On the specification and verification of atomic swap smart contracts[*]

Ron van der Meyden
meyden@cse.unsw.edu.au

UNSW Sydney

Abstract. Blockchain systems and smart contracts provide ways to securely implement multi-party transactions without the use of trusted intermediaries, which currently underpin many commercial transactions. However, they do so by transferring trust to computer systems, raising the question of whether code can be trusted. Experience with high value losses resulting from incorrect code has already shown that formal verification of smart contracts is likely to be beneficial. This note investigates the specification and verification of a simple form of multi-party transaction, *atomic swaps*. It is argued that logics with the ability to express properties of strategies of players in a multi-agent setting are conceptually useful for this purpose, although ultimately, for our specific examples, the less expressive setting of temporal logic suffices for verification of concrete implementations. This is illustrated through a number of examples of the use of a model checker to verify atomic swap smart contracts in on-chain and cross-chain settings.

1 Introduction

Many commercial transactions presently make use of intermediaries, independent of the interacting parties. The role of these intermediaries is to overcome deficits of trust between the transacting parties, ensuring that none are able to act in a way that causes the others losses due to deviations from agreed terms of exchange.

One of the main potential benefits of blockchain technology is its ability to eliminate the use of trusted intermediaries in multiparty interactions, enabling parties to engage in reliable peer-to-peer transactions without having to place trust in third parties. The ability to ensure atomicity of transactions composed of multiple independent exchanges is one significant example of this capability.

While blockchain and smart contract solutions have the attractive property of being able to enforce such atomic transactions, they do come with risks of their own. Trust placed in a person is not in fact eliminated, rather, it is replaced by trust in the correct behaviour of the smart contract code. In effect, the fact that there is an intermediary in the transaction is not changed, but a human intermediary is replaced by an intermediary composed of smart contract code executing on a blockchain platform.

---

[*] Version of Sep 10, 2018.

Trust in code is not always warranted: code can be complex, and behave in unexpected ways. This is particularly the case when, as in the case of smart contracts on the blockchain, code is open to inspection, and malicious agents may interact with it. There have already been a number of instances where large losses have been incurred as a result of poorly written smart contracts, that were vulnerable to attacks enabling the attackers to either steal assets [Pop16], or make them unavailable [Ltd17] to the rightful owners. There is therefore a need for smart contracts, which may control large amounts of monetary value, to be subjected to rigorous quality assurance before trust is warranted.

The strongest form of quality assurance used in computer science uses *formal methods*, involving the use of precise, mathematically grounded models to rigorously represent specifications and implementations, and to *prove* that the implementations meet the specifications. To ensure that the reasoning in these proofs is correct, they are represented in formal logics, and reasoning steps are checked (or even automatically constructed) using software tools. While such tools may themselves contain errors, the level of attention to detail that they provide is far beyond human capacity for tedious checking of large volumes of reasoning steps, and the methodology has been highly successful in identifying errors that had escaped human attention.

There is significant interest in the blockchain community in the application of formal methods to blockchain systems and smart contracts. The first step in such an application is to develop mathematically precise specifications of the desired behaviour and properties of the systems to be developed. In this note, we conduct a case study of one of the simplest cases of a smart contract, in order to develop an understanding of the types of specifications that are required in this area.

Specifically, we consider the case of an atomic swap of digital assets. The scenario is very simple: Alice and Bob each own an asset, and they would like to exchange ownership. We explain this scenario informally, and sketch code for a simple smart contract implementing an atomic swap in Section 2. We then consider a number of potential approaches for formal specifications against which such code might be judged for correctness. In Section 3 we consider the suitability of pre-condition/post-condition specifications. Section 4 argues that a richer form of specification captures the intended behaviour better, by making use of logics that contain constructs relating to *strategies* played by agents in a multi-agent setting. We argue that these constructs provide a conceptually useful high level language in which to express issues of concern in smart contract specification, but also that they have some limitations, and may ultimately be eliminable when it comes to verifying concrete implementations.

We illustrate this in Section 5, where we show that it is possible to automatically verify that a concrete implementation of atomic swap satisfies its specification using just temporal logic model checking, without making use of strategic operators. In Section 6, we go on to apply the methodology of Section 5 to a harder problem: cross-chain atomic swaps, in which the assets to be

swapped reside on different blockchains. Section 7 concludes with a discussion of questions for future consideration and related work.

2 Atomic Swaps

Suppose Alice holds an asset a and Bob holds an asset b. They would like to execute a swap, in order to reach a state where Alice holds b and Bob holds a. The problem that this presents is that one of the parties may not behave as expected. If Alice first transfers a to Bob, then she is at risk that Bob will not uphold his end of the bargain, and transfer b to Alice. Alice would then face the loss of a without compensation. There is a similar problem if Bob first makes the transfer to Alice. (Alice and Bob's recourse to the courts to enforce the exchange contract may not be effective in a blockchain setting, where Alice and Bob may know each others' identity only via their public keys, in which case they would not even know who should be served with a legal claim.)

One commercially applied solution to this problem is for Alice and Bob to make use of an intermediary known as an "escrow agent". We will call this agent "Esther" here. Alice and Bob each transfers their asset to Esther, and once she has received both a and b, Esther transfers b to Alice and a to Bob. This solution no longer requires that Alice and Bob trust each other, but it does require that both trust that Esther will act in their best interests. Were she malicious, Esther could cause many forms of harm. She could abscond with both Alice and Bob's assets herself, transfer a to Bob before receiving b from Bob (enabling Bob to abscond with both assets), or transfer just one of the assets and retain the other herself. Other more subtle forms of malice might be possible on Esther's part, like retaining the assets for an extended period before transferring them in order to collect part of their revenue streams for herself. This solution is only effective if Alice and Bob can justifiably believe that Esther is not malicious in any of these ways. Finding a person who can be trusted in this way may be a challenge, particularly if Alice and Bob live far from each other. Even if they can find an appropriate Esther, she is only human, so might still behave maliciously even if she has been trustworthy in the past.

The promise of blockchain and smart contract technology for this problem (at least for digital, or digitally represented, assets) is that they enable Esther to be replaced by a computer program that is in-corruptible and does not act with malicious intent. Both Alice and Bob can carefully inspect this program and form the justified belief that it behaves in exactly the way that they were expecting of Esther. If this program runs on blockchain platform that Alice and Bob trust will reliably execute this programs exactly according to its specified semantics, then they can be satisfied that their transaction will be executed without risk of loss.

