A TLA+ formal proof of a cross-chain swap

Zeinab Nehai, Francois Bobot, Sara Tucci, Carole Delporte-Gallet, Hugues Fauconnier

A TLA+ Formal Proof of a Cross-Chain Swap

Zeinab Nehaï
Université Paris-Saclay, CEA, List
IRIF, Université de Paris
France

Sara Tucci-Piergiovanni
François Bobot
Université Paris-Saclay, CEA, List
F-91120, Palaiseau, France

Carole Delporte-Gallet
Hugues Fauconnier
IRIF, Université de Paris
Paris, France

ABSTRACT

Blockchains are a specific type of distributed ledgers structured by a sequence of blocks of transactional data linked to each other. The use of blockchains has increased over time, and several new blockchains are emerging. It is therefore essential to enhance the interoperability between blockchain implementations to allow decentralised trading. One way to achieve this is with *Cross-Chain Swap* protocols. These protocols are critical systems as they handle assets. Therefore, it must be sure that the system does not contain errors. In this paper, we describe the *Cross-Chain Swap* problem in a formal way. We define safety and weak-liveness properties that guarantee no correct participant will be worse-off in an asynchronous system. Moreover, we provide a formally proved Byzantine fault-tolerant protocol that satisfies the swap specification. The protocol abstracts the blockchain enough to suit various distributed ledger frameworks aiming to perform a cross-chain swap. In addition, we illustrate how the described abstract protocol can be instantiated in a blockchain system.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**.

KEYWORDS

cross-chain, blockchain, byzantine fault-tolerance, formal methods, tla+

1 INTRODUCTION

Context and Motivation. A *blockchain* system is a distributed ledger that maintains a continuously growing history of unalterable ordered information organised in a chain of blocks. The most well-known and used blockchains are Bitcoin [15] and Ethereum [2]. A feature that has given rise to a strong interest in blockchains is writing *smart contracts* [22]. These are sequential and executable programs that run in blockchains. They permit trusted transactions and agreements to be carried out among parties without the need for a central authority while keeping transactions traceable, transparent, and irreversible. Blockchain technology has become popular both in industry and academia. Thus, its use has seen a considerable increase in recent years. As a result, it is necessary to develop infrastructures that allow transactions across multiple blockchains. One such solution is what we call "*Cross-Chain Swap*". This solution was first mentioned in a discussion on a bitcoin forum [18], and since then, many people are interested in it. The motivation of this concept is to enable exchanging assets or tokens between different blockchains.

At a high level, the idea is to have a set of participants settling transactions. For example, Alice transfers a red coin to Bob, which transfers a green coin to Charlie, which transfers to Alice a digitised asset. A distributed protocol is responsible for realising the swap among participants. However, in the current literature, swap specifications do not agree on what a swap protocol should guarantee in terms of safety and liveness properties [7]. In addition, underlying timing and failure assumptions vary from a protocol to another [5][8][25][19]. In synchronous solutions [5][8], based on timed actions, a swap can result in a correct but slow participant being worse-off at the end of the swap. Zakhary *et al.* [25] are the first to propose a protocol in which correct "asynchronous" participants are never worse-off at the end of the swap. By getting close to the well-known *two-phase* commit algorithm [1], participants in [25]'s lock their asset at the beginning of the protocol. Afterwards, a coordinator either authorises or aborts all the transfers.

Furthermore, to date, very little work has focused on the formal verification of such protocols hindering their safe application[16], more so with Byzantine. The difficulty of proving a distributed protocol in the presence of Byzantine failures is well-known due to its ability to deviate arbitrarily from the protocol, which poses difficulty in representing its behaviour in formal tools [10].

Our Approach. This paper introduces a formal specification for *Cross-Chain Swap* in the presence of Byzantine participants, and an abstract swap protocol formally proved, inspired by [25]. The formal specification separates the swap problem from the protocol in a clear way. We define two properties, *safety* and *weak liveness*, which guarantee, in an *asynchronous* system, that no correct participant will end up worse-off. Moreover, we provide a semi-automatic proof of the protocol using TLA+ [12], a language for formal specification of distributed systems. It has a model-checker, TLC [24] and a proof system, TLAPS[14].

