle$.

<pre> 1 when { 2 a -> c1, 3 a -> c2 4 } after t -> { close } </pre>	<pre> 1 when { 2 a -> c1, 3 p chooses fresh within [0, 0] -> { 4 when { 5 a -> c2 6 } after t -> { close } 7 } 8 } after t -> { close } </pre>
--	---

■ **Figure 4** A contract without disjoint guards before and after applying the transformation.

4.1 Restriction - Making Observable Actions Unobservable

The strategy of the transformation is to transform a **when** contract having a long list of guarded commands into a new contract having a structure of cascading **when** contracts whose guarded command lists are length less than or equal to $k + 1$. To accomplish this, we introduce a choice action with a choice name that does not occur in the original contract called c . The guard in the contract takes the form $(p \text{ chooses } c \text{ within } [0, 0])$, so that inputs of the form $\langle \langle \text{choice}, p, c, 0 \rangle, \# \rangle$ induce a transition.

► **Definition 15** (Restricted Choice Action). *Given a choice name $c \in \mathbf{String}$ we define the choice action, $\alpha_c \stackrel{\text{def}}{=} \langle \text{choice}, p, c, 0 \rangle$.*

The action α_c is used to traverse the cascading **when** structure introduced by the transformation. As long as c does not occur as choice name in P , α_c can safely be used to internally transition along the chain of cascading **whens** introduced by **sg** in $\lceil P_c \rceil$ (see Figure 3). Adding α_c to the restriction set hides those transitions when evaluating $\lceil P_c \rceil$. This allows us to show that there is a bisimulation between $P \setminus \mathcal{A}$ and $\lceil P_c \rceil \setminus (\mathcal{A} \cup \{\alpha_c\})$.

4.2 Verifying the Transformation

To be a candidate for the transformation requires a kind of predictability in the behavior of the contract to be transformed. Specifically, the guards in every **when** having more than k guarded commands must be disjoint, in the sense that at most one guard evaluates to true in any state-action pair.

The example contracts in Figure 4 demonstrate the issue that occurs if the guards are not disjoint. In the original contract on the left there is no way to get to the **c2** continuation. This is because the guards in a **when** are evaluated in the order they appear in the list. If the guarded command $(a \rightarrow c1)$ is triggered, then $(a \rightarrow c2)$ would have been triggered as well. $(a \rightarrow c1)$ will always continue to **c1** before $(a \rightarrow c2)$ is ever evaluated. On the other hand, the transformed contract on the right can reach both **c1** and **c2**. If the label $\langle a, \# \rangle$ reaches **c1**, then the sequence $\langle \alpha_{\text{fresh}} \cdot a, \# \rangle$ reaches **c2**, which was previously unreachable. Also note that α_{fresh} is hidden, so to any observer, the labels $\langle \alpha_{\text{fresh}} \cdot a, \# \rangle$ and $\langle a, \# \rangle$ appear to be identical. As this example shows, the transformed contract may have more behaviors than the original contract, and there is no evaluation of the original contract that will simulate those behaviors. By requiring the guards to be disjoint, we guarantee that there is no unreachable code uncovered by the transformation. In practice, most contracts are written to have disjoint guards. For example, the three-party escrow contract in Figure 1 has disjoint guards since all choice guards in the **when** are for different parties or allow different choices of values for the vote, and there is only one deposit guard.

While outside the scope of this work, Appendix B describes a transformation that will make all guards in the lists of guarded commands disjoint for contracts where programmers find the need to write non-disjoint guards. The transformation would maintain the behavior of the contract by prioritizing the guard that appears earlier in the **when**. By modifying the logic of the later guards, it is possible to remove the logic that overlaps with the earlier guard while maintaining the non-overlapping logic. Since bisimulations are equivalence relations, once a bisimulation is shown to exist using the disjoint transformation, the disjoint transformation can be composed with the **sg** transformation, and the result would be observationally equivalent to the original contract.

► **Definition 16** (Disjoint Guards). *A contract-state pair $\langle C, \sigma \rangle$ is disjoint if, for each contract in $\langle C, \sigma \rangle$ of the form (when gs after $t \rightarrow cont$), the list of guarded commands gs is disjoint. gs is disjoint when, for all l and r such that $gs = l \mathrel{++} r$, the following two properties hold in for all actions in all execution states:*

- i.) *If l contains a guard that evaluates to true then all guards in r evaluate to false.*
- ii.) *If r contains a guard that evaluates to true then all guards in l evaluate to false.*

Next, we define a binary relation between original processes and their transformed counterparts, and prove it is a bisimulation.

