Standardized crypto-loans on the Cardano blockchain

Dmytro Kondratiuk[1], Pablo Lamela Seijas[1][0000−0002−1730−1219], Alexander Nemish[1], and Simon Thompson*[1,2][0000−0002−2350−301X]

[1] IOHK, Hong Kong, dmytro.kondratiuk@iohk.io, pablo.lamela@iohk.io, alexander.nemish@iohk.io, simon.thompson@iohk.io
[2] School of Computing, University of Kent, UK, s.j.thompson@kent.ac.uk

Abstract. Crypto-loans are innovative financial instruments that allow trustless peer-to-peer lending, and potentially providing a safe and convenient source of liquidity for cryptocurrency holders. In this paper we explore a smart contract framework for building standardised crypto-loans using the Marlowe domain-specific language and the ACTUS standard for financial contracts.

Keywords: ACTUS · blockchain · Cardano · finance · Haskell · Marlowe · smart contract · static analysis

1 Introduction

Smart contracts – programs that run in a blockchain environment – can be defined in a variety of ways [9]. Many such approaches are general purpose, and can be used to program any kind of contract it makes sense to run on a blockchain; moreover, they tend to be expressive enough to be Turing complete (in some cases with restrictions on the runtime environment). For example, Plutus, the general-purpose language running on the Cardano blockchain [3], is a dialect of Haskell. Another approach is to develop special-purpose or *domain-specific languages* (DSLs) which embody a particular application domain: Marlowe [8] is a high-level DSL for writing financial contracts on the Cardano blockchain.

In this paper we explore ways in which contracts described in ACTUS (Algorithmic Contract Types Unified Standards) can be defined in the contract languages Marlowe, Plutus and Haskell. Of course, Plutus or Haskell are able to express these contracts, but rendering them in Marlowe brings extra advantages. Marlowe is defined to provide a range of guarantees by design: a Marlowe contract will only make a finite number of interactions with its environment, and its lifetime can be read off from the code for a contract; moreover, when the contract terminates, any assets held by the contract will automatically be returned to the participants. None of these guarantees can be provided by a general purpose language.

* Corresponding author.

Each Marlowe contract has a finite set of possible execution paths, and so it is possible to analyse the complete behaviour of a contract without running it. Such *static analysis*, based on SMT solving [12], can be used to check properties of a contract; for example, it is possible to check whether a contract will honour all the `Pay` constructs that it contains, however it is executed. In the case that a `Pay` can fail, the analysis gives an example trace showing how that failure happens. The language design and static analysis provide assurance that Marlowe contracts are much less likely to "misbehave" than contracts written in a general-purpose language like Solidity.

In implementing ACTUS on Cardano we are able to provide further assurance in three other ways. First, we are able to use the declarative nature of Haskell to transliterate ACTUS formulas term-by-term into an executable form in Haskell. Secondly, we are able to use random, property-based testing to validate the Haskell implementation against another written in Java. Finally, we are able to automatically *generate* ACTUS contracts in Marlowe from the terms – i.e. parameters – of the contracts; for a simple loan these would include the start and end dates of the loan and the amount loaned (the 'principal'). The generated contracts use the executable spec in calculating values of cash flows in the Marlowe contracts.

The contribution of our work is to show that that it is possible to implement financial contracts on blockchain in a way that *multiple forms of assurance* are provided: from the language itself, from the static analysis, and from custom property verification. The development environment for Marlowe, the Marlowe Playground [3], also provides a simulation environment for contracts for stepping forwards (and backwards) through contract execution, and thus allowing users to validate that contracts perform as they should. In addition, implementing ACTUS provides a suitable benchmark against which to assess the design of Marlowe; we illustrate how the implementation has led to the addition of a conditional expression construct to the language.

In the remainder of the paper, Section 2 covers the relevant financial background, including the ACTUS financial standard. Section 3 builds an executable specification of ACTUS in Haskell, and this is used in Section 4 to generate the Marlowe code for an ACTUS contract from the contract terms. Section 5 explains how tokens are used to represent ownership of roles in a running contract, and Section 6 describes how we provide assurance that contracts behave as they should. Section 7 examines related work and Section 8 concludes.

A note on notation: `typewriter` font will be used for Marlowe constructs while *math* font will be used for mathematical formulas and pseudo-code.

2 Financial contracts

In this section we give a brief introduction to financial contracts, and to loans in particular, and then describe the ACTUS financial standard.

[3] `https://alpha.marlowe.iohkdev.io/#/`

2.1 Crypto-loans

A loan is a form of debt incurred by an individual or other entity. The lender advances a sum of money to the borrower. In return, the borrower agrees to a certain set of terms including any finance charges, interest, repayment date, and other conditions [10].