The protocol is blockchain agnostic and relies on an abstraction that we call "*proof-of-action*" to cope with Byzantine participants. The idea is that a participant sends proof to the coordinator of having prepared the transaction correctly. The coordinator, assumed correct, will verify the sending proof and authorise the swap only if all the participants sent valid proofs.

Notably, both the *proof-of-action* and the assumption of the coordinator's correctness can be achieved in a blockchain context. The *proof-of-action* can be implemented by extracting correct information from participants' blockchain [25][6]. In [25], the *proof-of-action* represents a part of the blockchain's history that must contain the needed information to authorise the swap while in [6] the *proof-of-action* is a sequence of blocks and their certificates.

The assumption of the coordinator's correctness relies on the coordinator's logic implemented as a smart contract. This implementation allows considering the coordinator as correct and public. A smart contract is a passive entity and, its execution is triggered only by participants' calls; thus, it will always be responsive.

Moreover, the persistence of the coordinator execution trace allows a slow process to get the coordinator decision asynchronously and then either retrieve the asset (if the swap is authorised) or recovers the asset initially possessed (if the swap is aborted).

Crucially, the *proof-of-action* and the coordinator correctness restrict the Byzantine behaviour in a predetermined and detectable way. Consequently, it helps us carry out the formal proof of the protocol so that the presence of Byzantine failures does not add significant complexity. In our setting, a Byzantine participant could try to double-spend an asset already locked in its blockchain. However, in the case where the asset is no longer in the hands of the Byzantine, the *proof-of-action* will not be valid, leading to the abortion of the transfer.

Let us finally note that, in our specification, a non-correct process might not retrieve (or recover) the asset, contrary to [25]. The atomicity property in [25] states that all or none of the asset transfers occur. We think that the specification introduced in our paper, which weakens the [25]'s atomicity, is more adapted to the blockchain context. Namely, each participant is responsible for retrieving/recovering its asset or not at the end of the swap.

Contributions. In summary, our contributions are the following:

- We propose a specification for cross-chain swaps resilient to Byzantine failures.
- We describe a formal protocol that satisfies the specification, relying on abstracted blockchain properties.
- A semi-automatic proof of the protocol is provided using the TLA+ tool.

The paper is organised as follows: Section II describes the problem specification and the swap definition. Section III introduces the protocol specification that includes the protocol of each participant and the description of the swap. Section IV provides the formal proof, using TLA+, of the protocol. Section V illustrates how the abstract protocol can be instantiated in blockchain systems. A state-of-the-art review of existing work concerning cross-chain protocols is described in section VI. Finally, section VII gives some conclusions.

2 PROBLEM SPECIFICATION

In this section, we define the system model of the swap problem and its specification.

The system is composed of a set of participants Π and a set of assets Λ. Π can be defined as $\Pi = \Pi_s \cup \Pi_r$ with Π_s the set of participants transferring assets and Π_r the set of participants receiving transferred assets. An asset has a unique owner. A participant can transfer its asset's ownership to another participant. Throughout the study, we will use the term "asset" for reasons of clarity but, it should be remembered that it is the "asset's ownership" that is transferred and not the physical asset.

Each participant is asynchronous and has a local clock. In addition, messages are digitally signed; hence we assume that they cannot be forged.

Failure Model. A participant who never fails in the system is said to be *correct*. A *Byzantine* is a participant for whom nothing can be assumed about its behaviour. There is no bound on the number of *Byzantine* participants in the system.