Figure 1 shows the code for such an "escrow" smart contract. Rather than use any specific smart contract programming language, we write psuedo-code. Intuitively, the code consists defines a collection of functions that may be called by any agent. The function `initialise` is called automatically when the smart

contract is initiated on the blockchain, and sets up initial values of the local variables `depositedA`, `depositedB`, which are intended to represent whether Alice and Bob, respectively, have transferred the expected asset to the smart contract. We assume that `initialise` can only be called at initiation, and has null effect if anyone attempts to call it at any other time.

```
Contract Escrow
{
depositedA, depositedB : Bool

initialise { depositedA := False ;  depositedB := False }

depositA  { if sender = Alice and value = a then depositedA := True }

depositB  { if sender = Bob and value = b then depositedB := True }

finalise { if depositedA and depositedB
              then { depositedA := False; depositedB := False;
                      send(a,B); send(b,A) } }

cancelA { if depositedA then { depositedA := False; send(a,A) }  }

cancelB { if depositedB then { depositedB := False; send(b,B) }  }
}
```

Fig. 1. An atomic exchange smart contract

The function `depositA` allows Alice to transfer the asset a to the smart contract. The keyword `sender` denotes the agent who is calling a function. We assume that a function call may be made with an asset attached - the effect of attaching an asset with a function call is for the asset to be transferred to the smart contract. The smart contract code may check the value of the attached asset using the keyword `value`. Thus, the code for `depositA` checks that the function has been called by Alice and has asset a attached. If so, then Alice is credited with the deposit by setting `depositedA` to be true, otherwise this function call has no effect. There is a similar function `depositB` allowing Bob to deposit his asset.

If all goes well, and both assets have been deposited, then the function `finalise` may be called by any agent (but presumably, will be called by either Alice or Bob, since noone else stands to gain anything by doing so) and has the effect of transferring the two assets back to Alice and Bob in such a way as to effect the swap.

It may happen however, that Alice deposits her asset, but Bob does not. This would be a problem for Alice, since the asset is then no longer under her control,

but under the control of the smart contract, but she has received nothing in return. To protect Alice against the eventuality that Bob does not deposit his asset (or does not do some in a timely fashion), there is a function `cancelA` that enables Alice to cancel her participation in the exchange and recover her asset a. A similar function is available to Bob.

We abstract cryptographic concerns from our presentation - typically Alice and Bob would be represented via their public keys, and their instructions to perform actions in the smart contract would be sent as cryptographically signed messages. Note that blockchain systems do not provide guarantees with respect to the order of execution of messages submitted for processing by the agents - this is under the control of network operators (who are typically called "miners" in the area.) Smart contract code is typically *reactive*, i.e., the state of the smart contract changes only when some agent (or another smart contract) sends it a message - there are no spontaneous events such as events triggered by timers.

The escrow smart contract is relatively short, but it already displays some complexity, with several functions maintaining an internal state that consists of assets held by the contract, as well as variables used for book-keeping of those assets. One might wonder if there are any unexpected interactions between these functions, and whether, in combination, they capture the possible behaviours that Alice and Bob expect of their interaction. The code already has some implicit invariants that underly its correctness. For example, why it is safe for the send operations to be performed in `finalise`, `sendA` and `sendB`? The reason is that the functions interact in such a way as to ensure that the following statement is an invariant that holds in all states reachable from any sequence of calls: if `depositedA`, then the smart contract holds asset a, and similarly, if `depositedB`, then the smart contract holds asset b. Even if one considers this obvious for this contract, for even a marginally more complex contract, it would be desirable to have an independent *specification* against which the code could be evaluated for correctness.

3 Pre-condition/Post-condition specifications

One commonly used form of formal specification of code is the use of *pre-conditions* and *post-conditions*. In the *refinement calculus* [Mor94,BvW98] notation, these can be written using statements in the form $[pre, post]$, where *pre* and *post* are formulas that assert some property of the state of the system. The intuitive meaning of this statement is that it is a specification of a program. A concrete program P *implements* (or *refines*) this statement if, whenever it is executed from an initial state satisfying the pre-condition *pre*, at termination the program is in a state satisfying the post-condition *post*. There are weak and strong version of this semantics: in the weak (partial completeness) semantics, termination is not guaranteed, in the strong, the program is required to terminate whenever started at a state satisfying ϕ.

It would appear at first that this specification approach can be used to specify the Escrow smart contract. Write $\texttt{holds}(x, y)$ to mean that agent x holds asset

y. Then (abbreviating Alice and Bob respectively as A and B) we could try to specify the goal of Alice and Bob's exchange by the statement

$$[\texttt{holds}(A,a) \wedge \texttt{holds}(B,b),\ \texttt{holds}(A,b) \wedge \texttt{holds}(B,a)] \qquad (1)$$

which is satisfied by a program that (provided it is started in the correct initial state) guarantees on termination that the assets have been swapped.

Unfortunately, this specification is too strong to be implemented in a blockchain setting. Pre-condition/Post-condition specifications are applicable to settings with a single "thread", or locus of control, and this is not the case on the blockchain, where we need to consider not only the execution of smart contract code, but also the actions of other agents in the system, in this case Alice and Bob. Evidently, action by both Alice and Bob is required to bring the smart contract of Figure 1 to the desired final state, and there is nothing that the contract can do on its own to ensure that Alice and Bob each execute their part of the deal. Indeed, in order to apply precondition/post-condition specifications, we first need to clarify what we mean by termination in such a multi-agent setting. For the moment, we take a terminated state to be one reached by executing an action called "finalise", and assume that the other actions are disabled after this function has been called. (This can easily be programmed into the smart contract by adding a variable `done` that is initially false and set true by `finalise` when this is called in a state where `depositedA` and `depositedB` hold.)

We could attempt to interpret the specification (1) with the weak (partial completeness) semantics in order to cover the possibility that Alice or Bob does not play their part, and treat such a behaviour as a case of non-termination. However, this is also unsatisfactory. Consider a variant of the contract of Figure 1, in which we delete the actions `cancelA` and `cancelB`. This satisfies the weak interpretation of specification 1, since now the only way to reach the final state is for both Alice and Bob to deposit their assets and then call `finalise`. Failure by either Alice or Bob to play their part is treated as non-termination, and the specification has nothing to say about this eventuality. However, from Alice and Bob's perspective, this is a fatal deficiency of the specification: this variant of their contract allows a run in which Alice deposits her asset, but Bob does not, with the effect that Alice has forever lost control of her asset, without compensation.

Since it seems to be necessary to accommodate the possibility that the implementation terminates without the exchange having occurred, one might be tempted to weaken the specification to the following

$$\left[\texttt{holds}(A,a) \wedge \texttt{holds}(B,b),\ \begin{array}{l} (\texttt{holds}(A,b) \wedge \texttt{holds}(B,a)) \\ \vee (\texttt{holds}(A,a) \wedge \texttt{holds}(B,b)) \end{array} \right] ,$$

which allows that on termination, the system has reverted to its original state. However, this also does not capture our intent for the system. There is a very simple program ($skip$) which implements this specification just by doing nothing! Clearly, this is not what Alice and Bob intend: they do wish to engage in the exchange, but to do in a way that protects them from malicious behaviour by the

other. At the very least, some additional constraint would need to be included in the formal specification to capture this. It is not at all clear how to express Alice and Bob's preference within a pre-condition/post-condition specification.