Cryptocurrency-backed loans must have *collateral* when there is no trust between party and counterparty. While a loan is usually settled in a stable-coin currency (e.g. USDT/USDC), collateral is typically denominated in a cryptocurrency (e.g. BTC). The purpose of such a loan is to give the borrower access to the fiat value of their crypto-funds without actually selling them for fiat. The borrower pays interest in exchange for gaining liquidity.

Every loan has a positive net payoff (return minus investment) that is either rendered as a one-time payment – often called a *zero-coupon bond* (ZCB) – or by scheduling payment of the interest. The rate of interest could be fixed throughout the lifetime of a contract: for example, zero-risk bonds have a fixed interest proportional to the inflation rate. However, in the generic case the interest rate is variable and depends on an external factor agreed in advance, and the rate is periodically updated by observing the state of that factor.

Such loans often represent an investment in a particular venture or industry. As a somewhat fictional example, one could imagine a cryptocurrency miner who decided to scale their crypto-farm: a loan (in USD) with variable interest that directly depends on cryptocurrency prices would be more attractive for a miner because it would directly correlate with miner's profits. For example, if the price of the cryptocurrency goes down in a particular month the borrower would have to pay lower interest, and so would always pay a fixed share of the profits. In a more traditional setup the interest rate could depend on prices of other commodities: a canonical example would be a power plant taking a loan with interest depending on electricity prices.

In both cases, the prices of cryptocurrency or electricity become a driver for the interest rate. However, one cannot simply take the bare price of the asset and turn it into a rate. In order to make *units of measurement* compatible with each other adjustments should be made. Fluctuations of the interest rate driver are embedded thus:

$$\Delta_r = \mathit{capfloor}(\mathit{driver} * \mathit{multiplier} + \mathit{spread} - \mathit{interestRate}_{t-1})$$

$$\mathit{interestRate}_t = \mathit{capfloor}(\mathit{interestRate}_{t-1} + \Delta_r),$$

where *capfloor* is a function that limits the range of fluctuation, and so limiting the lender's exposure to risk:

$$\mathit{capfloor}(x) = \mathit{max}(\mathit{min}(x, \mathit{floor}), \mathit{cap})$$

The *spread* parameter here loosely represents the difference between the average prime rate that the lender expects – the *benchmark yield* – and the rate imposed by the driver: the higher the spread, the higher the resulting interest rate. The multiplier rescales the interest rate curve in order to represent the changes to

be made converting between different units of measurement: how many rate percentage points you would get for a USD-to-kilowatt conversion and so on.

In the context of ACTUS and similar frameworks, there is one more factor influencing interest rates thorough scaling:

$$interestPayment = interestScalingFactor * interestRate * notional$$

This scaling is dynamic and loosely adjusts for variance (volatility) of the asset that the interest rate driver represents.

Interest accrual and capitalisation. A counterparty might decide to reinvest profit received as interest from the loan. In the simplest case, this renders as compound interest. This can be modelled through interest accrual and capitalisation (conversion of income or assets into capital); for instance, contracts from the ACTUS specification accrue interest between interest payments and can transfer interest to a notional during interest capitalisation event (IPCL).

Overall, variable interest rates introduce a certain risk for a lender, thus they can be subject to hedging. While any instrument that depends on the same risk factor (interest rate driver) would suffice, the most popular way to hedge a variable interest rate loan is an interest rate swap. This instrument allows two (or more) parties to exchange their incomes - one from a fixed interest rate loan, the other from a variable-rate loan.

Counterparty risk. Trustless setups, especially ones in the cryptocurrency world, including decentralised smart-contracts and exchanges, require no trust between party and counterparty involved in a contract. In case of a loan, this literally means that counterparty has zero obligation to pay the money back, thus rendering the loan useless for a party. Such risks are usually addressed by introducing collaterals, as in the following scenario.

1. Alice would like to borrow 1000 USD
2. She has Bitcoin assets cost around 1500 USD, which she intends to hold throughout a year, so Alice has high confidence in the market (she expects prices to double or triple)
3. Bob would like to lend 1000 USD and get an interest higher than traditional interest rate offered by banks (let's say 15% instead of 10%). He is either bearish or neutral towards Bitcoin.
4. Alice transfers her BTC as collateral to a contract, and Bob transfers his USD to Alice
5. If Alice pays the interest and notional on time, and the BTC price does not render collateral worthless, she can get her collateral back; otherwise the loan gets liquidated and the collateral is transferred to Bob.

