Efficient static analysis of Marlowe contracts

Pablo Lamela Seijas[1[0000−0002−1730−1219]], David Smith[1[0000−0003−1859−8007]], and Simon Thompson[1,2[0000−0002−2350−301X]]

[1] IOHK, Hong Kong, pablo.lamela@iohk.io, david.smith@tweag.io, simon.thompson@iohk.io
[2] School of Computing, University of Kent, UK, s.j.thompson@kent.ac.uk

Abstract. SMT solvers can verify properties very efficiently, and they offer increasing flexibility on the ways those properties can be described. However, the way in which properties are described can affect the computational cost of verifying them considerably. In addition, it is quite hard to predict how different changes will affect the computational cost. In this paper, we discuss the lessons we have learned while implementing and optimising the static analysis functionality for Marlowe, a domain specific language for describing self-enforcing financial smart-contracts that can be deployed on a blockchain.

1 Introduction

Thanks to static analysis, we can automatically check beforehand whether payments promised by a Marlowe contract can be fulfilled in every possible execution of the contract. For example, if a Marlowe contract has passed the static analysis, we will have a very high assurance that whenever the contract says we will be paid a certain amount of money, the contract will indeed have enough money to make such payment.

State of the art libraries like SBV [5] allow straightforward static analysis of arbitrary Haskell functions, with very few restrictions on how those functions are implemented. However, static analysis can be very computationally expensive, because it essentially needs to check every possible execution of the contract.

For example, in the case of Marlowe, contracts are typically interactive and must deal with inputs arriving from different users in different orders, and sometimes not arriving at all. In addition, Marlowe allows several inputs to be grouped in a single transaction, and different decisions may be taken depending on the content of those inputs, or in the combinations of contents of those inputs. All these alternatives multiply the search space of static analysis and can easily compose exponentially and thus make the whole approach unusable in practice even for moderately sized contracts.

This paper contributes a number of approaches that can be used when optimising static analysis, examples extracted from a case-study where we applied these approaches, and an overview of the techniques used to test the correctness of the optimisations.

The case study presented is that of the optimisation of Marlowe's static analysis. During this work, the techniques applied allowed us to reduce the analysis time of an implementation that followed the semantics closely and took a couple of minutes to analyse contracts of a few kilobytes, to one where the same contract would take less than a second, and where a four-person crowdfunding contract that when fully expanded occupies about 19 megabytes, can be analysed in around 10 minutes using Z3. In the original implementation, we were not able to analyse the 19 megabyte contract because it ran out of memory after half an hour.

There are many things that can be done to improve the efficiency of static analysis. We have classified them as lightweight and heavyweight.

Lightweight modifications are local and can be done without fundamentally changing the implementation. We consider three main ideas:

- **Removing unnecessary parts from the analysis**. If they do not affect the property that is being verified then we can just remove them from the implementation and reduce its complexity.
- **Avoiding high level abstractions**. High level abstractions aid reasoning and avoid errors, but also introduce complexity that is not necessarily needed.
- **Reducing search space by normalising parameters**. If there are several ways of representing some inputs, and the different representations have no impact on the analysis, we can remove all but one when implementing static analysis.

Heavyweight modifications are more fundamental approaches that require considerable changes to the structure of the implementation. We consider two main ideas:

- **Reducing search space by using normalised execution paths relevant to the property**. Instead of using search space to model inputs, we use it to model possible executions, and we ensure that we only represent each possible group of equivalent executions once, i.e: executions with the same normal form.
- **Minimizing the representation of inputs and outputs**. We can benefit from encoding inputs and outputs as concisely as possible, and discarding information that we can infer in other ways.

In the following sections, we introduce the semantics of Marlowe as a case study (Section 2), we go through a general approach to static analysis (Section 3), then we explore both lightweight (Section 4.1) and heavyweight (Section 4.2) optimisation techniques in more detail, and we illustrate them with examples of how they apply to Marlowe static analysis. In Section 5, we briefly discuss some ways in which property based testing can be used to address potential problems that may arise from the implementation of the heavyweight optimisations described in the previous section. In Section 6, we present empirical results that show the effects of heavyweight optimisations on the execution time and memory usage of Marlowe's static analysis.

2 Marlowe and how its design helps static analysis and correctness

Firstly, we will briefly introduce the semantics of Marlowe, the guarantees that are offered implicitly by the semantics, and how the design choices facilitate static analysis and make it decidable. A more detailed explanation of Marlowe can be found in [10].

2.1 Structure of Marlowe contracts

Marlowe contracts are able to receive payments, store money and tokens, ask for input from the participants, and redistribute stored money and tokens back among participants. The decision about which and when of these actions are carried out is ~~taken by~~ the contract on the basis of the information that is available at each point in time. Participants may correspond to either individual public keys or to tokens (`Roles`). In turn, `Roles` may be controlled by separate independent contracts.

The Marlowe language is written as a set of mutually recursive Haskell data types. Marlowe contracts regulate interactions between a finite number of participants, determined before the contract is deployed.

The main data type in Marlowe is called `Contract`, and it represents the logic and actions that the contract allows or enforces. The outmost constructs of the `Contract` represent the actions that will be enforced first and, as those constructs become enforced, the `Contract` will evolve into one of its continuations (sub-contracts), and the process will continue until only the construct `Close` remains.