4 Logics with Quantification over Strategies

Temporal logics [Eme90] contain operators that express properties of a world whose state changes over time. They have been found useful for specifying and reasoning about concurrent systems, and are widely used for verification of such systems, particularly in the context of computer hardware development. There are several variants of such logics: *linear time* logics are concerned with what happens in a single history, and *branching time* logics take the view that time has a tree-like structure, with multiple *possible* futures at each instant.

Specification of concurrent settings involving multiple, possibly adversarial agents has been addressed in some types of logic called *alternating temporal logic* [AHK02], and the more general *strategy logic* [CHP10]. These are extensions of branching time temporal logics. The semantic setting for these logics is an environment, called a *concurrent game structure* in the literature, in which some set of agents are each equipped with a set of actions that they may perform. At each moment of time, agents that are scheduled to act select one of their actions to perform, and a concurrent game structure describes how the selected actions cause a state transition.

A *strategy* for an agent is a rule for selecting an action of the agent in each possible state of the system. For example, the set of conditions in Figure 2 give a strategy for Alice in the environment created by the atomic exchange contract of Figure 1. Here the construct `do...od` is a nondeterministic loop statement, consisting of a set of clauses $C \rightarrow A$. The statement executes by repeatedly performing one of the actions A for which the condition C is true. If there are several possible such actions, the choice is made non-deterministically.

```
do
  holds(A,a) ->  depositA;
  depositedA and depositedB -> finalise;
  otherwise -> skip
od
```

Fig. 2. A strategy for Alice

Alternating temporal logic introduced a specification construct $\langle\!\langle G \rangle\!\rangle \phi$, in which G is a set of agent names, and ϕ is formula. Intuitively, the formula $\langle\!\langle G \rangle\!\rangle \phi$ says that the group of agents G has a strategy that guarantees, whatever the other agents in the system do, that the formula ϕ will hold. The formula ϕ states

some property of the future. For example, ϕ could be the formula `Eventually` ψ, which holds if ψ is true at some time in the future.

Using this construct, we can capture more of the requirements discussed above. Write I for the initial condition $\mathtt{holds}(A,a) \wedge \mathtt{holds}(B,b)$ and F for the desired final condition $\mathtt{holds}(A,b) \wedge \mathtt{holds}(B,a)$. Then the formula

$$\langle\!\langle A, B \rangle\!\rangle (I \Rightarrow \mathtt{Eventually}\ F)$$

says "Alice and Bob have a strategy that, if started in a state satisfying I, guarantees that (whatever any other agent does) eventually F."

We can also express the idea that Alice should be able to recover her asset by some means (even if Bob is not cooperating) using the formula

$$\langle\!\langle A \rangle\!\rangle \mathtt{Eventually}\ \mathtt{holds}(A,a)\ .$$

This says "Alice has a strategy that ensures (whatever any other agent does) that eventually Alice holds asset A". Obviously, this would be true trivially (without Alice having to do anything) at any time that Alice already holds a, but it also ensures that Alice has some course of action that guarantees that she recovers a in case, e.g., a has been placed under the control of a smart contract.

However, this latter requirement may not be realisable. In a blockchain implementation of the smart contract of Figure 1, Alice's strategy for recovering her asset (after she has already called `depositA`), would be to send message invoking action `cancelA`. This message will take some time to spread through the miner network, and it may take some additional time before a miner chooses to incorporate it into a block. In the mean time, Bob may have been able to call `finalse` (preceded by `depositB`), and effect the swap. Miners may have chosen to prioritise Bob's messages. This means that the action actually under Alice's control (sending the request) is not guaranteed to eventually make $\mathtt{holds}(A,a)$ true. However, in the event that Bob "wins" this race, the contract does result in $\mathtt{holds}(A,b)$ being true. From Alice's point of view, this was her initial objective in doing business with Bob, so she should be satisfied, even if her attempt to cancel failed. (The point of the cancellation facility is to prevent her asset being locked in the contract as a result of Bob's potential malice,) This suggests that the weaker formula

$$\langle\!\langle A \rangle\!\rangle \mathtt{Eventually}\ (\mathtt{holds}(A,a) \vee \mathtt{holds}(A,b))$$

is a more satisfactory way to express Alice's ability to recover from malicious activity by Bob: it may be paraphrased as "Alice may recover *an* asset". A symmetric formula

$$\langle\!\langle B \rangle\!\rangle \mathtt{Eventually}\ (\mathtt{holds}(B,b) \vee \mathtt{holds}(B,a))$$

expresses Bob's ability to recover an asset. We write $Recoverable_A$ and $Recoverable_B$ for these formulas, respectively.

Let us now combine these pieces, with some qualification. We do not require Alice and Bob to have the ability to recover their assets in all possible situations.

For example, if Alice decides not to cooperate with Bob, and transfers her asset to Carol instead, then she then has no right to expect to recover her asset. Alice needs to be protected from Bob's malice only so long as she is engaged in her interaction with Bob. This suggests combining our requirements into a single formula as follows:

$$\langle\!\langle A, B \rangle\!\rangle \left(I \Rightarrow \left(\begin{array}{l} \texttt{Eventually } F \;\wedge \\ (Recoverable_A \;\wedge\; Recoverable_B) \;\; \texttt{Awaiting } F \end{array} \right) \right)$$

We use here a binary temporal construction α `Awaiting` β, the meaning of which is that formula α continues to hold up to the moment that formula β holds. It allows that β never holds, in which case α holds forever.[1]

However, this formula is still not quite what we want. It says that Alice and Bob retain the ability to recover an asset, so long as both are playing the strategy intended to eventually result in the final condition F, up to the moment that this final condition holds. But what of a situation where Alice plays her part, but Bob is malicious from the start? Here we are not in the scope of a situation (covered by the operator $\langle\!\langle A, B \rangle\!\rangle$) where both Alice and Bob cooperate. The formula is therefore silent on this situation, and leaves it completely open whether Alice can recover if this happens. In attempting to constrain the situations ensuring recovery, we have gone too far. What we need instead is that Alice is protected from all possible behaviours of Bob, just based on the fact that Alice has been playing her part to cooperate with Bob.

To express this, we seem to need a richer language with the expressiveness to name stategies. This expressiveness was introduced in a class of logics known as strategy logic [CHP10]. This class of logics contain operators similar to the following.[2]

- $\exists \alpha \!:\! p(\phi)$, expressing that there exists a strategy (named α) for player p, such that formula ϕ holds.
- $\Box \phi$, expressing that whatever strategy any player plays from the present state onwards, ϕ holds,
- p `plays` α, expressing that from the present moment onwards, player p plays strategy α.