2.2 ACTUS

The Algorithmic Contract Types Unified Standards (ACTUS) [1] define the logic embedded in legal agreements that eventually turn the contract terms into actual

cash flows, or more generally business events. Most of its basic contract types represent different variations of lending contracts. ACTUS provides additional benefit of being regulatory friendly, and the ACTUS foundation provides a set of tools allowing Monte-Carlo simulations of ACTUS contracts.

ACTUS relies on a state machine formalism in order to describe the behaviour of a given contract. Every *payoff* – i.e. transfer of funds in or out of a contract – can be inferred for any given state. Every state can be derived from previous events and observed risk factors:

$$payoff_i = POF(state_i)$$

$$path_i = STF(ct, ev_1) \circ STF(ct, ev_2) \circ \ldots \circ STF(ct, ev_i)$$

$$state_i = path_i(INIT(ct)),$$

where *ct* stands for contract terms, *INIT* returns initial state, *sched* returns scheduled events, *STF* takes *contract terms*, *event*, and *state* and returns the next state, and *POF* returns the *payoff* in a state.

2.3 Oracles

In order to support variable interest rates and scaling, ACTUS requires a smart contract to be able to observe the value of a given risk factor, such as an interest rate, at a particular point in time t. This is due to the state of the risk factor not being known at instantiation time.

$$riskfactor_{it} = O_{rf}(i, t)$$

In the case of the Cardano blockchain, these values are usually provided through an *oracle* mechanism[11]. An oracle could be a trusted party providing necessary data or network of parties under consensus [2].

From a Marlowe DSL perspective, the exact mechanism that provides external data is less important, as Marlowe abstracts over IO by requiring a particular type of input – a `Choice` – that is protected with a cryptographic signature by the source of the choice. As a result, the event of receiving data from an oracle is treated the same as receiving numeric input in other languages

3 Building an executable specification of ACTUS

ACTUS is defined in a textual specification[4] which, while expressed in mathematical notation, is essentially informal. In this section we describe how this specification is turned into an *executable* version by rendering it in Haskell. This translation is in fact a transliteration, since notation, variable names and so forth are respected.

[4] Available from `https://www.actusfrf.org/techspecs`.

3.1 Rendering the specification in Haskell

The ACTUS standard is specified in terms of scheduling, payoff and state transition functions that are polymorphic on event and contract type, as noted in Section 2.2 above. The specification also follows quite specific naming conventions that are incompatible with Haskell's conventions. The executable specification follows original ACTUS conventions as closely as possible in order to ease code base maintenance when faced with updates of the ACTUS spec repository[5].

Using Haskell itself as a DSL for explicitly encoding formulas without using advanced language idioms also simplifies code generation. In case of ACTUS this comes at a cost reduced type-safety, handling nullable values explicitly introduces risk of exceptions. However this risk is addressed using property-based testing, and in particular QuickCheck generators. This is discussed in more detail in Section 6 below.

3.2 Utilising polymorphism to abstract over basic operations

In order to keep our executable specification independent of the carrier – whether it is a smart-contract engine, proof assistant, analytical framework or even machine learning model – we abstract over the underlying representation of state variables,

```
-- Definitions/ContractState.hs
data ContractStatePoly a b = ContractStatePoly
{
  tmd     :: b
  , nt    :: a
  , ipnr  :: a
  , ipac  :: a
  , feac  :: a
  , fac   :: a
  , nsc   :: a
  , isc   :: a
  , prf   :: ContractStatus
  , sd    :: b
  , prnxt :: a
  , ipcb  :: a
} deriving (Show)
```

and arithmetic operations,

```
-- Ops.hs
class ActusOps a where
   _min :: a -> a -> a
   _max :: a -> a -> a
```

[5] https://github.com/actusfrf/actus-techspecs

```
   _zero :: a
   _one :: a

class ActusNum a where
   (+) :: a -> a -> a
   (-) :: a -> a -> a
   (*) :: a -> a -> a
   (/) :: a -> a -> a

class YearFractionOps a b where
   _y :: DCC -> a -> a -> a -> b

class DateOps a b where
   _lt :: a -> a -> b --returns pseudo-boolean

class RoleSignOps a where
   _r :: ContractRole -> a
```

Thus, every formula in the executable spec could be instantiated to:

- a formula on some *atomic* type, like `Double` or `Day`, which could be used to directly compute cash-flows for analytical purposes or precompute payoffs for smart contracts that do not depend on oracles; or
- a formula representing a piece of *abstract syntax*, e.g. a Marlowe `Value` or `Observation`, that could be used to generate smart contracts that depend on oracles or to generate code in another language, such as Agda.