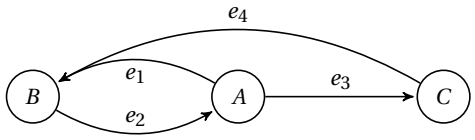


Figure 1: A swap graph S with $\Pi = \{A, B, C\}$ and $m = 4$ assets

Swap Model. A swap S is modelled as a directed graph $S = (\Pi, E)$ (see figure 1). S is composed of a set of vertices Π (the set of participants) and a set of labelled edges $E = \{e_1, e_2, ..., e_m\}$. The label of an edge is the transferred asset. Each edge of S transfers a unique asset from the set of assets involved in the swap; Λ. Consequently, $|E| = m$ represents the total number of transferred assets in S. An edge is defined as $e_i = (s, a_i, r) \in \Pi_s \times \Lambda \times \Pi_r$ with $i \in \{1, .., m\}$, $s \neq r$, and a_i the label of the edge that designates the transferred asset. Moreover, Π_s is the set of participants transferring assets, "*sources*" (vertices with outgoing edges) and Π_r the set of participants receiving transferred assets, "*recipients*" (vertices with incoming edges).

Note that a participant who is both a source and a recipient will have two different identifiers. For example, participant A in figure 1 is a source for edges $\{e_1, e_3\}$; hence it will be identified by s_A, and is a recipient, identified by r_A, for the edge $\{e_2\}$.

We assume that, before the swap, the graph is constructed by all the participants. Thereby, they agree with its configuration (the graph construction is out of the scope).

Problem Definition. A *swap* is a distributed transactions model. The objective is to transfer assets between participants across multiple distributed ledgers. An asset can be a cryptocurrency or a certificate of a physical asset's ownership. The goal is to transfer an asset's ownership in a trustless environment without an intermediary. In a swap, the number of participants and assets is finite. A participant is a user of any distributed ledger involved in the swap; it can be either a *source* or a *recipient*. The *source* transfers its asset's ownership, and the *recipient* receives a transferred asset's ownership. There is no limit to one source transferring multiple assets and one recipient receiving multiple transferred assets within the same swap. The swap problem is specified as follows [1]:

- *Consistency*. For any *correct* source s_1 of an edge $e_1 = (s_1, a_1, r_1)$ and *correct* recipient r_2 of an edge $e_2 = (s_2, a_2, r_2)$, at the end of their execution, either s_1 owns a_1 or r_2 owns a_2.
- *Ownership*. No asset owned initially by a *correct* source is ownerless forever or, no asset intended to be transferred to a *correct* recipient is ownerless forever.
- *Retrieving*. If all participants are *correct* then all recipients will *retrieve* their intended assets.

Consistency is a *safety* property, and *Ownership* is a *weak-liveness* property. Both properties are satisfied in an *asynchronous* system. *Retrieving* property is a *strong liveness* property that assumes a *synchronous* system.

[1] The specification is inspired by [6]

The *Consistency* property states that no correct participant will end up worse-off. Since the system tolerates Byzantine participants, the classical atomicity definition "*all-or-nothing*" cannot be applied, as said in [6]. It is impossible to force a participant to initiate the transfer of its asset. For this reason, safety is intended to be weaker than classical atomicity while ensuring that a correct participant will always terminate safely. Moreover, reasoning about a pair of correct source and recipient is sufficient to extrapolate the property to all pairs of correct participants, thus avoiding the limitation of checking the execution completion of all correct participants. The *Ownership* property assumes that a Byzantine participant may choose never to *retrieve* its asset(s) (if the swap is authorised) or to *recover* its asset(s) (if the swap is aborted) and to leave the asset(s) ownerless (the asset is neither owned by the source nor by the recipient). However, a slow participant that is not Byzantine will never end up worse-off. Thereby, it will always either *retrieve* or *recover* its asset(s) asynchronously. Note that "*recovers*" is only used for sources. It can be translated by the restitution of the asset to the source. "*Retrieves*" is only used for recipients meaning receiving the transferred asset by the recipient.

The *Retrieving* property state the desired outcome in the case where all participants are correct. It avoids any empty protocols.

3 PROTOCOL SPECIFICATION

This section describes the protocol specification that details the asset representation, which defines asset states and transitions and the participants' state machines. Moreover, this section details the different phases of the abstract protocol $\mathscr{P}_{swap}$.