Note that, when G is the set of agents $\{p_1, \ldots, p_n\}$, the alternating temporal logic formula $\langle\!\langle G \rangle\!\rangle \phi$ is equivalent to

$$\exists \alpha_1 \!:\! p_1 \ldots \exists \alpha_n \!:\! p_n \Box ((\bigwedge_{i=1\ldots n} p_i \texttt{ plays } \alpha_i) \Rightarrow \phi) \;.$$

---

[1] Expressing this in linear time temporal logic, α Awaiting β is $(\alpha \texttt{ Until } \beta) \vee \texttt{Always } (\alpha \wedge \neg\beta)$, which says that either α `Until` β (meaning β holds at some time in the future, and α holds at every moment up until that time), or `Always` $(\alpha \wedge \neg\beta)$ (meaning that α is true and β false, now and at all times in the future). We do not use the stronger formula α `Until` β here to allow for easier comparison with a later formula, which has contexts where we cannot guarantee that β will ever hold.

[2] Since our focus at this point is just to identify what kind of expressiveness we need, we avoid a commitment to any specific logic, or going into formal details of the semantics.

Using these constructs, we can write the following:

$$\exists\alpha{:}A\exists\beta{:}B\left(I\Rightarrow\left(\begin{array}{l}\Box((A\ \texttt{plays}\ \alpha\wedge B\ \texttt{plays}\ \beta)\Rightarrow\texttt{Eventually}\ F)\wedge\\\Box(A\ \texttt{plays}\ \alpha\Rightarrow(Recoverable_A\ \texttt{Awaiting}\ F))\ \wedge\\\Box(B\ \texttt{plays}\ \beta\Rightarrow(Recoverable_B\ \texttt{Awaiting}\ F))\end{array}\right)\right)$$

Intuitively, here α and β are Alice and Bob's strategies when they cooperate. The first of the three inner formulas therefore says that the final condition F is guaranteed to eventually hold if they both cooperate. The following two formulas say that, each is assured, based just on the fact that they are playing their cooperative strategy, and independent of what the other agent (or any third party) does, that until such time as the final condition F holds, they have retained the ability to recover one or the other asset by executing some strategy.

This significantly improves our specification of the atomic swap. However, there remains a subtle issue. Recall that the formula $Recoverable_A$ says that there is a strategy by which A can eventually recover an asset. It does *not* say that A knows what this strategy is. This strategy may also vary from moment to moment, while A is waiting to determine whether Bob is cooperating, and has not yet decided to attempt to recover an asset. There is a difference between knowing that there exists something satisfying some property, and there existing something that is known to satisfy some property! This distinction is well known in the philosophy literature as the *de dicto* - *de re* distinction.

Particularly once we introduce cryptographic constructs, this distinction matters. Suppose we modify the smart contract of Figure 1 by adding a 256 bit integer variable `hash`, enable Bob (but not Alice) to set this variable at most once using a new function `setHashB`, and modify Alice's operation `cancel` so that it has an effect only if Alice provides a string that hashes under the SHA-256 hash function to the value of `hash`. The modifications are shown in Figure 3.

```
hash : Int256

initialise { depositedA := False ;  depositedB := False ;  hash := 0}

setHashB(h:Int256)  { if sender = Bob and hash = 0 then { hash:= h }  }

cancelA(x:String) { if depositedA  and sha256(x) = hash
                       then { depositedA := False; send(a,A) }  }
```

Fig. 3. A variant of actions in the atomic exchange smart contract

It is now the case that while cooperating, and before F holds, Alice will have always have a strategy for recovery of an asset: the strategy is to call `cancelA` with a string whose hash is `hash`. In case Bob manages to update the value from 0 to a new value (which he can do at most once), Alice should repeat this with

the inverse of the new value. However, cryptographic hash functions like SHA-256 have the property of requiring large amounts of computation to compute an inverse. Therefore, this strategy may not be immediately known to Alice, and she may need to conduct an infeasible amount of computation to discover such a string! (Alternately she could call `cancelA` with all strings in succession until one work, but this again requires a very large expected effort, as well as a very high expected cost in transaction fees in a blockchain setting that charges her for each call.) It would therefore be misleading to say that Alice is protected against Bob's malice because she has a strategy for recovery of an asset.

There exist proposals for strategy logics that add operators that talk about what agents know as well as what they are able to do ([HvdM18] is one recent work, with references to others), but it is not clear that they resolve this particular problem.[3] Instead, we resolve it by making the implicitly quantified strategy in $Recoverable_i$ explicit, and moving it to the the front of the formula. For $i = A, B$, write $Recoverable_i(\alpha)$ for the formula

$$\Box(i \text{ plays } \alpha \Rightarrow \texttt{Eventually } (\texttt{holds}(i, a) \lor \texttt{holds}(i, b))) \ .$$

This says that strategy α can be played by agent i to recover one of the two assets. Then we can overcome the issue of agent's not knowing the correct current value of a strategy that changes from state to state by means of the alternate specification

$$\exists \alpha, \alpha_R : A \exists \beta, \beta_R : B(Swap(\alpha, \alpha_R, \beta, \beta_R))$$

where $Swap(\alpha, \alpha_R, \beta, \beta_R)$ is the formula

$$I \Rightarrow \begin{pmatrix} \Box((A \text{ plays } \alpha \land B \text{ plays } \beta) \Rightarrow \texttt{Eventually } F) \land \\ \Box(A \text{ plays } \alpha \Rightarrow (Recoverable_A(\alpha_R) \texttt{ Awaiting } F)) \ \land \\ \Box(B \text{ plays } \beta \Rightarrow (Recoverable_B(\beta_R) \texttt{ Awaiting } F)) \end{pmatrix}$$

Here α and β are Alice and Bob's strategies when they cooperate, and α_R and β_R are the strategies that Alice and Bob can play to recover an asset should they decide the other is not cooperating. We have finessed the de dicto - de re issue by identifying specific strategy pairs (α, α_R) and (β, β_R). If the specification is valid, witnessed by these strategies, then Alice and Bob, provided with these strategy pairs, will know what to do.

5 Model checking an atomic swap

The specification we have developed above uses quantification over strategies. This is appropriate for an abstract specification of a swap contract: it gives maximum flexibility to the implementer to decide what concrete operations the

---

[3] For one thing, according to the usual semantics of knowledge, agents have unbounded computational powers, so these logics would in fact (contrary to intuition) state that Alice knows a strategy for recovering an asset when this requires solving a hash puzzle.

smart contract should provide, and how the agents should use those operations in order to achieve the desired effect.