► **Definition 17** (Split Guards Relation). *For all $c \in \mathbf{String}$ let \mathbb{SG}_c be the following binary relation over \mathbf{Proc} :*

$$\begin{aligned} \mathbb{SG}_c \stackrel{\text{def}}{=} \{ & \langle \langle C, \sigma_1 \rangle \backslash \mathcal{A}, \lceil C, \sigma_2 \rceil_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \rangle \\ & \mid c \text{ fresh in } \langle C, \sigma_1 \rangle, \text{ disjoint } \langle C, \sigma_1 \rangle, \\ & \sigma_1 = \langle \Gamma, e, accts, choices_1, t, ps, asrt \rangle, \\ & \sigma_2 = \langle \Gamma, e, accts, choices_2, t, ps, asrt \rangle, \\ & \forall x \in \mathbf{String}, p \in \mathbf{Party}. x \neq c \text{ implies } choices_1[(x, p)] = choices_2[(x, p)] \} \end{aligned}$$

The binary relation \mathbb{SG}_c is a relation in $\mathbf{Proc} \times \mathbf{Proc}$, where the second element is the result of applying the transformation function (**sg** c) to the first element. Also note, the original contract must have disjoint guarded command lists; and the Faustus execution states in the two elements of the relation are equivalent up to the choices for the new choice name c . We will use \mathbb{SG}_c to show a bisimulation between an original contract and a transformed contract under the condition that the choice name used in the transformation is not in the original contract.

► **Lemma 18.** *For all $c \in \mathbf{String}$, \mathbb{SG}_c is a weak simulation over $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$.*

► **Lemma 19.** *For all $c \in \mathbf{String}$, \mathbb{SG}_c^{-1} is a weak simulation over $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$.*

Proof. The proofs of Lemmas 18 and 19 are done in Isabelle/HOL by induction on the structure of the experiment relation (\Rightarrow), and by cases on the structure of the transition relation (\rightarrow). Assume membership in the relation (\mathbb{SG}_c or \mathbb{SG}_c^{-1}) and an arbitrary transition for the first member of the relation. Then for each case of the transition relation, we show the second member of the relation will transition on an experiment containing the same label to another element of \mathbf{Proc} that maintains membership in the relation (\mathbb{SG}_c or \mathbb{SG}_c^{-1}). ◀

► **Theorem 20** (Transformation Verification). *Let $\langle C, \sigma_1 \rangle \backslash \mathcal{A}, \lceil C, \sigma_2 \rceil_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \in \mathbf{Proc}$ be states in the Faustus LTS $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$, for some arbitrary $c \in \mathbf{String}$ such that:*

- i.) *c is fresh for C and σ_1 , i.e. c does not occur as a choice name in either.*
- ii.) *Guarded command lists in $\langle C, \sigma_1 \rangle$ are disjoint.*

iii.) $\sigma_1 = \langle \Gamma, e, accts, chs_1, t, ps, asrt \rangle$
iv.) $\sigma_2 = \langle \Gamma, e, accts, chs_2, t, ps, asrt \rangle$,
v.) $\forall x \in \mathbf{String}, r \in \mathbf{Party}. x \neq c \text{ implies } chs_1[(x, r)] = chs_2[(x, r)]$.
 Then $\langle C, \sigma_1 \rangle \backslash \mathcal{A} \approx \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\})$.

Proof. Assume an arbitrary $c \in \mathbf{String}$, $\langle C, \sigma_1 \rangle \backslash \mathcal{A}, \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \in \mathbf{Proc}$, and that (i.) through (v.) hold. From Lemma 18, Lemma 19, and Definition 12 of bisimulation, we can conclude that $\mathbb{S}\mathbb{G}_c$ is a bisimulation. Then by the definition of $\mathbb{S}\mathbb{G}_c$, we know that $\langle \langle C, \sigma_1 \rangle \backslash \mathcal{A}, \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \rangle \in \mathbb{S}\mathbb{G}_c$. Thus $\langle C, \sigma_1 \rangle \backslash \mathcal{A} \approx \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\})$. ◀

With Theorem 20 we have verified contracts have the same observable behavior before and after the transformation is applied. Programmers can safely apply the transformation to their contracts using the initial execution state and empty restriction set before they are loaded onto the blockchain. The conditions required to guarantee the bisimulation is valid are: *i.)* the choice name used in the transformation is fresh, and *ii.)* the guarded command lists in the original contract are all disjoint.