There are 5 constructs of type `Contract`:

- `Close` – signals the end of the life of a contract. Once it is reached, all the money and tokens stored in the accounts of the contract will be refunded to the owner of each of the respective accounts.
- `If` – immediately decides how the contract must continue from between two possibilities and depending on a given Boolean condition that we call an `Observation`. The value of an `Observation` can depend on previous choices made by participants, on amounts of money and tokens remaining, or other factors.
- `Let` – immediately stores a `Value` for later use. Expressions of type `Value` in Marlowe are evaluated to integer values and they depend on information available to the contract at the time of evaluation. For example, a `Let` construct can be used to store the minimum slot of the slot interval of the current transaction, or the amount of money available in a particular account at a given point in time, for use at a later stage of the contract execution.
- `When` – waits for an external input. The `When` construct also specifies a timeout slot: after this slot has been reached, the `When` construct expires, and no longer accepts any input. There are three types of input:
  - `Deposit` – Waits for a participant to deposit an amount of money or tokens (specified as a `Value`) in the contract. The amount is added to one of the accounts of the contract.

- `Choice` – Waits for a participant to make a choice. A choice is represented as an integer number from a set specified by the contract.
- `Notify` – Waits for an `Observation` to be true. Because contracts are reactive (they cannot initiate transactions), it is necessary for an external actor to `Notify` the contract that the `Observation` has become true. However, this notification does not have to be carried out by a participant, anybody can do it.

– `Pay` – immediately makes a payment between accounts of the contract, or from an account of the contract to a given participant. The amount transferred is specified as a `Value`.

2.2 Semantics

As we mentioned in Section 2.1, Marlowe contracts are passive: their code is executed as part of the validation of transactions that are submitted to the blockchain. These transactions need to be submitted by participants or their representatives (e.g. user wallets) and their validation is carried out atomically and deterministically.

Each transaction may include a list of inputs, a set of signatures, minimum and maximum slot numbers, a set of input UTxOs (incoming money and tokens), and a set of outputs (outgoing money and tokens). For a transaction to be valid in Marlowe, the transaction must have the same effect for every slot within that slot range (be deterministic). For example, if a transaction has a minimum slot number which is before a timeout, and a maximum slot which after a timeout then the transaction will fail with the error `AmbiguousSlotInterval`, since it would not be deterministic.

A key aspect of the Marlowe semantics is that it checks that a particular transaction is valid given the current state and contract. Because transactions are deterministic, there should be no reason why someone accidentally sends a transaction that is invalid for a given `State` and `Contract`, since it will only result in a cost to that participant.

It is possible that, due to a race condition, a participant will send a transaction that no longer applies to a running `Contract` and `State`, but such a transaction would simply be ignored by the blockchain.

The type signature of the transaction validation function is:

```
computeTransaction :: TransactionInput -> State -> Contract
                   -> TransactionOutput
```

This function can be factored into four main functions:

– `reduceContractStep` - this function simplifies the topmost construct that does not require an input to be executed (i.e. anything but a `When` that has not expired or a `Close` when accounts are empty). It only simplifies the `When` construct if it has expired (i.e. the timeout specified in the `When` is less than or equal to the minimum slot number). In the case of the `Close` contract, it only refunds one of the accounts at a time.

- `reduceContractUntilQuiescent` - this function calls `reduceContractStep` repeatedly until it has no further effect (i.e. it reaches a fixed point).
- `applyInput` - this function processes one single input. The topmost construct must be a `When` that is expecting that particular input and has not expired.
- `applyAllInputs` - this function processes a list of inputs. It calls `applyInput` for each of the inputs in the list, and calls `reduceContractUntilQuiescent` before and after every call to `applyInput`.

The `State` stores information about the amount of money and tokens in the contract at a given time, together with the choices made, `Let` bindings made, and a lower bound for the current slot:

```
data State = State { accounts    :: Map AccountId Money
                   , choices     :: Map ChoiceId ChosenNum
                   , boundValues :: Map ValueId Integer
                   , minSlot     :: Slot }
```

2.3 Extra considerations

Many of the design decisions behind Marlowe have been made with the aim of preventing potential errors. For example:

- **Classification of money and tokens into accounts** separates concerns. Marlowe will never spend more money or tokens than there are in an account, even if there is more available in the contract. But when required to pay more than is in an account, it will pay as much as is available in the account, in order to remain as close as possible to the original intention of the contract.
- **Account identifiers include an account owner**. An account owner is a participant that will get the money or tokens remaining in an account when a contract terminates. At the same time, the only construct that can pause the execution of a contract is the `When` construct, which has a timeout, this ensures that all contracts eventually expire and terminate. Together, these properties ensure that no money or tokens are locked in the contract forever.
- **No negative deposits or payments**. Marlowe treats negative amounts in deposits and payments as zero. However, at the same time, if there happens to be a request for deposit with a negative amount, it continues gracefully: it will still wait for a null deposit to be made, and it will continue as if everything is correct; this way, the expected execution of the contract is disturbed as little as possible.
- **Upper limit in the number of inputs** that a Marlowe contract can accept throughout its lifetime. This limit is implied by the maximum number of nested `When` constructs in the contract, since only one input per `When` can be accepted. At the same time, transactions that have no effect on the contract are considered invalid, thus there is also a limit in the maximum number of transactions a contract can accept throughout its life. This bound prevents DoS attacks, and it makes static analysis easier. We discuss this in more detail in Section 3.

