Using Bisimulation to Formally Verify the Correctness of Smart Contract Transformations*

Midwest PL Summit - November 22, 2024 - University of Chicago

Kegan McIlwaine & James Caldwell
Department of Electrical Engineering and Computer Science
University of Wyoming

*This work was funded by a grant from IOG, Wyoming state funds, and the University of Wyoming

Characters in This Play

Marlowe …… Financial smart contract Domain Specific Language (DSL)
Faustus …… A financial smart contract language
The blockchain …… The Cardano blockchain with the Ada token

Haskell …… The implementation language
Isabelle …… The theorem prover

Structural operational semantics …… Formal approach to specifying system behaviors
The compiler …… Translates Faustus into Marlowe
CCS …… Calculus of Communicating Systems
Bisimulation …… An equivalence relation between processes

Act I

Verifying the Faustus to Marlowe Compilation

Marlowe

- Marlowe is a domain specific language (DSL) embedded in Haskell for writing financial contracts on the Cardano blockchain (Seijas and Thompson, 2018).
- The syntax of a Marlowe contract is defined as a Haskell algebraic datatype with 6 constructors, one for each kind of contract:

```
data Contract = Close
              | Pay Party Payee Token IExp Contract
              | If BExp Contract Contract
              | When [Case] Timeout Contract
              | Let Identifier IExp Contract
              | Assert BExp Contract
```

The Faustus Smart Contract Language

- Easier to write Faustus syntax than Marlowe embedded in Haskell.
- Constructs of the language include:
  - Contracts, declarations, modules, effects (payments, assertions, and variable reassignments), parties, guarded cases, actions, choices, tokens, and an expression language (integer, Boolean, and POSIX time).
- Faustus supports declarations of parameterized abstractions for all constructs of the language (including declarations).
- Concurrency operators from deterministic CCS (Milner, 1980; Camilleri & Winskel, 1991; Liquori & Mendler, 2023) for specifying interactions between parties and the contract.

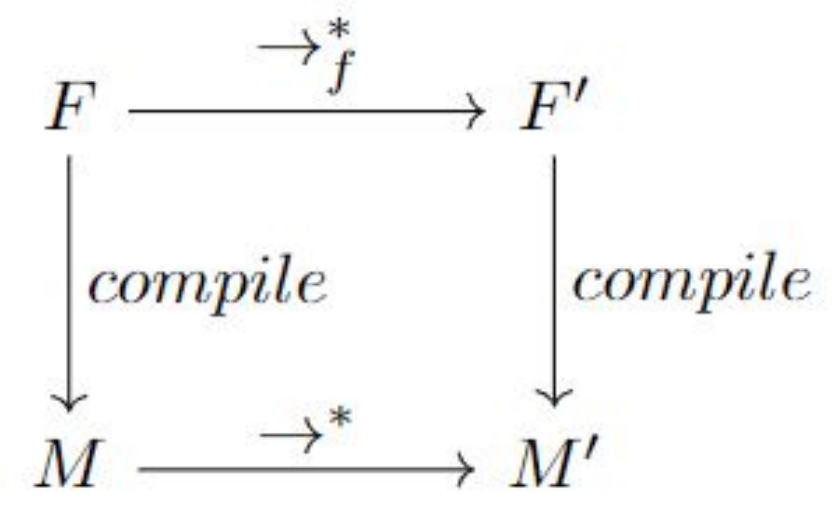
  (**a1 <+> a2**) - Choices between Actions. (Do this, or do that)
  (**a1 ||| a2**) - Actions that are expected concurrently. (All possible interleavings)
  (**a1 -> a2**) - Sequencing of Actions. (Do this, then do that)

Example Faustus Contract

```
when {
      lender deposits 100 ada -> {
            pay !borrower 100 ada from lender;
            when {
                  borrower deposits 110 ada -> {                              // Repay the loan
                        pay !lender 110 ada from borrower;
                        close
                  },
                  lender chooses forgive -> { close },                        // Forgive the loan
                  notify start > 250 -> {                                     // Late fee
                        when {
                              borrower deposits 120 ada -> {                  // Repay with late fee
                                    pay !lender 120 ada from borrower;
                                    close
                              }
                        } after 500 -> { close }
                  }
            } after 500 -> { close }
      }
} after 100 -> { close }
```

Compiler Preservation of Semantics

A commutative diagram specifying the correctness of the Faustus compiler. We have proved this diagram commutes using the Isabelle interactive theorem prover for a subset of Faustus using techniques from Nipkow and Klein (2014). We're victims of the expression problem (Wadler, 1998).