5 Conclusions and Future Work

In this paper we have described an Isabelle formalization of a bisimulation verifying that smart contracts written in Faustus and their transformed instances are observationally equivalent. The formalization introduces a notion of hidden actions in a way different from Milner's CCS [34]. Also, Hoare's CSP [21] has a restriction operator, but it differs in a number of ways; perhaps most significantly, his restriction sets enumerate the allowable actions.

There are a few clear ways this work can be continued. The first is that we plan to apply this methodology to many more transformations. First among them will be to formalize, in Isabelle, one that transforms an arbitrary guarded command list into one that is disjoint. A strategy for this transformation is described in Appendix B. Since the verification of the transformation in this paper requires the guarded command lists in the contract to be disjoint, it would be useful for programmers to have more contracts that the transformation can be applied to.

Also, we plan to apply these techniques to other smart contract programming languages (perhaps Solidity [8]). In Faustus, labels for the transition relation are clearly given in the syntax of the contract as guards in the guarded commands of a **when** contract. In Solidity, the **require** statements act like guards. Marmosler and Brucker [28] have formalized the semantics of Solidity in Isabelle/HOL, which would serve as a good starting point.

Another area of investigation is applying the additional methodologies used for Timed CCS. We include time information as part of the labels in the Faustus LTS and use that information to update the **State** time information according to the Faustus semantics. Timed CCS variants use clock signals as labels in processes transitions [5, 27]. In order to show a bisimulation between Faustus processes, the **State** information must be included in the relation. By moving the time information out of the **State** and using the clock labels that timed CCS variants use, it should reduce the complexity of the relations required to verify smart contract transformations.

References

- 1 Cardano protocol parameters reference guide | Cardano Docs. URL: <https://docs.cardano.org/about-cardano/explore-more/parameter-guide>.

- 2 Internet Crime Complaint Center (IC3) | Cyber Criminals Increasingly Exploit Vulnerabilities in Decentralized Finance Platforms to Obtain Cryptocurrency, Causing Investors to Lose Money. URL: <https://www.ic3.gov/PSA/2022/PSA220829>.
- 3 isabelle · master · Secure Systems Collaborative / Public / WABL / Faustus V2 · GitLab, February 2025. URL: <https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/tree/master/isabelle>.
- 4 Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A. Schett. Super-optimization of smart contracts. *ACM Trans. Softw. Eng. Methodol.*, 31(4):70:1–70:29, July 2022. doi:10.1145/3506800.
- 5 Henrik Reif Andersen and Michael Mendler. An asynchronous process algebra with multiple clocks. In Donald Sannella, editor, *Programming Languages and Systems — ESOP ’94*, pages 58–73, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- 6 Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, November 2006. doi:10.1007/s10009-006-0010-1.
- 7 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-54455-6_8.
- 8 The Solidity Authors. Solidity — Solidity 0.8.29 documentation, 2024. URL: <https://docs.soliditylang.org/en/latest/>.
- 9 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity Verification of Smart Contracts. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods*, pages 256–266, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-76554-4_14.
- 10 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor utxo blockchains. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroSP)*, pages 235–267, 2024. doi:10.1109/EuroSP60621.2024.00021.
- 11 Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18*, pages 83–100, New York, NY, USA, October 2018. Association for Computing Machinery. doi:10.1145/3243734.3243795.
- 12 Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 222–247, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17138-4_10.
- 13 Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. Behavioral simulation for smart contracts. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, pages 470–486, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386022.
- 14 Brian Bush. marlowe-cardano/marlowe/best-practices.md at main · input-output-hk/marlowe-cardano · GitHub, January 2023. URL: <https://github.com/input-output-hk/marlowe-cardano/blob/main/marlowe/best-practices.md#avoid-or-break-up-complex-logic-in-the-contract>.
- 15 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, pages 525–539, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-54455-3_37.
- 16 James Chapman, Arnaud Bailly, and Polina Vinogradova. Applying Continuous Formal Methods to Cardano (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture*, pages 18–24, Milan Italy, August 2024. ACM. doi:10.1145/3677998.3678222.