3 Making Marlowe semantics symbolic

In this section, we briefly present and reflect on a technique that can be used to convert a concrete implementation of a Haskell function into a symbolic one by using the SBV library, and to use this symbolic implementation for static analysis. In particular, we explore this technique in the context of the Marlowe semantics.

This approach corresponds to our first attempt at implementing static analysis for Marlowe contracts, and it is a systematic approach that can be carried out with very few assumptions.

3.1 Overview

The SBV library supports implementing Haskell functions in a way that the same implementation can be used:

- With concrete parameters, as a normal Haskell function
- With symbolic variables, so that properties can be checked for satisfiability using an SMT solver
- As part of QuickCheck properties, for random testing.

Parameters that can be used symbolically are wrapped in a monad called `SBV`. Functions that depend on symbolic parameters also return values wrapped in the `SBV` monad.

The idea is therefore to convert our implementation to use values wrapped in the `SBV` monad, by replacing all operations on those values with symbolic functions. In order to simplify this work, `SBV` provides symbolic versions for many of the most common built-in functions that are either overloaded or have similar names.

Our semantics transaction processing function would thus become:

```
computeTransaction :: SBV TransactionInput -> SBV State
                   -> SBV Contract
                   -> SBV TransactionOutput
```

We can then define a function that takes a list of transactions instead, and we would just need to write a property that says that the result of executing the list of transactions for a given contract does not have any warnings (the property we are trying to verify). Then we can pass the property to SBV and it will tell us which transaction lists break the property.

Unfortunately, there are a couple of issues with this approach that we review in Section 3.2.

3.2 Additional considerations

SBV does not currently support complex custom data types. At the time of writing, SBV has some support for nonary data types, but not for data types

with parameters. It does provide symbolic versions of the `Either` type and the `Tuple` type. Thus, it is possible to have an isomorphism between custom data types and nested `Either` and `Tuple` type-synonyms. Our original implementation automatically generates conversion functions between these by using Template Haskell. By using this technique the symbolic implementation can remain similar to the concrete one.

For example, the following data structure:

```
data Input = IDeposit AccountId Party Money
           | IChoice ChoiceId ChosenNum
           | INotify
```

Would be translated to the following type synonym:

```
type SInput = SBV (Either (AccountId, Party, Money)
                          (Either (ChoiceId, ChosenNum)
                                  ()))
```

But this approach cannot address recursive datatypes, let alone mutually recursive datatypes. And the `Contract` type is mutually recursive.

Even if we could represent mutually recursive datatypes, it would not be feasible to directly translate the semantics, because a more general limitation of SMT solvers is that they cannot, in general, deal with arbitrarily sized computations. In other words, if termination of a symbolic function is bounded by a parameter that can be arbitrarily large, SMT solvers will often fail to terminate when trying to validate a property, since doing so often requires a proof by induction. We discuss how to address this problem in Section 3.3.

3.3 Adapting the semantics

In order to guarantee termination of the SMT validation, we need a concrete bound for the property that we want to validate.

Related work often addresses this problem by manually establishing a bound on the amount of computation, e.g: limiting the number of computation steps analysed, the number of times loops are unrolled [3,6,7].

In the case of Marlowe, we have natural bounds that we can use because, given a concrete contract, we can easily infer:

- The maximum number of inputs that can have an effect on the contract
- The maximum number of transactions that can have an effect on the contract
- All of the account, choice, and `Let` identifiers that will be used in the contract
- The number of participants that will participate in the contract

From this data, we can also deduce an upper bound for:

- The number of times that `computeTransaction`, `reduceContractStep`, and `applyInput` may be called.

– The number of accounts that the contract will use, and an the number of elements there may be in each of the associative maps that comprise the `State` of the contract at any given point of the execution.

If we keep the `Contract` parameter of the semantics concrete, we can use it to bound the execution, because every call to `reduceContractStep` will either make no progress or remove one of the constructs, with the exception of `Close`. But for `Close` we know we just have to call it as many times as the maximum number of accounts there can possibly be, since every time we call it it will refund one of the accounts.

Thus, the symbolic transaction processing function becomes:

```
computeTransaction :: SBV TransactionInput -> SBV State
                   -> Contract
                   -> SBV TransactionOutput
```

However, there is one more problem: the output `Contract` returned by the function, which is wrapped inside the `TransactionOutput`, is symbolic, since it depends on the current `TransactionInput` and `State`, which are both symbolic.

We get around this problem by using a continuation style. Instead of returning the `TransactionOutput` directly, we take a continuation function that takes the concrete `Contract` and the symbolic version of `TransactionOutput` without the `Contract`.

Thus, the symbolic transaction processing function will look something like:

```
computeTransaction :: SymVal a => SBV TransactionInput
                   -> SBV State -> Contract
                   -> (SBV TransactionOutput ->
                       Contract -> SBV a)
                   -> SBV a
```

In practice, we also include some extra information about bounds, and we make some other parts of `TransactionOutput` concrete, in addition to the `Contract`.