This approach of abstracting formulas has a limitation of not allowing conditionals to be expressed in an abstract way: in other words, there is no `ActusIf` typeclass. Luckily most of conditional expressions in ACTUS specification don't depend on variable state of a contract, they depend on `ContractTerms` that are known in advance during contract generation. This allows us to dispatch appropriate formulas during generation rather than execution.

The only exception to this are the rare situations where we need to compare 2 state variables and choose either $formula'$ or 0 depending on the result of the comparison result:

$$formula(st) = \begin{cases} formula'(st) & var_1(st) < var_2(st) \\ 0 & otherwise \end{cases}$$

We rely on a pseudo-Boolean *less than* function in order to address that:

$$formula(st) = pseudoLt(var_1(st), var_2(st))) * formula'$$

$$pseudoLt(a, b) = Cond(a > b, 1, 0)$$

3.3 Contract term representation and explicit applicability

In order to simplify serialisation and deserialisation of contract terms across ACTUS related services maintained by Cardano we rely on "superposed" representation of contract terms: all ACTUS contract types are represented with the same type.

While such a representation allows both encoder and decoder to express any ACTUS contract terms, it also allows for invalid combinations of terms (for example *PRNXT* cannot be applied to a PAM contract), which means contracts require specific validation that is implemented by means of the applicability function:

$$Applicability : ContractTerms \to Bool$$

ACTUS standard defines a family of applicability functions polymorphic on contract type:

$$Applicability : ContractType \times ContractTerms \to ApplicabilityType$$

where applicability could be: *none*, *always*, *nullable*, or *multiple*.

In order to build superposed contract terms type for such functions, we have to resolve conflicting applicability types for merged contract terms using the following resolution rules:

$$weaken(a_1, a_2) = \begin{cases} nullable & a_1 = none \wedge a_2 = always \\ nullable & a_1 = always \wedge a_2 = none \\ a_1 & priority(a_1) > priority(a_2) \\ a_2 & otherwise \end{cases}$$

$$priority(x) = \begin{cases} 0 & x = always \\ 1 & x = none \\ 2 & x = nullable \\ 3 & x = multiple \end{cases}$$

where a_i is the applicability of a given term of the ith contract type to be merged.

4 Generating Marlowe contracts from standardised ACTUS contract terms

In this section we describe how concrete Marlowe contracts are generated from the terms – i.e. parameters – of standard ACTUS contracts. We also describe the implementation of the system, and reflect on the limitations of generating contracts in Marlowe, where contracts have a predefined lifetime and a predefined collection of interactions with contract participants.