3.1 Representation of Asset's States in a Swap

In this part, we introduce a representation of the asset's possible states in the swap. For the proof of the protocol, detailed later, we project the possible states of an asset a_i as follows (figure 2(a)): the state "OwS" characterises *Owned by its Source*; the original owner s_i. This state is reached in the initial state and when the source recovers its asset. "LOCKED" state is when s_i locks the asset and designates the new owner of the asset (the receiver of the asset; r_i). "OwR" state, *Owned by its Recipient*, is when the asset has been retrieved by r_i (the new owner). We introduce an additional state "OTHER" that characterises all other states beyond the swap. For example, if an asset is transferred to a participant who is not part of the swap or transferred without following the swap transfer's rules, the asset is set to "OTHER". We will detail this point later.

The participants have operations that, once computed, cause a change in an asset's state. The protocol interacts with assets through trigger events ϵ_i, where $i \in \mathbb{N}$. Triggers make it possible to modify assets' states.

An asset can change its state legally (following an action made by a correct participant; plain edges $\rightarrow$) or illegally (following an action made by a Byzantine participant, dashed edges $\dashrightarrow$).

Sources have two operations: (1) $LockAsset(a_i, r_i)$; locking the asset a_i and assigning r_i as the new owner of the asset (ϵ_1^a in figure 2(a)); (2) $RecoveringAsset(a_i)$; recovering the asset a_i and becoming again the owner of a_i (ϵ_2^a). Recipients have one operation: $RetrievingAsset(a_i)$; retrieving the asset a_i and becoming the new owner of a_i (ϵ_3^a). The implementation of these operations depends on the protocol, which is detailed later.

Moreover, Byzantine actions can also change the assets' state. Their actions are the following (see figure2(a)):

- ϵ_4^a: a Byzantine source transfers directly the asset to the recipient without passing through the swap.
- ϵ_5^a: a Byzantine recipient, once retrieves the asset, can send back the asset to the original owner, the source.
- $\epsilon_{\{6,7\}}^a$: a Byzantine source or recipient can transfer its asset to an unknown participant or lock it somewhere or all other action not recognised by the swap.

We can see from figure 2(a) that there is no illegal action from "LOCKED". This state reflects the locking asset respecting the swap's rules. Therefore, once an asset is legally locked, it can only be legally unlocked. In addition, we did not represent the outgoing edges from "OTHER", as this would not add any decisive information since the outgoing edges would cancel the incoming edges.

3.2 The Abstract Protocol $\mathscr{P}_{swap}$

The abstract protocol, $\mathscr{P}_{swap}$, is modelled as a set of state machines that influences the assets introduced in Section 3.1.

The following working hypotheses ensure the clarity of the protocol: (1) The protocol is tolerant to unbounded Byzantine faults. (2) A *proof-of-action* (detailed later) allows countering Byzantine behaviour. (3) A *proof-of-action*, once provided, cannot be forged, even if a Byzantine provides the proof.

3.2.1 Overview of the Protocol $\mathscr{P}_{swap}$. $\mathscr{P}_{swap}$ is inspired by the defined protocol in [25]. The idea is similar to the well-known *two-phase commit* protocol [1]. The two-phase commit ensures that a transaction either commits or aborts for all the participants. It avoids the undesirable outcome that the transaction commits for one participant and aborts for another. For a two-phase commit to taking place, a special entity, known as a *coordinator*, is required. The coordinator decides whether to commit or abort the transaction and communicates the result to all the participants.

In $\mathscr{P}_{swap}$, the coordinator is defined as a public entity. We assume a communication channel between the coordinator and each participant, but no direct communication among the participants during the swap. The behaviour of each participant is independent of the others. On the other hand, the coordinator behaviour influences the participants and vice versa. We make no assumptions about the behaviour of participants.

We assume that correct participants can evaluate the correctness of the coordinator. Hence, if the coordinator is Byzantine, the swap could not start in the first place. From this premise, if the swap starts we assume the coordinator correct to simplify the description of the protocol. To prove their wish to commit the swap, all sources must lock their asset(s). The coordinator has the role of authorising the swap, or not, by giving a decision to the swap participants. Only the coordinator decision can unlock the assets. The possible decisions are the *redeem* decision to authorise the swap or the *refund* decision to prohibit the swap.