4 Making static analysis efficient

In this section, we explain the techniques that we mentioned in Section 1 in more detail, and we illustrate those techniques with examples from the static analysis implementation of Marlowe.

4.1 Lightweight modifications

Some of the modifications that we can apply to a static analysis implementation are localised, and they do not fundamentally affect the overall structure of the implementation. This means that the implementation can remain intuitive and close to the original semantics and model that we are trying to analyze, as well as making changes less likely to introduce errors than a more serious restructuring.

Removing unnecessary parts from the analysis. This is probably the most straightforward optimization to implement. When we use the same or similar code for both the analysis and the implementation, we may end up including code that is not relevant to the analysis.

In the case of Marlowe, this was the case of the `Close` construct. The `Close` construct refunds all the money and tokens remaining in the accounts. At the time of analysis, we do not know how much is available in each account, because it may depend on the value of a choice made by the user in runtime. For that reason, the generated constraints must include comparisons for every account. This makes the implementation of the `Close` construct very expensive to execute symbolically.

As it turns out, we do not need to implement the semantics of the `Close` construct at all, because it is impossible for the `Close` construct to produce a failed payment (we have also proven this[3]: it only pays whatever there is available. So we can safely remove it from the analysis.

Avoiding high level abstractions. High level libraries like SBV, and even standards like SMTLib, support the use and construction of considerably high level abstractions, e.g: custom data-types, list, sets... Unfortunately, even though high level abstractions make reasoning about code easier, they often prevent certain optimisations, since they abstract out aspects of the implementation that in our particular case may be concrete.

For example, in the case of Marlowe's static analysis, we initially implemented a symbolic associative map primitive with the only limitation that it needed a concrete bound in the number of elements. This is straightforward to realise using the symbolic implementation of list and tuple, both provided by SBV. However, this can be very costly: searching a list symbolically requires constraints that check the key of every element in the list up to the maximum capacity of the associative map, because it is not known at the time of symbolic execution how many elements the list has.

However, in Marlowe we know the values of all the keys that we are going to use, because the contract is concrete, and only `Account`, `Choice`, and `Let` identifiers that are mentioned in the contract will ever make it into the `State`. So we do not need keys of the associative map to be symbolic, we can use a concrete associative map with symbolic values.

Nevertheless, using a partially concrete map prevents us from using it as the result of a function that takes symbolic parameters, so we will need to use a continuation style insead. An alternative is to assign numbers to each possible key, and store them in a symbolic list, this will at least prevent the search from having to compare a lookup with every key, since we will know in which position of the list each value is stored.

[3] `https://github.com/input-output-hk/marlowe/blob/master/isabelle/CloseSafe.thy` using Isabelle [12]) (last visited on 04/04/2020)

Reducing search space by normalising parameters. The higher the number of degrees of freedom of the input, the larger the search space, and the higher the load we are putting on the SMT solver. In many cases, we do not need to explore the whole search space to check satisfiability of a property. In particular, we may be able to prove that some sets of inputs are equivalent in terms of the property whose satisfiability we want to test.

For example, in the case of Marlowe's static analysis, Marlowe allows several inputs to be combined in one single transaction. This functionality is important because each transaction requires the issuer to pay fees, so combining several inputs in a single transaction can translate into important savings. On the other hand, it also means that static analysis must consider many more possibilities, since the number of ways of partitioning inputs in transactions grows exponentially in the number of inputs.

However, we can devise a normal form for input traces, in which there is a maximum of one input per transaction. We only need to make sure that, for every trace, if it produces a warning, there exists a trace with only one input per transaction that also produces a warning. Using the automated proof assistant Isabelle [12], we have shown that, indeed, splitting transactions into transactions with single inputs and the same slot interval as the original transactions does not modify the effect of those transactions on a contract[4].

This optimization reduces the search space considerably, but transactions may still have no inputs, so it still leaves room for many possible ways of combining the inputs in transactions. We explain how we reduced the search space even further in Section 4.2.

4.2 Heavyweight modifications.

When optimising, if our solution is a local minimum, small changes to the parameters will not grant any improvement to the value function. For that reason, in this section, we explore ways of optimizing that may imply considerable rewriting of our properties, constraints, and static analysis implementation in general.

Reducing search space by using normalised execution paths relevant to the property. One approach we can use that helps improve static analysis efficiency is to implement the property thinking of the property we want to test instead of trying to make it as close as possible to the full implementation of the system. We do not even need to think about the representation of a counterexample (we will discuss that in the next section), but only in what are the conditions for the property to be false.

Unfortunately, having static analysis be very different from implementation makes the approach much more error prone, and in making an argument for the trustworthiness of the analysis there are many more assumptions that need to

[4] `https://github.com/input-output-hk/marlowe/blob/master/isabelle/SingleInputTransactions.thy` (last visited on 29/04/2020)

be made and reasoned about. In Section 5, we explore ways of mitigating this problem.

For example, the main property we want to check is whether there is any possible execution that produces a failed payment. Thus, we look at executions instead of looking at inputs. The most complicated construct in terms of executions is the `When` construct, since it allows for two transactions that are separate in time to have different effects depending on when they are issued, all other constructs will get resolved instantly in one way or another. But, without loss of generality, we can structure them as Figure 1 shows: we can conceptually break the contract tree into subtrees, where each subtree has a `When` construct as its root, with the exception of the subtree that has the same root as the original contract tree.