On the other hand, the complexity of automated verification is, in general, higher for logics containing quantification over strategies than it is for the less expressive temporal logics on which they build. Quantification over strategies involves a version of the *synthesis* problem, that of finding a strategy satisfying a given property, from a potentially infinite set of possibilities. In multi-agent settings, this problem is computationally undecidable in its most general forms [PR90], and decidable cases tend to be highly restricted, but still computationally infeasible.

However, for our purposes, this complexity can be avoided. While the specification asks for the existence of a set of strategies, an implementation can be asked to deliver not only a smart contract satisfying the specification, but also a set of concrete strategies $\alpha, \alpha_R, \beta, \beta_R$ that witness the claim of existence. Verification of the claim amounts to checking that the formula $Swap(\alpha, \alpha_R, \beta, \beta_R)$ holds for these strategies. This formula has no remaining strategic operators, but only temporal operators, making it amenable to automated verification using model checking technology.

We demonstrate this by verifying the smart contract of Figure 1 with respect to a set of strategies for Alice and Bob. The strategies are that each either cooperates, transferring their asset to the smart contract and calling `finalise` when it holds both, or recovers by calling their `cancel` action. We have used the model checker MCK [GvdM04], but expect that it could be carried out without difficulty in other model checking systems.[4] The complete text of our MCK model is given in Appendix A.[5] MCK verifies the formulas in under a second.

A few issues arise in this exercise. One is how to represent the smart contract and the agent strategies. MCK provides a modelling language that separates systems into an environment (similar to the concurrent game structures used in the semantics of alternating temporal and strategy logics) and protocols executed by each of a number of agents. We use the environment to model the smart contract, and encode the possible strategies in the agent protocols. The environment's encoding of the smart contract is a straightforward transcription of the code of Figure 1.

Since the specification involves agents potentially switching from one strategy to another during the course of a run, we do not model each strategy as an individual protocol. Instead, we construct a single protocol for Alice that encodes the rules for the cooperating strategy α, the recovery strategy α_R, as well as a strategy in which Alice chooses her actions nondeterministically. This nondeterministic protocol covers all possible behaviours of Alice, and is included in order to enable reasoning about the situation where Alice acts maliciously. A variable `strategyA` taking a value in the set of constants `{Cooperate,Recover,Random}`

---

[4] The principal novelty of MCK is that it supports specification that concern the knowledge of agents, but we do not use this facility in the present paper.

[5] The MCK scripts discussed in this paper are also available as examples in the MCK web app at `http://cgi.cse.unsw.edu.au/~mck/mckform/`.

is used to select Alice's choice of strategy at each moment of time. This variable is assigned a value non-deterministically, and Alice's protocol then selects Alice's action in accordance with the corresponding strategy for Alice. We can then express "Alice plays the cooperation strategy α at all times in the future" by a formula `Always(strategyA = Cooperate)`. Some minor modelling details that motivate a small change to the formulas are discussed in the appendix.

6 Cross Chain Swaps

We considered above a swap performed in the context of a single blockchain. The blockchain community has also developed an approach to *cross-chain* swaps, in which the assets to be swapped exist on different blockchain systems. The specification approach discussed above continues to apply, but implementing it in such a setting is a harder problem. If Alice and Bob's assets reside on different blockchains, then we can no longer have them deposit both assets into the control of a single smart contract - we need to use a number of interoperating smart contracts residing on different chains.

A solution that has been proposed [Wikb,Wika] uses a combination of hash and time locks. A time lock is a constraint stating that an action (e.g., pay an asset to some party) cannot be performed until a specified time. A hash lock is an output value y of a cryptographic hash function h, and requires that the caller of an action supply a value x such that $h(x) = y$ in order to execute the action.

Suppose now that Alice's asset a resides on a blockchain BC_A, and Bob's asset b resides on another BC_B. A protocol for executing the swap then proceeds as follows:

1. Alice generates a random value s from a space large enough that Bob will not be able to guess it in a feasible amount of time. She keeps this secret until a later step of the protocol.
2. Alice creates a smart contract on BC_A with hash lock $h(s)$ and time lock t_A such that if Bob provides s before t_A then the asset a will be transferred to Bob, else a will revert to Alice after t_A.
3. Since Bob, can now see the value of $h(s)$, he is able to create a smart contract on BC_B with hash lock $h(s)$ and time lock t_B such that if Alice provides s before t_A then the asset b will be transferred to Alice, else will revert to Bob after t_B.
4. Before t_B, Alice uses s to collect b from Bob's smart contract. This reveals s onto Bob's blockchain.
5. Before t_A, Bob uses s to collect a from Alice's smart contract.

In order to be correct, the times t_A and t_B need to be selected so that the network delays in having smart contracts logged and actions executed on smart contracts is taken into account, and the agents get adequate time to collect the assets before the timeouts. In particular, we need that t_A is an adequate amount

greater than the latest time t_B that Alice might collect asset b and reveal the secret, to allow Bob to also collect asset a.

We have also modelled this protocol in MCK, and verified that it satisfies our specification. (Verification runs in about 17 seconds in this case.) The details are given in Appendix B. In this case, the main issue arising in the exercise is that if we are to stay in the domain of temporal logic model checking, it is necessary to abstract the hash lock, and the way that inhibits the performance of an action until the agent knows the secret. We have done so by means of a boolean variable that records whether Alice has revealed the secret in collecting b. This is a simple instance of the "perfect cryptography" assumptions that are commonly made in model checking cryptographic protocols.

A more realistic modelling would encode the hash function more explicitly (e.g., as a random oracle, or more ambitiously, as actual code) and state the specification probabilistically. A difficulty with such a more accurate model is that the computational cost of automated verification of probabilistic specifications is inherently higher than that of discrete specifications (if not undecidable), and such an approach will tend to require semi-automated theorem proving with larger amounts of human input to the verification process. We note, however, that there are correctness results for the abstract "perfect cryptography" models of the kind we have used, and we suggest that this is the preferred route for automated verification of protocols involving hash-locks. We leave a more detailed exploration of this issue for future work.

7 Discussion

Our examination of the question of how to specify an atomic swap smart contracts suggests that while pre-condition/post-condition specifications do not suffice, ideas from logics with quantification over strategies are conceptually useful for stating requirements, but in practice temporal logic verification methods may suffice for verifying concrete implementations of smart contracts, and the strategies that are used to operate them. We conclude by discussing some issues we have left unaddressed, which provide questions for future work.