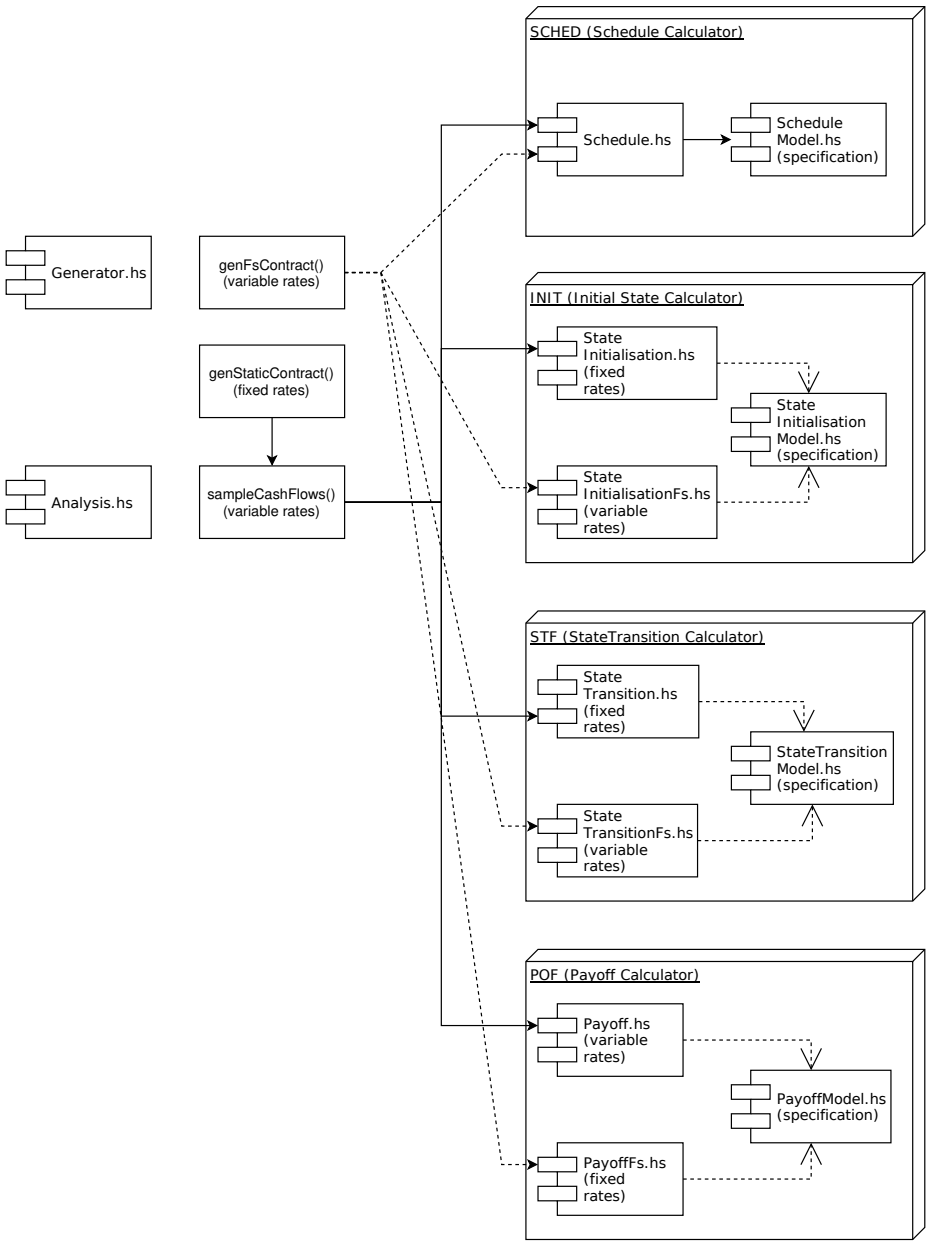

Fig. 1. Modules responsible for contract generation.

4.1 Overall architecture

A generated contract is essentially a continuation chain of smaller contracts:

$$chainlink(t) = receiveData(t) \circ calculatePayoff(t) \circ processPayoff(t)$$

$$contract(ct) = collaterals(ct) \circ INIT(ct) \circ \prod_{t \in SCHED(ct)} chainlink(t)$$

where the component *receiveData* asks an oracle for Marlowe Choice if needed, and *calculatePayoff* calculates the payoff formula. For fixed-rate contracts this is optimised into a pre-calculated constant function. The *processPayoff* function awaits the `Deposit` of a payoff amount from a party: if the deposit is made them it directs the funds to a counterparty, otherwise it transfers the collateral to the counterparty and closes the contract.

The principal components of the system are shown in Figure 1. There are three categories of components representing ACTUS functions: specification with formulas, formula wiring for fixed rates (produces precomputed payoffs) and formula wiring for variable rates, which produces Marlowe-code that computes payoffs. Different types of wiring correspond to different implementations of *ActusOps* implementations (either *Double* or *Value*).

The chain is generated from the fixed schedule of events, known in advance of execution, using the *SCHED()* function from ACTUS, as shown in Figure 2.