Act II

Formalizing Faustus Processes, User Interactions, and Transformations

Faustus Process Transitions

Faustus Processes are a Faustus Contract paired with the current State.

We now label transitions in Faustus processes as either observable or internal (but not both) from the users' perspectives.

- Observable transitions in the operational semantics require user input. Labeled with the user input.
- Unobservable transitions do not require user input. Labeled with τ.

The operational semantics of Faustus define the transition relation, P =>e P', where P transitions to P' on observable transitions e with zero or more unobservable transitions before and after each observable transition.

Bisimulations Between Faustus Processes

A set R is a simulation if whenever a pair of Faustus processes, P and Q, are in R then P =>e P’ implies there exists a Q’ such that Q =>e Q’ and (P’, Q’) ∈ R.

R is a bisimulation if it and its converse are simulations.

If R is a bisimulation and (P, Q) ∈ R. Then P and Q are said to be observationally equivalent.

A Smart Contract Transformation (Bush 2023)

The number of guarded actions in a **when** may require too much time to check before running out of gas. Assume k+1, (k ≥ 0), can be checked in the allotted time.

Split the **when** into multiple **when**s using **notify true**. Any user action matching **a_j** (where j > k) needs a preceding **notify** action leading to another **when** which includes **a_j**. This strategy is repeated inductively. Then, guard checking occurs across multiple **when**s.

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      .
      a_n -> c_n
} after 100 -> { close }
```

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      a_k -> c_k
      notify true -> when {
            a_{k+1} -> c_{k+1}
            .
            .
            .
            a_n -> c_n
      } after 100 -> c_t
} after 100 -> c_t
```

A Smart Contract Transformation (Bush 2023)

There are two issues:

1. If the jth clause is **notify true -> c_j**, where j <= k, there is no simulation.
2. The **notify** action is observable.

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      .
      a_n -> c_n
} after 100 -> { close }
```

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      a_k -> c_k
      notify true -> when {
            a_{k+1} -> c_{k+1}
            .
            .
            .
            a_n -> c_n
      } after 100 -> c_t
} after 100 -> c_t
```

Act III

Fixing and Verifying the Transformation

Fixing the Smart Contract Transformation

Strategy addressing issue 1:

Split the **when** into cascading **whens** using a *fresh* choice name that doesn't exist in the contract.

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      .
      a_n -> c_n
} after 100 -> { close }
```