One issue that our specifications have not covered is the question of what is permitted to vary besides control of the assets a and b. It can be expected that some other things *will* vary in any implementation. For example, in order to execute the smart contracts involved, Alice and/or Bob are likely to have to pay transaction fees, so their holding of crypto-currency is also likely to have to vary. It may be desirable to place some bound on their total costs. On the other hand, beyond this, we would not want an implementation that requires Alice and Bob to incur losses in their holdings of other assets. It would therefore be desirable to add a statement to the specification indicating exactly what is allowed to vary in a solution. In the refinement calculus, this is often done by means of a *frame specification*. For example, a pre-condition/post-condition specification $[\phi, \psi]_X$ asks for a program satisfying $[\phi, \psi]$, while modifying *no more than* the variables in the set X. It would be desirable to add such a constraint to our specifications,

but temporal logic and its extensions usually lack the expressive power for such constraints. Some steps in this direction have been taken in the exploration of connection between temporal logics and separation logic [Bro13], and it would be interesting to explore this direction for our context.

In our model checking exercise, we have verified a concrete implementation that talks only about Alice and Bob's holdings of a and b, so it seems reasonable that it follows that a frame condition stating that nothing else varies will be true of this implementation. However, it would be desirable to have a precise statement of this claim. For one thing, an implementation containing actions whose execution requires all the available time and/or space in a block will prevent other actions being performed, so it is not true that a smart contract runs entirely free from interference with other contracts on a disjoint set of assets, particularly when these have deadlines of their own.

Various improvements are possible to our modelling, that we have not pursued since they are orthogonal to our main objective of eliciting the type of specification and verification required for the smart contracts we consider. We have not attempted to build an accurate model of the way that agents submit their transaction to the network, or the way that miners select those transactions for execution. Our models assume that agents offer an action for execution at each step, and can retract those actions instantaneously. In a real network there would be delays, uncertainties and transaction costs in doing so. Duplicate or conflicting transactions offering different transaction fees is something that may need to be incorporated in the miner model. In this event, it may then become relevant to consider game theoretic analysis: we expect that miners will opt to take a higher transaction fee, although a miner who has not yet received a message offering a higher gas price may still use a lower price transaction.

We have considered only bilateral swaps in this paper. Herlihy [Her18] generalises from two-party swap protocols to an arbitrary number of parties with a set of transfers represented using a directed acyclic graph. His protocol uses multi-signature hash-locks, where a chain of signatures on the secret is required to open a hash-lock. To verify this protocol using temporal logic techniques would require an additional abstraction to be made to represent which agents have signed a particular message. While we do not think that this would be difficult to model, it would be an interesting direction to explore.

Finally, what we have verified in our work is an abstract model of smart contract code rather than code in a native blockchain smart contract language. It would be beneficial to bridge this gap. Two approaches are possible. One is to verify an abstract modelling notation that comes equipped with a correctness preserving translation to native blockchain code. The other is to have a tool that verifies temporal specifications directly on blockchain code. We note however, that what needs to be verified is not just the smart contract, but also the strategies for operating it, and the latter are off-chain, and generally represented in some other language. A further complication is that in the multi-chain setting, we need to deal with multiple, possibly quite distinct smart contract languages for each chain. This suggests that the former route, of conducting

verification using notations that abstract from chain-specific details, may be the cleaner route to pursue. A further benefit of this would be that it could provide automated support for correct implementation of details that are abstracted by the framework for verification purposes, such as the hash locks and signatures.

References

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-Time Temporal Logic. *Journal of the ACM*, 49(5):672–713, 2002.

[Bro13] Rémi Brochenin. *Separation Logic: Expressiveness, Complexity, Temporal Extension*. Phd thesis, École Normale Supérieure de Cachan, 2013.

[BvW98] R. Back and J. von Wright. *Refinement Calculus, A systematic introduction*. Springer, 1998.

[CHP10] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Strategy logic. *Information and Computation*, 208(6):677–693, 2010.

[Eme90] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. MIT Press, 1990.

[GvdM04] Peter Gammie and Ron van der Meyden. MCK: model checking the logic of knowledge. In *Proc. Computer Aided Verification, 16th International Conference, CAV*, pages 479–483, 2004.

[Her18] M. Herlihy. Atomic cross-chain swaps. In *Proc. ACM Symp. on Distributed Computing*, 2018. Version at `arXiv:1801.09515`.

[HvdM18] Xiaowei Huang and Ron van der Meyden. An epistemic strategy logic. *ACM Transactions on Computational Logic*, 2018. To appear, a preliminary version is at `arXiv:1409.2193`.

[Ltd17] Parity Technologies Ltd. A postmortem on the parity multi-sig library self-destruct. `https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`, 15 November 2017. Accessed Sep 2018.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 2nd ed. edition, 1994.

[Pop16] Nathaniel Popper. Hacker may have taken $50 million from cybercurrency project. The New York Times, 17 June 2016. See also `http://en.wikipedia.org/wiki/The_DAO_(organization)`.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. of IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 746–757. IEEE, 1990.

[Wika] Bitcoin Wiki. Atomic cross-chain trading. `https://en.bitcoin.it/wiki/Atomic_cross-chain_trading`, Accessed Sep 2018.

[Wikb] BitCoin Wiki. Hashed timelock contracts. `https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts`, Accessed Sep 2018.

A   MCK model of an on-chain swap contract

The following gives MCK code for the on-chain swap contract and the strategies that operate it. The codes starts by declaring a number of types and defining the environment, by listing a number of variables that make up the environment

state. The MCK statement `init_cond` is used to constrain initial states of the environment: all states that satisfy the condition in this statement are initial. Note that since the variables `strategyA`, `strategyB` are not mentioned, they may take any value in an initial state. The `define` statements can be understood as macro definitions. An `agent` statement introduces an agent and identifies the protocol that the agent uses to choose its actions. The protocol arguments in this statement are used to alias the protocol's parameter variables to variables in the environment.

The `transitions` statement gives (nondeterministic) code that is used to make a single transition of the environment state, depending on actions that have been selected by the agents as inputs to the transition. In protocols, an action `a` is denoted by `<<a>>`. In the environment, there is a corresponding atomic proposition (i.e., boolean variable) $i.a$ for each agent i, representing the fact that the agent is performing action a in the current state transition. For example, in our model, Alice has an action `Deposit`, and there is a corresponding atomic proposition `Alice.Deposit`.

A statement `if ... fi` is a Dijkstra style nondeterministic guarded choice statement. A statement $[[vars|\phi]]$ also nondeterministic. The list of variables $vars$ in its left hand side are all the variables whose value may change when running this statement. The formula ϕ in the right hand side gives a condition on a pair (s, s'), where s, s' are states of the environment, through use of state variables to refer to component s and "primed" version of state variables to refer to s'. Started in a state s, this statement can make a transition to any state s' (in which only the variables in `vars` may have been changed) such that the pair (s, s') satisfies the formula ϕ. For the particular application of this statement, `[[ strategyA,strategyB, turn | True ]]` has the effect of non-deterministically selecting values for each of the listed variables.

The model schedules one of the agents to have its choice of action executed in each transition. The variable `turn` indicates the scheduled agent. Each statement `fairness` gives a condition that must be true infinitely often in each run. Thus `fairness = turn == AliceP` says that Alice is treated fairly in the sense that she gets infinitely many turns in each run (but she may have to wait an arbitrary amount of time between turns).