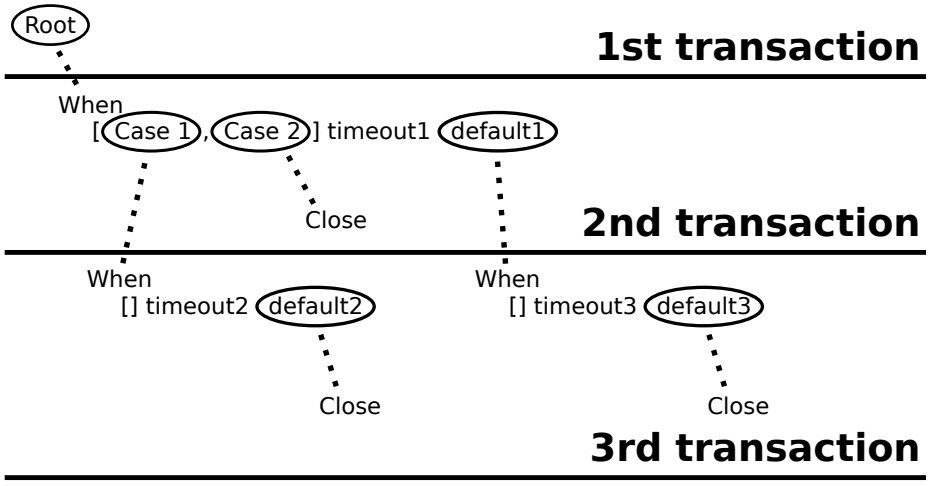

Fig. 1. Distribution of transactions with respect to a contract

Each level of subtrees will correspond to a transaction, i.e: the root subtree will correspond to the first transaction, the set of subtrees that are children of the first subtree (in the original tree) will correspond to the second transaction, and so on. There may be some paths which do not require as many transactions/subtrees as other paths, because they go across fewer `When` constructs before finding a `Close` construct.

We can assign one transaction to each path of execution before a `When`, because if the maximum slot number of the transaction is lower than the slot number in the timeout and the transaction has no inputs, then the execution will stop before the `When`, so we need to account for that possibility. In other words, if two parts of the execution path may have different slot numbers when we go through them, we need to consider them as separate transactions, because they may have payments that depend on those values.

On the other hand, if the minimum slot number of the transaction is greater or equal than the slot number in the timeout, then the first transaction will execute past the `When` and into the timeout branch, and we will not need the second transaction. In the same way, if we include an input that is allowed by the `When` into the first transaction, then the first transaction will continue into the corresponding `When` branch, and we will not need the second transaction.

We cannot just constrain the maximum slot number of the first transaction to be less than the timeout in the `When` because, if we do that, then we will miss executions where one transaction expires several `When` in a row, or where it expires one `When` and provides the right input for later `When`. And we cannot not constrain the maximum slot number of the first transaction because then we will not know whether the first transaction goes beyond the `When` or not.

We get around this problem by allowing the slot numbers of the first and second transactions to be equal. If this happens, we will find out that one of the transactions will be marked as `UselessTransaction` by the semantics when we look at the counterexample. So we just need to filter out all transactions that produce `UselessTransaction` warnings in the final result.

Note however that, in Section 4.1, we already reduced the number of inputs per transaction to a maximum of one, and the number of transactions to the maximum number of nested `When`s plus one. But now we have also assigned each of the transactions to a part of the contract, so we do not need a symbolic list of transactions, we can have a finite concrete list of symbolic transactions and we know in advance which of the transactions we are working on at each moment during the symbolic execution.

One subtle detail that we realised during testing is that, even though the diagram above shows that a `When` belongs to the new transaction, because the `When` will not be consumed by the previous transaction, `Observation`s in the `Notify` cases will correspond to the information there was in the `State` before the `When` was executed, with the exception of the slot interval, which will still be considered to be that of the transaction that triggers the `When` (the new transaction). The same is true for `Value`s in the `Deposit` cases, since the amount to deposit must be calculated without considering the effects of the deposit itself, we need to refer to the `State` before the deposit was made.

Minimizing the representation of inputs and outputs. When we initially implemented the efficient version of static analysis for Marlowe, we did not pay any attention to the inputs and outputs. The first version we wrote of the SBV property would not take any symbolic parameters. Originally, it would simply take a concrete `Contract` as input, and it would return a symbolic Boolean that determined whether the `Contract` was valid or not. However, if a `Contract` turns out to be invalid, we will also want to know why, so we later modified the property to give a counterexample that illustrated what went wrong. The original implementation still used some intermediate symbolic variables, but they were anonymous, and they were created during the exploration of the contract.

A simple way of obtaining a counterexample is to modify the output of the function to return the offending trace using the symbolic `Maybe` type. Surprisingly, this small change increases the amount of time required by the symbolic analysis severalfold.