```
when {
      a_1 -> c_1,
      a_2 -> c_2,
      .
      .
      a_k -> c_k
      advancer chooses advance -> when {
            a_{k+1} -> c_{k+1}
            .
            .
            .
            a_n -> c_n
      } after 100 -> c_t
} after 100 -> c_t
```

Showing a Bisimulation

- Create a process, ADV, that runs concurrently with the users sending actions and the Faustus Process. Users send their actions to ADV which will forward them to the contract.
  a. Case: user action that matches **a_j** where j <= k
     ADV forwards user action to contract.
  b. Case: user action that matches **a_j** where j > k
     ADV sends “advance”, then sends user action in two transactions.
- The original Faustus process before transforming the contract is F.
- The new Faustus process after transforming the contract is F_t.
- The new process users interact with is F’=ADV⌢F_t. The ⌢ operator links the outputs from the left to the inputs of the right (Milner 1999).
- We have proved there is a bisimulation between F and F’ in Isabelle.

Credits (Citations)

Camilleri, J., & Winskel, G. (1991). CCS with priority choice. [1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science, 246–255. https://doi.org/10.1109/LICS.1991.151649

Lamela Seijas, P., & Thompson, S. (2018). Marlowe: Financial Contracts on Blockchain. In T. Margaria & B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice (pp. 356–375). Springer International Publishing. https://doi.org/10.1007/978-3-030-03427-6_27

Leroy, X. (2006). Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 42–54. https://doi.org/10.1145/1111037.1111042

Liquori, L., & Mendler, M. (2023). Strong Priority and Determinacy in Timed CCS.

Milner, R. (Ed.). (1980). A Calculus of Communicating Systems (Vol. 92). Springer. https://doi.org/10.1007/3-540-10235-3

Milner, R. (1999). Communicating and mobile systems: The π-calculus. Cambridge University Press.

Nipkow, T., Klein, G., & Klein, G. (2014). Concrete Semantics: With Isabelle/HOL (2014 edition.). Springer Nature. https://doi.org/10.1007/978-3-319-10542-0

Wadler, P. (1998, November 12). The Expression Problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Questions

Process Algebras

- Syntactical building blocks for having processes branch, interact in sequence, and run concurrently.
- CSP (Hoare, 1978) provided the first formal framework for defining processes and how they evolve and interact concurrently.
- CCS (Milner, 1980???) provided a similar formal framework for defining concurrent processes.
  - Differentiates between internal and observable reactions in and between processes.
  - Formalized bisimulation for showing processes are observationally equivalent.
- Pseudo-code for a zero coupon bond contract could be written:

  lender deposits *amount* into *borrower* account ->
  emit payment to *borrower* of *amount* ->
  borrower deposits *amount* + *interest* into *lender* account ->
  emit payment to *lender* of *amount* + *interest*

Marlowe

- Marlowe is a domain specific language (DSL) for writing financial smart contracts on the blockchain.
- Marlowe lacks traditional control flow operators (other than **When** or **If** branching). Control is explicitly passed using a continuation passing model.
- All Marlowe programs are fully expanded instances of the syntax tree, *i.e.* they are inline programs that always terminate.
- A Marlowe contract executes until closed, or it requires user input that hasn't occurred before timing out.

```
data Contract = Close
              | Pay Party Payee Token Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
              | Assert Observation Contract
```

A Marlowe Contract

This is a zero coupon bond contract where one user, the “Lender”, sends a loan to another user, the “Borrower”. The “Borrower” later pays the loan back with interest added.

Some simple auction contracts end up being over one gigabyte of text.

```
When [
  (Case
     (Deposit
        (Role "Lender")
        (Role "Lender")
        (Token "" "")
        (ConstantParam "Amount"))
     (Pay
        (Role "Lender")
        (Party
           (Role "Borrower"))
        (Token "" "")
        (ConstantParam "Amount")
        (When [
           (Case
              (Deposit
                 (Role "Borrower")
                 (Role "Borrower")
                 (Token "" "")
                 (AddValue
                    (ConstantParam "Interest")
                    (ConstantParam "Amount")))
              (Pay
                 (Role "Borrower")
                 (Party
                    (Role "Lender"))
                 (Token "" "")
                 (AddValue
                    (ConstantParam "Interest")
                    (ConstantParam "Amount")) Close))]
           (TimeParam "Payback deadline") Close))))]
  (TimeParam "Loan deadline") Close
```

Example Contract

When contracts wait for user inputs.

$\mathbf{a_i}$ **->** $\mathbf{c_i}$ are guarded contracts. The contract $\mathbf{c_i}$ is triggered if action $\mathbf{a_i}$ occurs in the stream of user inputs.

There are three kinds of actions users can input: **notify**, **deposit**, or **choice**.

If $\mathbf{a_i}$ is **notify bExp** and a **notify** occurs in the input stream when **bExp** is true, then $\mathbf{c_i}$ is triggered.

```
when {
      loaner deposits 100 ada -> {
            pay !loanee 100 ada from loaner;
            when {
                  loanee deposits 110 ada -> {
                        pay !loaner 110 ada from loanee
                  }
            }
      },
      a_2 -> c_2,
      .
      .
      .
      a_n -> c_n
} after 100 -> close
```

If $\mathbf{a_i}$ is a **deposit** action and a matching **deposit** occurs in the input stream, then $\mathbf{c_i}$ is triggered.

If $\mathbf{a_i}$ is a **choice** action and a matching **choice** occurs in the input stream, then $\mathbf{c_i}$ is triggered.

Characters in This Play

DSL ……..... Domain specific languages

Marlowe …….

Faustus ……..

The blockchain …….