This is followed by statements `spec_obs`, which give the formulas to be model checked. (The component `obs` in this keyword indicates the semantics to be used for knowledge modalities - since there are none in our specification, this is irrelevant here.) The formulas are those from the body of the paper, using MCK's notation (consistent with the literature) for the temporal operators: A says "on all runs from the present state", G says "at all future times", F says "at some future time", and $\alpha U \beta$ says α holds up until some future time where β holds.

The specifications contain one minor deviation from the formulas given above: we need a reformulation of the formula

$$A \text{ plays } \alpha \Rightarrow (Recoverable_A(\alpha_R) \text{ Awaiting } F) \,.$$

(A revised version of the MCK scripting under development will help to avoid this deviation.) The reason for this is that we encode agents' choice of strategy as a state variable, and the specifics of MCK's execution model, in which, at each transition, the state is used to select agent actions which are then used as inputs to the environment transition code. On a run where A plays α, at a point where A switches to playing α_R, she is already committed to playing α for the next step. Thus, rather than saying that A always plays α, we talk about what happens when A switches to playing α_R in a situation where she has always played α in the past. For this, it is convenient to also build in a variable `playedCoopA` that tracks whether Alice cooperated at all times in the past.

```
type Strategy = {Cooperate,Recover,Random}
type Holder = {AliceH,BobH,Contract,Other}
type Player = {AliceP,BobP}

done : Bool

depositedA : Bool
holdera : Holder

depositedB : Bool
holderb: Holder

strategyA : Strategy
strategyB : Strategy

turn : Player

{- the following track whether a player has always
   cooperated up to the present time -}
playedCoopA : Bool
playedCoopB : Bool


define swapped = holdera == BobH /\ holderb == AliceH

define alice_safe = holdera == AliceH \/ holderb == AliceH

define bob_safe = holdera == BobH \/ holderb == BobH

init_cond = holdera == AliceH /\ holderb == BobH /\
            playedCoopA /\ playedCoopB /\
            neg (done \/ depositedA \/ depositedB )

agent Alice "player" (strategyA,depositedA,depositedA,depositedB)

agent Bob "player" (strategyB,depositedB,depositedA,depositedB)
```

```
transitions
begin
 -- process Alice or Bob's action, if it is their turn
 if
  turn == AliceP ->
    if
    Alice.Deposit /\ holdera == AliceH ->
  begin depositedA := True ; holdera := Contract end

[] Alice.Cancel /\ holdera == Contract ->
        begin depositedA := False; holdera := AliceH end

[] Alice.Finalize /\ depositedA /\ depositedB ->
  begin depositedA := False ; depositedB := False ;
          holdera := BobH ; holderb := AliceH end

        -- GiveToOther means give away one of your assets
        [] Alice.GiveToOther /\ holdera == AliceH -> holdera := Other
        [] Alice.GiveToOther /\ holderb == AliceH -> holderb := Other

[] otherwise -> skip
    fi
[] turn == BobP ->
   if
       Bob.Deposit /\ holderb == BobH ->
          begin depositedB := True ; holderb:= Contract end

    [] Bob.Cancel /\ holderb == Contract /\ depositedB ->
          begin depositedB := False; holderb := BobH end

    [] Bob.Finalize /\ depositedA /\ depositedB ->
          begin depositedA := False ; depositedB := False ;
                  holdera := BobH ; holderb := AliceH end

    -- GiveToOther means give away one of your assets
    [] Bob.GiveToOther /\ holdera == BobH -> holdera := Other
    [] Bob.GiveToOther /\ holderb == BobH -> holderb := Other


    [] otherwise -> skip
   fi
 fi ;

-- update the record of whether a player has always cooperated
if strategyA /= Cooperate -> playedCoopA := False fi ;
if strategyB /= Cooperate -> playedCoopB := False fi ;

 -- nondeterministically select each player's strategy, and whose turn it is
 [[ strategyA,strategyB, turn | True ]]
```

```
        end


-- Both Alice and Bob get infinitely many turns in each run
fairness = turn == AliceP
fairness = turn == BobP



spec_obs = "If Alice and Bob always play Cooperate,
                      then eventually the swap is successful"
   A( (G(strategyA == Cooperate /\ strategyB == Cooperate)) => F swapped)



{- In the following, note "A awaiting B" is "(A until B) or (always A and not B)" -}

spec_obs = "If Alice has always cooperated to now, then she can get an asset back
                      by playing recover, awaiting such time as the swap has occurred"
  A( ( (playedCoopA => ((G strategyA == Recover) => F alice_safe)) U swapped )
      \/
      G ( (playedCoopA => ((G strategyA == Recover) => F alice_safe))
          /\
          neg swapped ))


spec_obs = "If Bob has always cooperated to now, then he can get an asset back
                      by playing recover, awaiting such time as the swap has occurred"
  A( ( (playedCoopB => ((G strategyB == Recover) => F bob_safe)) U swapped )
      \/
      G ( (playedCoopB => ((G strategyB == Recover) => F bob_safe))
          /\
          neg swapped ))



{- Note that because a finalise operation might race with a cancel operation, recovery
   of your own asset is not guaranteed! -}

spec_obs =  "(FALSE) Alice is always able to ensure that she will eventually hold asset a,
                      by playing strategy Recover"
  A( ( (playedCoopA => ((G strategyA == Recover) => F holdera == AliceH )) U swapped )
      \/
      G ( (playedCoopA => ((G strategyA == Recover) => F holdera == AliceH))
          /\
          neg swapped ))



spec_obs =  "(FALSE) Bob is always able to ensure that he will eventually hold asset b,
```

```
                          by playing strategy Recover"
  A( ( (playedCoopB => ((G strategyB == Recover) => F holderb == BobH )) U swapped )
      \/
      G ( (playedCoopB => ((G strategyB == Recover) => F holderb == BobH))
          /\
          neg swapped ))
```

```
protocol "player" (strategy : Strategy, deposited : Bool,
                              depositedA : Bool ,depositedB : Bool)

begin
do
     strategy == Cooperate /\ neg deposited  -> <<Deposit>>
[]   strategy == Cooperate /\ depositedA /\ depositedB  -> <<Finalize>>

[]   strategy == Recover /\ deposited  -> <<Cancel>>

[]   strategy == Random -> <<Deposit>>
[]   strategy == Random -> <<Cancel>>
[]   strategy == Random -> <<Finalize>>
[]   strategy == Random -> <<Skip>>
[]   strategy == Random -> <<GiveToOther>>

[]   otherwise -> <<Skip>>
od
end
```