The solution to this increase in execution time was to pass as input a fixed list of transactions, each of which being represented by a tuple with four symbolic integers:

1. An integer representing the minimum slot
2. An integer representing the maximum slot
3. An integer representing the `When` case whose input is being included in the transaction, where zero represents the timeout branch (and there being no inputs to the transaction).
4. An integer representing the amount of money or tokens (if the input is a `Deposit`), or the number chosen (if the input is a `Choice`)

If we do not need a transaction, we just set all four numbers to -1.

In order to translate this sequence of numbers into a proper list of transactions that is human readable and we can use to report the counter example, we need to iterate through the list and see the evolution of the contract with each transaction, using the concrete semantics. This way, we can gather the rest of necessary information, such as whether the transaction input is a `Deposit`. We can also see which of the transactions do not have an effect, i.e: they produce `UselessTransaction` as we mentioned in Section 4.2.

We use this separation of concerns between the static analysis and the concrete semantics to our advantage, as an opportunity for applying property based testing, as we explain in Section 5.1.

5 Testing for consistency and equivalence

Errors often occur when optimising or translating. If the static analysis implementation is close to the actual implementation, it is much less likely that a mistake will be made when writing it. On the other hand, if we have to craft a static analysis implementation that is equivalent (in some sense) to the original, but is written in a very different way, it is more likely that we will make assumptions which do not hold.

In order to avoid these problems, we combine the use of automated proof assistants to verify our assumptions and the use of property based testing to test for consistency and equivalence.

5.1 Testing for consistency

If our static analysis is not replicating all the functionality of the semantics, we can use potential discrepancies as an opportunity for testing, as shown in Figure 2. We generate random contracts and we apply the static analysis to them in order to try to find a counterexample that produces warnings. If we

cannot find any counterexamples then the test passes, but if we find one we test it on the semantics and see whether the counterexample indeed produces warnings in the semantics too, if it does not we have found a problem either on the static analysis or the semantics.

A limitation of this approach is that it only tests for false positives, and it does not prevent false negatives, those can be covered by testing for equivalence (see Section 5.2).

In addition, we can add assertions to the process. In the case of Marlowe, if the counterexample breaks one of the assertions or is formed incorrectly, it would also mean that there is a problem with the static analysis. For example, it may be that the counterexample refers to a `Case` of a `When` that does not exist, or that it has invalid or ambiguous intervals. If it has `UselessTransactions` is ok, because we are doing that on purpose, as we mentioned in Section 4.2.

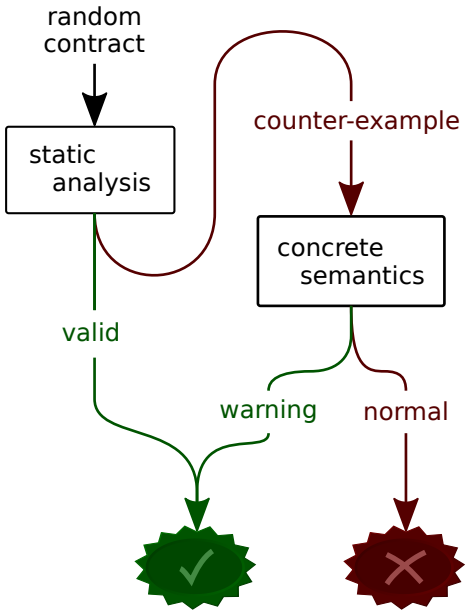

Fig. 2. Property based testing for consistency

5.2 Testing for equivalence

If we have two static analysis implementations, we have another opportunity for testing. In our case, we have one efficient implementation that is very different from the semantics and one inefficient implementation that is much closer to the semantics. We can do this as shown in Figure 3. We generate random contracts,

we feed them to both static analysis implementations, and we compare the results. If the results are the same then test passes, if they are different then one of the implementations is wrong.

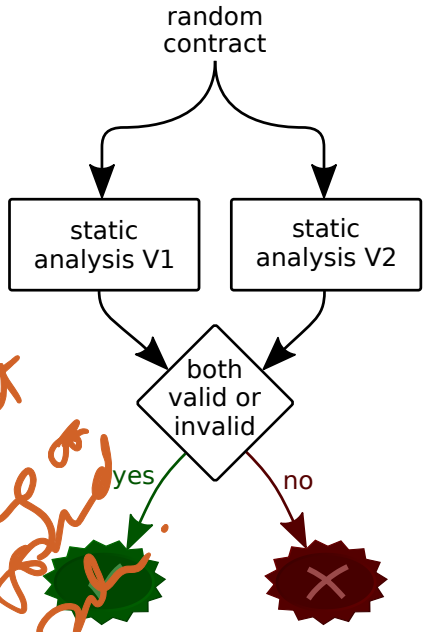

Fig 3. Property based testing for equivalence

This approach covers both types of errors, i.e: false positives and false negatives (for both of the implementations), but the execution time of the tests is bounded from below by the slower of the two implementations. The consistency approach is thus more efficient in finding false positives.

6 Measurements

In our experiments, the heavy weight optimisations considerably reduced the requirements of both processing time and memory for Marlowe's static analysis. In Tables 1, ~~2, 3,~~ and 4, we present the results of measuring the performance of static analysis on four example Marlowe contracts, before[5] and after[6] the heavyweight optimisations.