3.2.2 Proof-of-actions. The coordinator and participants can verify executed actions in the swap. To do this, we use a *proof-of-action*: the coordinator and participants can produce proof that a

given action or state change was correctly done. This proof cannot be falsified. If any proof is false, then it will be automatically detected. If a given action was correctly done, the proof is valid.

3.2.3 Participants State Machines. $\mathscr{P}_{swap}$ interacts with participants of the swap where their behaviour is represented by a state machine structured with the following elements (Γ, Q, Σ, δ, q_0, F).

- Operations Γ: a finite set of operations.
- Vertices Q: a finite set of states, represented by circles and labelled with unique designator symbols.
- Label symbols Σ: a finite collection of label symbols.
- Edges δ: represents transitions from one state to another as caused by the label. A transition is written as $\delta : Q \times \Sigma \rightarrow Q$.
- Start state q_0: represents the initial state, where $q_0 \in Q$ drawn by a "start" incoming arrow.
- Accepting state F: represents the final state drawn by a double circle, where $F \in Q$. When this state is reached, the state machine can no longer evolve into another state.

Moreover, a label in Σ contains three parts (each one is optional), written $q \xrightarrow{\epsilon;\sigma;\omega} q'$: an action name ϵ, a guard σ expressing a condition and an operation name ω. A guard is a condition to satisfy the transition and, an action is an event that allows taking the transition. An action can be a sending message action, denoted by the discrete action $\epsilon!$, or a receiving message, denoted by the discrete action $\epsilon?$. An operation ω is the computation of an operation in Γ. The symbol $\oslash$ is used where the label does not contain one of the three parts. We now introduce the protocol of each participant (see figure 2):

The Publisher. The publisher is a participant in Π. Its role is to publish the swap graph to the coordinator with the action ϵ_1^p : $publish!$ from figure 2(e). A publisher can also be a source and/or a recipient.

The Coordinator. The role of the coordinator is to coordinate the evolution of the swap. The coordinator gives the authorisation to carry out the swap or not by changing states. Its state machine is public; therefore, any state updates are known to all. As explained previously, the coordinator evolves according to the participants' behaviour (see figure 2(b)): in ϵ_1^c, the coordinator waits for the publisher to execute the $publish!$ action. Then, in ϵ_2^c, the coordinator waits for the participants to ask for a refund decision ($askRF!$) or a redeem decision ($askRM!$). If σ_3^c is true (resp. σ_4^c), it satisfies σ_4^s, σ_5^r (resp. σ_6^s, σ_3^r) from figures 2(c)2(d). We define a predicate ValidTransfer() as the conjunction of the swap's conditions to allow the transfer of assets. When ValidTransfer() is satisfied, assets are ready to be retrieved by their recipient. We define a second predicate, AbortTransfer(), which characterises the conditions for an asset to be recovered by its source. When AbortTransfer() is satisfied, assets are ready to be recovered by their source. The predicates are mutually exclusive. Both ValidTransfer() and AbortTransfer() predicates are conditioned by a valid *proof-of-action*. Their implementation depends on the protocol detailed later.

Sources. The role of the source is to transfer assets to recipients. Let us introduce the following four predicates: CorrectSwap() is the conjunction of; (1) the source's local graph and the published one are identical and (2) the source's local timeout is not reached. NoDecision() is true when after some time the coordinator has not made a decision. AuthoRM() is true when the coordinator state machine is in "OkRM" state. AuthoRF() is true when the coordinator is in "OkRF" state. Note that, all the predicates, except NoDecision(), need a *proof-of-action* to be satisfied. This part is detailed in the protocol description.

The protocol of a source is in figure 2(c): to start, the source checks the status of the graph. If the graph does not satisfy CorrectSwap(), then it exits the swap (σ_1^s). Otherwise (σ_2^s), it computes the ω_2^s operation and locks its asset a_i and assigns the new owner r_i. Consequently, ϵ_1^a from figure 2(a) of its asset is triggered. Then, the source sends a request message to the coordinator to give a redeem decision through the ϵ_3^s action. The source adds proof that the locked asset operation has been executed properly. Hence, this step allows the coordinator to assess the validity of the $LockAsset(a_i, r_i)$ operation executed by the source.