B MCK model of a cross-chain atomic swap protocol

The following is MCK code for a cross-chain atomic swap protocol using hashed
time locks (with the hash checking and knowledge of the secret abstracted using a
single boolean variable rather than a large range of possible values for the secret).
Because of the timeouts, we are able to combine the Cooperate strategy and the
Recover strategy into a single strategy for each player, in which they perform the
recovery action in case their smart contract times out without the other player
having claimed the asset. This also allows for a simpler, stronger specification
statement concerning recovery. All the specifications in the following script are
verified to hold.

```
type Strategy = {Cooperate,Recover,Random}
type Holder = {AliceH,BobH,ContractA,ContractB,Other}
type Player = {AliceP,BobP}
```

```
type Time = {0..20}

{-
   The secret x is abstracted - type Secret is used to model
   whether a player knows the secret.  If so, their Claim action is effective
-}
type Secret = {Known,Unknown}


holdera : Holder
holderb: Holder

strategyA : Strategy
strategyB : Strategy

turn : Player
time : Time

{- the following model each player's view of Alice's secret -}
viewSecretA : Secret
viewSecretB : Secret


{- Whether an asset has been paid into the contract.
   These are used to prevent paying the asset into the contract
   once again in the event it is recovered. -}
depositedA : Bool
depositedB : Bool

timeoutA : Time
timeoutB : Time


-- the following track whether a player has always cooperated up to the present time
playedCoopA : Bool
playedCoopB : Bool


define swapped = holdera == BobH /\ holderb == AliceH

define alice_safe = holdera == AliceH \/ holderb == AliceH

define bob_safe = holdera == BobH \/ holderb == BobH


init_cond = holdera == AliceH /\ holderb == BobH /\
            viewSecretA == Unknown /\ viewSecretB == Unknown /\
            playedCoopA /\ playedCoopB /\ time == 0 /\
   -- (constant) timeouts for Alice and Bob's contracts
```

```
              timeoutA == 8 /\ timeoutB == 6  /\ turn == AliceP /\
     neg (depositedA \/ depositedB)




agent Alice "playerA" (strategyA,holdera,holderb,viewSecretA,
                                  time,timeoutA,depositedA)

agent Bob "playerB" (strategyB,holderb,holdera,viewSecretB,
                                  time,timeoutB,depositedB)


transitions
begin
 -- process Alice or Bob's action, if it is their turn
 if
    turn == AliceP ->
    if
       Alice.Generate -> viewSecretA := Known
    [] Alice.Deposit /\ holdera == AliceH ->
               begin holdera := ContractA ; depositedA := True end

    -- when Alice claims from Bob's contract, she reveals the secret to Bob
    [] Alice.Claim /\ viewSecretA == Known  ->
          begin
              if holderb == ContractB -> holderb := AliceH fi ;
              viewSecretB := Known
          end

    [] Alice.Recover /\ holdera == ContractA /\ time >= timeoutA ->
          holdera := AliceH

         -- GiveToOther means give away one of your assets
    [] Alice.GiveToOther /\ holdera == AliceH -> holdera := Other
    [] Alice.GiveToOther /\ holderb == AliceH -> holderb := Other

    [] otherwise -> skip
    fi
[] turn == BobP ->
   if
      Bob.Deposit /\ holderb == BobH ->
         begin holderb:= ContractB ; depositedB := True end

    [] Bob.Claim /\ viewSecretB == Known /\ holdera == ContractA ->
           holdera := BobH

    [] Bob.Recover /\ holderb == ContractB /\ time >= timeoutB ->
           holderb := BobH
```

```
          -- GiveToOther means give away one of your assets
       [] Bob.GiveToOther /\ holdera == BobH -> holdera := Other
       [] Bob.GiveToOther /\ holderb == BobH -> holderb := Other

       [] otherwise -> skip
     fi
 fi ;

-- update the record of whether a player has always cooperated
if strategyA /= Cooperate -> playedCoopA := False fi ;
if strategyB /= Cooperate -> playedCoopB := False fi ;

-- Assume the two chains are synchronised, modelled by round robin scheduling
if     turn == AliceP -> turn := BobP
   [] turn == BobP -> turn := AliceP
fi ;

-- randomly select the strategy followed by each player at each step.
[[ strategyA,strategyB | True ]]  ;

-- tick the clock
time := time + 1
end


spec_obs = "If Alice and Bob always play Cooperate,
                     then eventually the swap is successful"
   A( (G(strategyA == Cooperate /\ strategyB == Cooperate)) => F swapped)

spec_obs = "If Alice always cooperates, she is always eventually safe"
  A( (G strategyA == Cooperate) => G F alice_safe )

spec_obs = "If Bob always cooperates, he is always eventually safe"
  A( (G strategyB == Cooperate) => G F bob_safe )



protocol "playerA" (strategy : Strategy,
    holder_mine : Holder,
    holder_other : Holder,
    viewSecret : Secret,
    time : Time,
    timeout: Time,
    deposited: Bool)

begin
do

    strategy == Cooperate ->
```

```
        if  -- generate a secret (which becomes Known)
            viewSecret == Unknown -> <<Generate>>
        -- unless you have done so before, deposit your asset
        []  viewSecret == Known /\ holder_mine == AliceH /\
                neg deposited -> <<Deposit>>
        -- make a claim as soon as you know the secret and Bob has deposited
        []  viewSecret == Known /\ holder_other == ContractB -> <<Claim>>
        -- try to recover your own asset at the earliest time after the timeout
        []  holder_mine == ContractA /\ time >= timeout -> <<Recover>>
        fi

[]  strategy == Random -> <<Generate>>
[]  strategy == Random -> <<Deposit>>
[]  strategy == Random -> <<Claim>>
[]  strategy == Random -> <<Recover>>
[]  strategy == Random -> <<Skip>>
[]  strategy == Random -> <<GiveToOther>>

[]  otherwise -> <<Skip>>
od
end




protocol "playerB" (strategy : Strategy,
    holder_mine : Holder,
    holder_other : Holder,
    viewSecret : Secret,
    time : Time,
    timeout: Time,
                    deposited : Bool )

begin
do

    strategy == Cooperate  ->
      if  -- once you see Alice has made her deposit, make yours
          holder_mine == BobH /\ holder_other == ContractA /\
          neg deposited -> <<Deposit>>

            -- make a claim as soon as you know the secret
        []  viewSecret == Known /\ holder_other == ContractA -> <<Claim>>

          -- try to recover your own at the earliest possible time
        []  holder_mine == ContractB /\ time >= timeout -> <<Recover>>
        fi

[]  strategy == Random -> <<Deposit>>
[]  strategy == Random -> <<Claim>>
```

```
[]   strategy == Random -> <<Recover>>
[]   strategy == Random -> <<Skip>>
[]   strategy == Random -> <<GiveToOther>>

[]   otherwise -> <<Skip>>
od
end
```