- 17 Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017. doi:10.1109/SANER.2017.7884650.
- 18 RChain Cooperative. Contract Design — RChain Architecture 0.9.0 documentation, 2017. URL: <https://architecture-docs.readthedocs.io/contracts/contract-design.html>.
- 19 Fritz Henglein, Christian Kjær Larsen, and Agata Murawska. A formally verified static analysis framework for compositional contracts. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 599–619, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54455-3_42.
- 20 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
- 21 C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science. Prentice/Hall International, Englewood Cliffs, N.J, 1985.
- 22 Philipp Kant, Kevin Hammond, Duncan Coutts, James Chapman, Nicholas Clarke, Jared Corduan, Neil Davies, Javier Díaz, Matthias Güdemann, Wolfgang Jeltsch, Marcin Szamotulski, and Polina Vinogradova. Flexible formality practical experience with agile formal methods. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming*, pages 94–120, Berlin, Heidelberg, 2020. Springer International Publishing. doi:10.1007/978-3-030-57761-2_5.
- 23 Olga Kharif. Crypto-Bridge Hacks Reach Over \$1 Billion in Little Over a Year. *Bloomberg.com*, March 2022. URL: <https://www.bloomberg.com/news/articles/2022-03-30/crypto-bridge-hacks-reach-over-1-billion-in-little-over-a-year>.
- 24 Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: Implementing and analysing financial contracts on blockchain. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 496–511, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54455-3_35.
- 25 Pablo Lamela Seijas, David Smith, and Simon Thompson. Efficient static analysis of marlowe contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 161–177, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61467-6_11.
- 26 Pablo Lamela Seijas and Simon Thompson. Marlowe: Financial contracts on blockchain. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 356–375, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-03427-6_27.
- 27 Luigi Liquori and Michael Mendler. Strong Priority and Determinacy in Timed CCS. Technical report, Inria ; University of Bamberg, December 2023. URL: <https://inria.hal.science/hal-04367635>.
- 28 Diego Marmosoler and Achim D. Brucker. A Denotational Semantics of Solidity in Isabelle/HOL. In Radu Calinescu and Corina S. Păsăreanu, editors, *Software Engineering and Formal Methods*, volume 13085, pages 403–422. Springer International Publishing, Cham, 2021. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-030-92124-8_23.
- 29 Kegan McIlwaine, Stone Olguin, and James Caldwell. Faustus: Adding formally verified parameterized abstractions to the smart contract language marlowe, 2022. Unpublished Manuscript. URL: https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/blob/master/papers/WTSC22_Authored.pdf.
- 30 Kegan McIlwaine, Stone Olguin, and James Caldwell. Developing faustus: A formally verified smart contract programming language, 2023. Unpublished Manuscript. URL: https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/blob/master/papers/iFM_2023.pdf.

- 31 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. London, UK, September 1-3, 1971, pages 481–489. William Kaufmann, 1971. URL: <http://ijcai.org/Proceedings/71/Papers/044.pdf>.
- 32 Robin Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157–173. Elsevier, 1975. doi:10.1016/S0049-237X(08)71948-7.
- 33 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1980. doi:10.1007/3-540-10235-3.
- 34 Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., USA, 1989.
- 35 Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1201–1242. Elsevier and MIT Press, 1990. doi:10.1016/B978-0-444-88074-1.50024-X.
- 36 Robin Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, USA, 1999.
- 37 Keerthi Nelaturu, Sidi Mohamed Beillahi, Fan Long, and Andreas Veneris. Smart contracts refinement for gas optimization. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 229–236, 2021. doi:10.1109/BRAINS52497.2021.9569819.
- 38 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 39 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer. doi:10.1007/BFb0017309.
- 40 Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351267.
- 41 Meixun Qu, Xin Huang, Xu Chen, Yi Wang, Xiaofeng Ma, and Dawei Liu. Formal Verification of Smart Contracts from the Perspective of Concurrency. In Meikang Qiu, editor, *Smart Blockchain*, pages 32–43, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-05764-0_4.
- 42 Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 478–493, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_30.
- 43 Nick Szabo. Smart Contracts, 1994. URL: <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart.contracts.html>.
- 44 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7), July 2021. doi:10.1145/3464421.

```

datatype FAction =
  Deposit FParty FParty Token FValue
| Choice FChoiceId "Bound list"
| Notify FObservation

datatype FStatement =
  Pay FAccountId FPayee Token FValue
| Assert FObservation
| ReassignVal Identifier FValue
| ReassignObservation Identifier FObservation
| ReassignPubKey Identifier FParty

datatype FGuardExpression =
  ActionGuard FAction
| GuardThenGuard FGuardExpression FGuardExpression
| GuardStmtsGuard FGuardExpression "FStatement list"
| DisjointGuard FGuardExpression FGuardExpression
| InterleavedGuard FGuardExpression FGuardExpression

datatype FContract =
  Close
| StatementCont FStatement FContract
| If FObservation FContract FContract
| When "FCase list" Timeout FContract
| Let Identifier FValue FContract
| LetObservation Identifier FObservation FContract
| LetPubKey Identifier FParty FContract
| LetC Identifier "FParameter list" FContract FContract
| UseC Identifier "FArgument list"
and FCase = Case FGuardExpression FContract