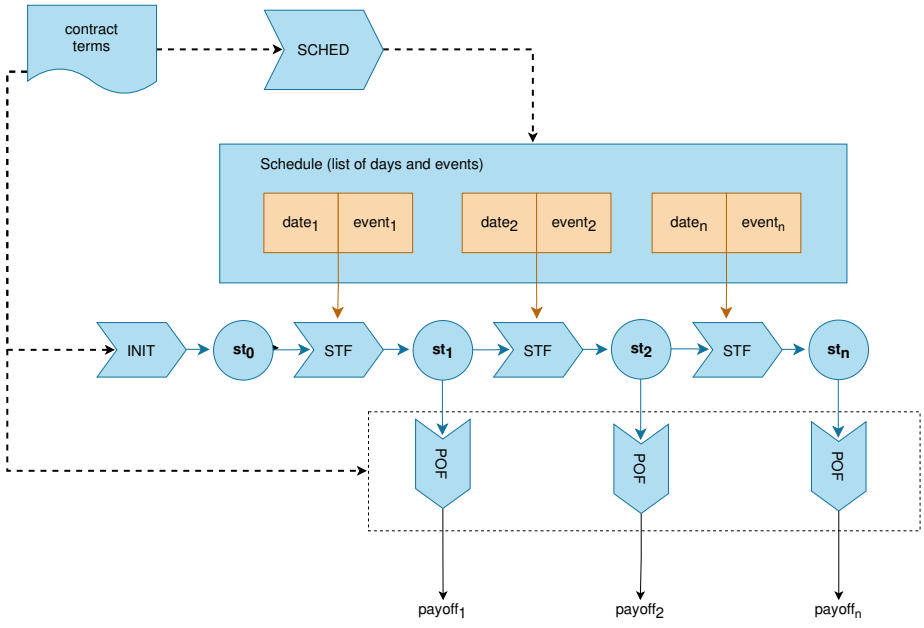

Fig. 2. Chain of sub-contracts representing ACTUS logic

4.2 Avoiding exponential growth

Marlowe contracts are finite, and in particular Marlowe itself does not have constructs for functions or recursion; these *are* available in the Haskell and JavaScript embeddings of Marlowe, but they are unrolled on translation to pure Marlowe. Naive usage of the `If` operator in Marlowe could lead to exponential growth of a contract, as in this pesudocode example:

```
if condition
  then
    perform_something1()
    continue()
  else
    perform_something2()
    continue()
```

Translating this to Marlowe would inline contents of `continue()` twice and, given that ACTUS contracts are essentially generated using continuation as an accumulator, this would lead to exponential explosion of the size of any ACTUS contract that has conditionals in their state transition logic.

An example of such logic would be cap/floor limitations on interest rates:

$$adjusted = max(min(original,floor),cap)$$

We addressed this issue by introducing the `Cond` expression construct in order to represent conditional expressions, rather than only conditional contracts as was the case before. Instead of using the `If` contract to decide the value of some variable, we use a conditional expression instead. `Cond` is a pure function that returns a value depending on a condition, in contrast to the `If` contract that chooses between two continuation contracts.

4.3 Limitations due to termination

Marlowe doesn't allow contracts that run indefinitely, even if their recursion is productive, as would be the case in a perpetual swap contract, for example. We therefore cannot support certain contract types from ACTUS specification, namely the ones that don't have a defined maturity date (like UMP).

There is a possible workaround: contracts with no maturity date could be represented as actors with a finite number of state transitions. We prototyped this approach, however it does seem more prone to errors comparing to rendering predefined schedules. More importantly, it greatly affects static analysis because the number of reduction steps in the contract grows from

$$N_{scheduled} = count(event)$$

to

$$N_{stateTransitions} = \frac{max(date(event)) - min(date(event))}{precision}$$

4.4 Fixed-point precision

For numeric types Marlowe supports Integers, while ACTUS is expressed in terms of real numbers. In order to model a real number in such a setup we rely on fixed-point precision. The algebra looks like this:

```
(+)   = AddValue -- x/n + y/n = (x + y)/n
(-)   = SubValue -- x/n - y/n = (x - y)/n
a * b = Scale (1 % marloweFixedPoint) $ MulValue a b
                            -- x/n * y/n = (x * y)/n^2
```

We scale all numbers with *marloweFixedPoint* factor - which only requires modification of *MulValue*. We plan to move Marlowe to fixed-precision numbers for the on-blockchain implementation available later in 2021.

4.5 Representing Actus state in a Marlowe contract

The Marlowe DSL does not support any notion of records, the variables can only be of type "Integer". In order to map contract state – *ContractStatePoly* – we pack a set of Marlowe variables of type `Value` and `Observation`, representing the previous $(t-1)$ state, passing them into *ContractStatePoly*, to apply the polymorphic state transition, and finally unpack *ContractStatePoly* into a set of Marlowe variables representing the state at t:

$$st_t = unpack(stateTransition(pack(st_{t-1}))),$$

where *pack* is a chain of `UseValue` constructs and *unpack* is a chain of `Lets`.

Representing state transitions in Marlowe Marlowe is a declarative language, and so in particular it does not support mutable variables. We therefore represent the state at stage t (st_t) literally through this naming convention:

```
variableName(name, t) =
  concat(name, '_', t)
generateAccessor(name, t) =
  UseValue variableName(name, t)
generateSetter(name, t, formula) =
  Let variableName(name, t) formula
```

4.6 Actus Labs

In order to demonstrate and test the capabilities of Actus generators, a visual online Blockly-based tool was developed for the Marlowe Playground. The Actus Labs tool, shown in Figure 3, allows users to construct contract terms visually to generate a corresponding Marlowe contract and then to try it out in a simulation environment.