[5] `https://github.com/input-output-hk/marlowe/blob/master/src/Language/Marlowe/Analysis/FSSemantics.hs` [last visited 19-05-2020]

[6] `https://github.com/input-output-hk/marlowe/blob/master/src/Language/Marlowe/Analysis/FSSemanticsFastVerbose.hs` [last visited 19-05-2020]

Auction contract – `Auction.hs`

Lightweight optimisations					
Num. participants	1	2	3	4	5
Execution time	0.2205s	4m 45.2015s	N/A	N/A	N/A
Generation overhead	76.61%	96.40%	N/A	N/A	N/A
× execution time	0.1689s	4m 34.9342s	N/A	N/A	N/A
Z3 overhead	23.39%	3.60%	N/A	N/A	N/A
× execution time	0.0516s	10.2673s	N/A	N/A	N/A
RAM usage peak	44,248KB	1,885,928KB	N/A	N/A	N/A
Heavyweight optimisations					
Num. participants	1	2	3	4	5
Execution time	0.01198s	0.0289s	1.1138s	1h 5m 45.1377s	N/A
Generation overhead	16.27%	29.64%	24.09%	80.86%	N/A
× execution time	0.001949s	0.008566s	0.268314s	53m 10.038344s	N/A
Z3 overhead	83.73%	70.36%	75.91%	19.14%	N/A
× execution time	0.010031s	0.020334s	0.845486s	12m 35.099356s	N/A
RAM usage peak	17,020KB	17,740KB	49,668KB	2,364,500KB	N/A
Contract size (chars)	275	3,399	89,335	4,747,361	413,784,559

Table 1. Execution cost measurement of auction contract

Crowdfunding contract – `CrowdFunding.hs`

Lightweight optimisations					
Num. participants	1	2	3	4	5
Execution time	0.6298s	50m 53.4597s	N/A	N/A	N/A
Generation overhead	85.02%	99.82%	N/A	N/A	N/A
× execution time	0.5356s	50m 47.9635s	N/A	N/A	N/A
Z3 overhead	14.98%	0.18%	N/A	N/A	N/A
× execution time	0.0943s	5.4962s	N/A	N/A	N/A
RAM usage peak	111,980KB	5,641,056KB	N/A	N/A	N/A
Heavyweight optimisations					
Num. participants	1	2	3	4	5
Execution time	0.0125s	0.041s	1.0515s	32m 15.4478s	N/A
Generation overhead	16.68%	25.36%	37.23%	69.83%	N/A
× execution time	0.0021s	0.0104s	0.3915s	22m 31.5232s	N/A
Z3 overhead	83.32%	74.64%	62.77%	30.17%	N/A
× execution time	0.0104s	0.0306s	0.6600s	9m 43.9246s	N/A
RAM usage peak	17,016KB	18,768KB	62,108KB	3,715,020KB	N/A
Contract size (chars)	857	12,704	364,824	19,462,278	1,690,574,798

Table 2. Execution cost measurement of crowdfunding contract

Rent contract – `Rent.hs`

Lightweight optimisations					
Num. months	1	2	3	4	5
Execution time	0.2850s	3.0303s	2m 53.2458s	3h 22m 13.0122s	N/A
Generation overhead	77.55%	91.25%	99.18%	99.94%	N/A
× execution time	0.2210s	2.7651s	2m 51.8252s	3h 22m 5.7324s	N/A
Z3 overhead	22.45%	8.75%	0.82%	0.06%	N/A
× execution time	0.0640s	0.2652s	1.4206s	7.2798s	N/A
RAM usage peak	42,052KB	221,960KB	1,237,092KB	9,160,616KB	N/A
Heavyweight optimisations					
Num. months	1	2	3	4	5
Execution time	0.0114s	0.0111s	0.01132s	0.01124s	0.01255s
Generation overhead	11.55%	11.95%	13.65%	13.69%	21.00%
× execution time	0.0013s	0.0013s	0.0015s	0.0015s	0.0026s
Z3 overhead	88.45%	88.05%	86.35%	86.31%	79.00%
× execution time	0.0101s	0.0098s	0.0098s	0.0097s	0.0099s
RAM usage peak	15,536KB	15,364KB	15,400KB	15,364KB	15,540KB
Contract size (chars)	339	595	852	1,109	1,366

Table 3. Execution cost measurement of crowdfunding contract

Coupon bond contract - ACTUS.hs

Lightweight optimisations					
Num. months	2	3	4	5	6
Execution time	0.8293s	5.8887s	1m 35.3930s	26m 36.4585s	9h 50m 22.3418s
Generation overhead	76.57%	74.91%	89.29%	72.34%	76.66%
× execution time	0.6350s	4.4112s	1m 25.1764s	19m 14.8781s	7h 32m 34.7672s
Z3 overhead	23.43%	25.09%	10.71%	27.66%	23.34%
× execution time	0.1943s	1.4775s	10.2166s	7m 21.5804s	2h 17m 47.5746s
RAM usage peak	74,180KB	209,724KB	940,924KB	3,283,384KB	13,483,908KB
Heavyweight optimisations					
Num. months	2	3	4	5	6
Execution time	0.0092s	0.0095s	0.0097s	0.0102s	0.0105s
Generation overhead	14.51%	15.50%	15.51%	19.71%	19.69%
× execution time	0.0013s	0.0015s	0.0015s	0.0020s	0.0021s
Z3 overhead	85.49%	84.50%	84.49%	80.29%	80.31%
× execution time	0.0079s	0.0080s	0.0082s	0.0082s	0.0085s
RAM usage peak	15,636KB	15,816KB	15,788KB	15,780KB	15,760KB
Contract size (chars)	479	635	791	947	1,103

Table 4. Execution cost measurement of coupon contract

Unfortunately, at the time of writing, we do not have a completely unoptimised version of the static analysis that we can use to compare, because the semantics of Marlowe have changed since we implemented such a version.

