1 Model Extraction

Model M of the original contract. We take the HTLC contract which is written in Solidity as an example. Solidity consists of one or more internal state variables (denoted as $v \in V$) and external methods ($f \in F$) that changes the state.

An invocation of a method $f(\bar{x})$ is typically implemented as follows. It first checks guard conditions on V and $\bar{x}$ of the method. If they are not satisfied, method body will be executed. Otherwise, the invocation would be reverted. Thus state variable of M can be extracted from the conditions.

Here shows the target HTLC contract. It has three main methods: *newContract*, *withdraw*, and *refund*. For example, *withdraw* method has four guard conditions:

```
haveContract(_contractId)
contracts[_contractId].receiver == msg.sender
ontracts[_contractId].withdrawn == false
contracts[_contractId].timelock > now
```

Since the first condition performs access control, which can be ignored in transpilation process. We can extract atomic propositions in M from the second and third conditions, where `locked == 1` iff .

2 Transpilation of Program

This section provides transpilation of the original program P and the enforced model M‘ into a transpiled program P'. (Here we assume that the program consists of Ethereum contracts written in Solidity and client program in Javascript.)

In our example, M' is a composition of two HTLC contracts (each for Security and Cash respectively). Thus we need to specialize the original HTLC contract for . As described in section ??, an action a in the enforced HTLC composite model M' are $C.f$, where C is, which is either `Sec` or `Cash`, a name of the ledger, and f is a method name.

2.1 Transpilation of Contract

Here we sketch transpilation the HTLC contract for the ledger C_0. We need to add a state variable `state` to the contract. We also add to each method a pre- and post-condition for the `state`. In Solidity it can be implemented using *modifier* mechanism. We take an transition $q_i \xrightarrow{C.f} q_{i+1}$ of M' (which means that the HTLC contract has a method $f()$), which is transpiled as follows.

(1) If $C \neq C_0$, add a method `function` C_f`(){}` to the transpiled contract. Its pre-condition would be `state ==` q_i and post-condition would be `state ==` q_{i+1}. This method just change the contract's state and does nothing else.

(2) If $C = C_0$, there are two subcases.

(a) If the execution of f branches, i.e., there is two following events on $C.f$; $q_i \xrightarrow{C.f} q_{i+1} \xrightarrow{C.f\_end} q_{i+2}$ and $q_i \xrightarrow{C.f} q_{i+1} \xrightarrow{C.f\_err} q'_{i+2}$, the original method f is converted to $C\_f()$. $C_f()$ first checks state pre-conditon and change the current state to q_{i+1}, then executes the body of $f()$. If it succeeds, it emits an event $C.f\_end$ and change its state to q_{i+2}. If the execution fails, it emits an event $C.f\_err$ and change its state to q'_{i+2}.

(b) If there's no such events, we simply transpile $f()$ of the contract into $C\_f()$, whose pre-condition on `state` is q_i and post-condition is q_{i+1}.

2.2 Transpilation of Client code

Transpilation of the client code is similar. Note that M' is specialized into two contracts on the separated ledgers. Thus the transpiled client need to synchronize states between both contracts.

Testcase	A. Original		B. Wrapper		C. Transpiled	
	Result	Gas	Result	Gas	Result	Gas
(1) Normal (both withdraw)	success	$450,928$	success	$450,928$	success	$661,473$
(2) Normal (both refund)	success	$404,354$	success	$404,354$	success	$695,941$
(3) Abnormal (Non-termination)	not detected	$372,568$	detected	$372,440$	detected	$664,855$
(4) Abnormal (Invalid state)	not detected	$427,641$	detected	$372,568$	detected	$661,775$
(5) Abnormal (TX. disorder)	error	$294,080$	detected	$294,080$	detected	$362,412$

Fig. 1 Evaluation result.

Assume that the original code includes TX invocation $C.f()$, where C is a ledger (or contract) name. We transpile it to invocations of methods the transpiled contracts $C.C\_f(); C'.C\_f()$, where C' is a name of the other ledger than C. If the action corresponds to the case (2b), that completes the transpilation. If it's the case for (2a), additional event handling is needed. The invocation of $C.C\_f()$ emits an event ev which is either $C.f\_end$ or $C.f\_err$. The transpiled client should subscribe the event and invoke the method of C' which corresponds to it. For example, if it received $C.f\_end$, it should call $C'.C\_f\_end()$.

3 Evaluation

(Maybe description about each implementation somewhere?)

We provide five testcases. In the testcase (1), both Seller and Buyer successfully withdraw assets. Both B and C implementations go through valid states ($q_2 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_8 \rightarrow q_{10} \rightarrow q_{12}$). But the C consumes more gas due to overhead of implementing state machines in contracts. Similarly, testcase (2) runs without errors by all implementation ($q_2 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_7 \rightarrow q_9 \rightarrow q_{11} \rightarrow q_{13} \rightarrow q_{12}$).

In testcase (3), Buyer fails to withdraw and terminate the protocol. Implementation B and C detects that protocol terminates errornous state, which A cannot detect. (TBW: how to write?)

In testcase (4), Buyer fails to withdraw and then Seller try to refund the security. It would result in the situation where both security and cash belong to the Seller. While implementation A pass the refund TX, B and C properly reject the refund. Because when the withdraw fails, the composite state machine is at q'_{12}. At the state `Sec.refund` transition is not allowed.

In testcase (5), `Cash.refund` is called right after `newContract` transactions. In this case, TX invocation in A results in an error since it does not reach timeout. On the other hand, B and C rejects to invoke `refund` due to they are at an invalid state to call the transaction.