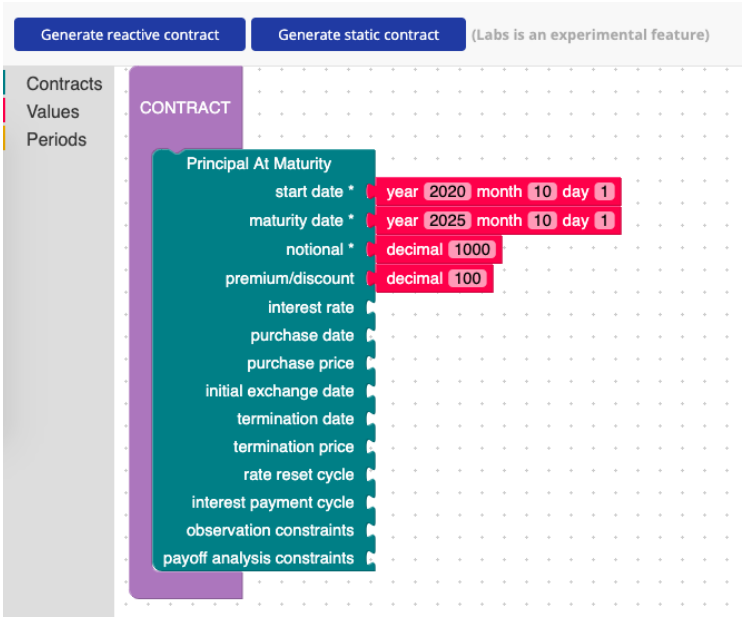

Fig. 3. Actus Labs - an online tool for generating Actus contracts for Marlowe.

5 Tokenization

Every participant of a Marlowe contact is described by a `Role` which is in its turn represented through a unique non-fungible token, created at the time that the contract instantiated on the blockchain. This makes every ACTUS contract a tradable security, allowing a participant to sell its *share* in a contract by selling a corresponding role token.

Role tokens can potentially allow more complex manipulation over such shares, especially when the share represents an incoming cash flow; in that case, participants send funds to a party represented by a given token. Such a token would represent a positive cashflow, which in turn could not only become tradable but could also allow the derivation of tokens representing *fractional parts* of a particular cash flow in a contract.

Moreover, this process turns ACTUS loans into derivatives. For example, contracts like *Interest Rate Swap* (and *Swaps* in general) could be approximated by an *Atomic Swap* of tokens representing incoming cash flows from loans. For example, if Alice has fixed income from a loan, or some other investment, and Bob has comparable but variable (fluctuating) income, Bob can hedge by swapping cash flows with Alice. If Alice's income is locked with `token1` and Bob's income is locked with `token2` then an atomic swap of those tokens is equivalent to a swap of cash flows.

6 Assurance

This section explains how we provide assurance to users of our Marlowe ACTUS contracts by means of property-based testing and SMT-based static analysis.

6.1 QuickCheck for cross-testing

First, we are able to test the executable Haskell implementation of ACTUS for smart-contracts by comparing it with an existing implementation written in Java. To do this a simple property-based test in QuickCheck [4] was introduced, where we generate contract terms and risk factors randomly to test the property.

$$\forall ct. \forall rf.\ getCashFlows(\textit{"haskell"}, ct, rf) \equiv getCashFlows(\textit{"java"}, ct, rf)$$

where *ct* represents contract terms, *rf* is risk factor model, and *getCashFlows()* returns a set of *(date,payoff)* tuples.

6.2 QuickCheck for verification

QuickCheck contract terms generators also allow us to check other properties of a Marlowe contract. This could be enhanced with Marlowe's static analysis feature by utilising the `Assert` operator:

```
do
let contractTerms = sample(qcgenerator)
    contract = generateMarloweContract(contractTerms)
    contractWithAssert = appendAssertion(contract, assertion)
runStaticAnalysis(contractWithAssert)
```

While this scenario does not cover all possible contracts, it could guarantee that property holds for a statistically significant fraction of a contract.

6.3 Static analysis for verification

Using static analysis for Marlowe [7] it is possible fully to check a particular contract instead of using random sampling. That would allow some refinements:

- More balanced sampling: the space of contract terms depends linearly on the coverage, e.g. if we cover 10% of all possible contract terms - we'll cover 10% of all contracts. Different contracts have different sets of risk factors, and so different search spaces. For instance, a contract with 3 risk factors (e.g. 3 observations of an interest rate) would span over n^3 values while a contract with 10 risk factors would span over n^{10}. Meanwhile, the space of all risk factors in all contracts doesn't linearly depend on the coverage - some contracts might have more risk factor observations and some - less. So covering 10% of all contracts doesn't necessary mean covering 10% of all possible observations.

- Dependency tracking: an SMT-solver is potentially likely to be more aware of execution paths that lead to the failure of a test, a feature that could significantly reduce search space.
- Completeness: if not timed out, SMT-solving is decidable while sampling is semi-decidable.