Using the `perf` tool [1], we have measured the execution time and the overhead of both the generation of the constraints and their solution by Z3 [4]. And we have measured the peak RAM usage of the whole process by using GNU's `time` tool [8].

In all the contracts, the implementation that has the heavyweight optimisations performs much better and scales further. In the case of the *auction* (Table 1) and *crowdfunding* (Table 2) contracts, whose size grows exponentially, both approaches quickly overwhelm the resources available in the execution environment. On the other hand, in the case of *rent* (Table 3) and *coupon bond* (Table 4) contracts, which grow linearly, by using the lightweight version, the problem becomes intractable much faster than with the heavyweight version.

Another conclusion that can be derived from the experiments is that, in the version with lightweight optimisations, most of the processing time seems to be spent during the generation of the constraints, and solving is done relatively quickly by Z3; while the opposite happens for the version with heavyweight optimisations.

These results suggest that SBV library is able to generate constraints in a way that they are handled efficiently by Z3, but the process itself can be costly if we are not careful. Nevertheless, the overall execution time required by Z3 is also lower in the case of the version with heavyweight optimisations and it grows more slowly, which suggests the optimisations described in this paper affect both stages of the process to some extent.

7 Related work

Work in [11] documents a similar effort to ensure correctness of implementations by using Haskell and SBV library; the authors also discuss performance of the analysis and apply this approach to non-functional requirements.

The idea of using constraint solvers for finding bugs is not new, and there has been a number of initiatives that have explored its application to the verification of assertions in programs written using general purpose programming languages [6,7]; as well as for the compliance with usage protocols [2,13].

[9] also applies constraint solvers for detecting problems in the usage of DSLs. The authors observe that SMT solvers have limited support for non-linear constraints such as exponentiation. This problem does not affect the current design of Marlowe because it does not support multiplication by arbitrary variables, and because all inputs are integer and bounded finitely.

8 Future work

In the future, we would like to extend static analysis to cover other potential problems in Marlowe contracts and to aid their development. We can envisage

that we can use static analysis to locate unreachable subcontracts, to allow developers to provide custom assertions and check their satisfiability, and even to allow users to inspect the possible maximum and minimum values that particular expressions can reach.

9 Conclusion

In this paper, we have summarized the lessons learned from our work on optimising the static analysis for Marlowe contracts. We have found that there are two clear approaches to static analysis using SMT, both have advantages and disadvantages. One is less error prone and straightforward, but inefficient and hard to test; the other approach is much more efficient, versatile, and testable, but more error prone.

We have also seen that many specific properties and restrictions characteristic of the target DSL can be utilized both as optimisation opportunities and, in our case, for completeness of the analysis. Symbolic execution of, for example, a turing complete language, would be intractable, and would require us to manually set a bound; but this is not in the case for Marlowe.

In the end, we have illustrated how to counteract the main disadvantage of the later approach, its propensity to errors, by using property based testing. This way we have obtained a static analysis implementation that is efficient, versatile, testable, and reliable.

On the other hand, for the static analysis of Marlowe contracts, we found out that when running statistics on the equivalence testing property, most of the bugs were false negatives in the straightforward implementation, and the optimised implementation seems to be more reliable thanks to the consistency tests that we run beforehand.

Another advantage of the optimised implementation is that, because it relies on fewer and simpler features, it is compatible with more SMT solvers which, in turn, means that it is less reliant on the correctness of a single SMT solver. If an SMT solver fails to give an answer, we can try a different one; if we want to ensure that a contract is valid, we can test it with several SMT solvers and ensure all of them agree.

References

1. Performance analysis tools for linux. `https://github.com/torvalds/linux/tree/master/tools/perf` [last visited 20-05-2020]
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: International SPIN Workshop on Model Checking of Software. pp. 103–122. Springer (2001)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)

4. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
5. Erkök, L.: Sbv: Smt based verification in haskell (2019)
6. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 281–292 (2008)
7. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. ACM SIGSOFT Software Engineering Notes **25**(5), 14–25 (2000)
8. Keppel, D., MacKenzie, D., Juul, A.H., Pinard, F.: GNU time tool. `https://www.gnu.org/software/time/` [last visited 20-05-2020] (1998)
9. Keshishzadeh, S., Mooij, A.J., Mousavi, M.R.: Early fault detection in dsls using smt solving and automated debugging. In: International Conference on Software Engineering and Formal Methods. pp. 182–196. Springer (2013)
10. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: implementing and analysing financial contracts on blockchain
11. Mokhov, A., Lukyanov, G., Lechner, J.: Formal verification of spacecraft control programs (experience report). In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell. pp. 139–145 (2019)
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
13. Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug detection. In: International Conference on Computer Aided Verification. pp. 139–143. Springer (2005)