```

■ **Figure 5** Isabelle: constructors for Statements, Guard Expressions, and Contracts.

A Faustus Abstract Syntax Tree

The Isabelle data type definitions in Figure 5 are the result of parsing a Faustus program. Each constructor represents a different type of the same syntactical object. The constructors of `FContract` each define a type of Faustus contract. Similarly, the constructors of `FGuardExpression` and `FStatement` each define a type of guard expression or statement that can occur in a Faustus contract, respectively. The constructors of `FAction` give the types of basic action guards for the actions that users can take while interacting with Faustus processes. The list inside the `When` contract is a list of `Cases`, the full constructor for a guarded command.

B The Disjoint Guards Transformation

The verification of the `sg` transformation requires guarded command lists in the original contract to be disjoint. In Figure 6 we provide an algorithm for creating disjoint guards when given two that may match on the same user action. Since guarded command lists and

```

fun left_prioritize_action :: "FState  $\Rightarrow$  FAction  $\Rightarrow$  FAction  $\Rightarrow$  FAction" where
"left_prioritize_action  $\sigma$ 
  (Deposit p11 p12 t1 v1)
  (Deposit p21 p22 t2 v2) =
  (if (evalFParty  $\sigma$  p11 = evalFParty  $\sigma$  p21)  $\wedge$ 
    (evalFParty  $\sigma$  p12 = evalFParty  $\sigma$  p22)  $\wedge$ 
    t1 = t2
  then Deposit p21 p22 t2 (Cond (ValueEQ v1 v2) (Constant (-1000)) v2)
  else Deposit p21 p22 t2 v2)" |
"left_prioritize_action  $\sigma$ 
  (Notify obs1)
  (Notify obs2) =
  (Notify (AndObs (NotObs obs1) (obs2)))" |
"left_prioritize_action  $\sigma$ 
  (Choice (FChoiceId cn1 p1) bounds1)
  (Choice (FChoiceId cn2 p2) bounds2) =
  (if (evalFParty  $\sigma$  p1 = evalFParty  $\sigma$  p2)  $\wedge$  cn1 = cn2
  then Choice (FChoiceId cn2 p2) (left_prioritize_bounds bounds1 bounds2)
  else Choice (FChoiceId cn2 p2) bounds2)" |
"left_prioritize_action  $\sigma$  _ a = a"

```

■ **Figure 6** Isabelle: constructors for Statements, Guard Expressions, and Contracts.

guard expressions prioritize the top/left, we can make contract guards disjoint by modifying the guard that has lower priority. A full Isabelle/HOL verification of this algorithm has not been completed yet, but it will use the bisimulation method described in this paper.

In Faustus, a negative deposit is not an action that a user can make. Thus, if two **deposits** are identical, to avoid a situation where the **deposit** guards overlap, we disable the second deposit guard by making its value negative. In this way, we make two **deposit** guards disjoint while prioritizing the first. Note that disabling the second will have no effect, because it is identical to the earlier one.

The algorithm handles **choice** guards much like **deposit** guards. Checking the equality of the choice names and parties, and then performing an interval difference operation gives disjoint choice guards.

In a simpler case, two **notify** guards can be made disjoint by taking the Boolean formula of the left guard $b1$ and combining it with the Boolean formula in the right guard $b2$ in the formula $\neg b1 \wedge b2$.

Finally, `left_prioritize_action σ _ a = a` handles all other cases, where the guards being prioritized are not the same, *e.g.* the case where the first is a **deposit** guard and the second is a **notify**

Figure 6 shows how to perform these operations on the Faustus AST.