6.4 Securing collateral logic with auto-refund warnings

By design, the Marlowe interpreter always refunds any assets held in a contract when it terminates. This is done in order to ensure that no funds are lost forever. The funds are returned to whoever's internal account holds them.

However, this presents as a problem in certain cases where ownership of the funds could not be determined automatically. For example, if Alice puts collateral in a crypto-loan contract she would formally maintain ownership, which means she would get automatically refunded when a `Close` construct is reached.

This implicit refund is easy to overlook by smart-contract developers as plain `Close` is often used as a default action in case of unexpected behaviour like timeouts, and especially a `Choice` timeout. This can easily lead to costly mistakes if Alice maliciously decides to exploit an auto-refund feature in order to get her collateral without paying back, as in the following scenario:

1. Alice creates a contract where a `Deposit` timeout would lead to `Close`
2. Bob doesn't know or test the timeout path. Even if Bob is a programmer, he might not be aware that `Close`ing the contract would cause the collateral to be refunded to Alice.
3. Alice puts her collateral in the contract and gets the notional from Bob.
4. Alice doesn't pay for the loan: i.e. there is a `Deposit` timeout.
5. Alice gets her collateral back.
6. Bob loses his notional.

In order to prevent this from happening, an additional `Auto-Refund` security check was introduced as part of static analysis tooling which notifies users about all `Close` constructs that can lead to automatic refunds and encourages users to write explicit logic for edge cases.

7 Related work

Unlike current mainstream DeFi lending approaches, our Marlowe ACTUS implementation relies on trade-matching instead of pooling. While asset pooling is proven to be superior to order-book based approaches when it comes to automated market makers, using it for lending has been shown to be more susceptible to attacks [5]. Moving trade matching off-chain also improves scalability of the protocol - every loan is a separate contract, thus there is no global state.

There are frameworks, like ISDA Common Domain Model [6], providing a more precise representation of the business processes involved in institutional trading as well as code-generation capabilities. However, due to its structural simplicity, ACTUS is more suited for generating code in a financial domain specific language like Marlowe rather than general-purpose one like Java or Haskell.

8 Conclusion

The Marlowe language was explicitly designed as a set of building blocks for financial contracts that could be combined by anyone familiar with basic programming. The Marlowe ACTUS generators improve on that by providing a way to automatically combine blocks based on standardised requirements specified by the user. Marlowe ACTUS also provides a toolkit for cross-testing against the original ACTUS spec, a framework for adding new contract types, cash-flow visualisations and verification tooling.

We are very grateful to colleagues at Quanterall and Finley and Kegan McIlwaine of the University of Wyoming for their contributions to this project.

References

1. ACTUS Homepage. https://www.actusfrf.org/, [last accessed 02-02-2020] 2.2
2. Beniiche, A.: A study of blockchain oracles. https://arxiv.org/abs/2004.07140 [last accessed 04-02-2020] (2020) 2.3
3. Brünjes, L., Gabbay, M.J.: UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms. In: Margaria, T., et al. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. Springer (2020) 1
4. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ICFP 2000. ACM, New York (2000). https://doi.org/10.1145/351240.351266 6.1
5. Flash-loan attack definition. https://www.coindesk.com/harvest-finance-24m-attack-triggers-570m-bank-run-in-latest-defi-exploit, [last accessed 02-02-2020] 7
6. ISDA Common Domain Model. https://www.isda.org/2019/10/14/isda-common-domain-model/, [last accessed 02-02-2020] 7
7. Lamela Seijas, P., Smith, D., Thompson, S.: Efficient Static Analysis of Marlowe Contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. Springer (2020) 6.3
8. Lamela Seijas, P., Thompson, S.: Marlowe: Financial Contracts on Blockchain. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. Springer (2018) 1
9. Lamela Seijas, P., Thompson, S., McAdams, D.: Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156 (2016), https://eprint.iacr.org/2016/1156 1
10. Loan definition. https://www.investopedia.com/terms/l/loan.asp, [last accessed 02-02-2020] 2.1
11. Mammadzada, K., et al.: Blockchain Oracles: A Framework for Blockchain-Based Applications. In: Asatiani, A., et al. (eds.) Business Process Management: Blockchain and Robotic Process Automation Forum. Springer (2020) 2.3
12. Vanegue, J., Heelan, S., Rolles, R.: SMT Solvers for Software Security. In: Proceedings of the 6th USENIX Conference on Offensive Technologies. p. 9. WOOT'12, USENIX Association, USA (2012) 1