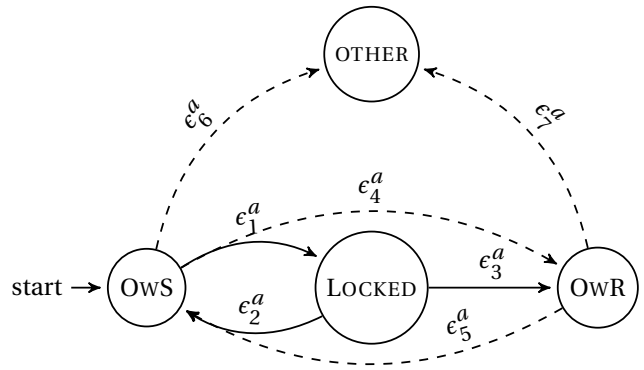
Depending on the coordinator decision, either the source exits the swap if σ_4^s is satisfied, or the source recovers its asset if σ_6^s is satisfied. The source computes the ω_6^s operation to recover its asset. Thereby, it satisfies ϵ_2^a from figure 2(a) of its asset. However, after some time, if no decision has been made, σ_5^s is set to true. The source asks for a refund decision by sending a request message to the coordinator through the ϵ_5^s action.

It is essential to clarify that figure 2(c) represents the source's state machine of one transfer. Indeed, a source may have more than one asset to transfer and must run the protocol for each one. Considering, separately, each of the participants' tasks for each asset simplifies the formalisation without loss of generalisation. Thereby, to help understand the protocol and afterwards helping the formal proof, a source transferring multiple assets will have different identification for each transfer asset. If we take the example of figure 1, A as a source will have the following identification : $\{s_{A_1}, s_{A_3}\}$.

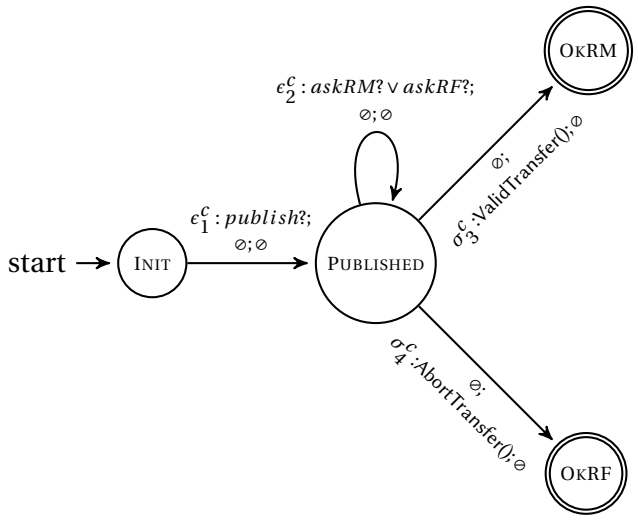
Recipients. The recipient is the new owner of an asset. The predicates defined above, CorrectSwap(), NoDecision(), AuthoRM() and AuthoRF() have the exact definition for recipients. Like the source, the recipient must run the protocol, figure 2(d), for each asset it receives, for the same reason defined above. For example, the participant B from figure 1 as a recipient will have as identification $\{r_{B_1}, r_{B_4}\}$ and the protocol for each one is the following: the recipient starts by checking the status of the graph (σ_1^r and σ_2^r). Depending on the coordinator decision, either the recipient exits the swap if σ_3^r is true, or the recipient retrieves its asset if σ_5^r is true. To retrieve its asset, the recipient computes the ω_5^r operation. Consequently, this triggers ϵ_3^a from figure 2(a) of its asset, and the recipient becomes the new owner. However, if σ_4^r is satisfied, the recipient asks for a refund decision through ϵ_4^r.

3.3 Detailed Description of the Protocol $\mathscr{P}_{swap}$

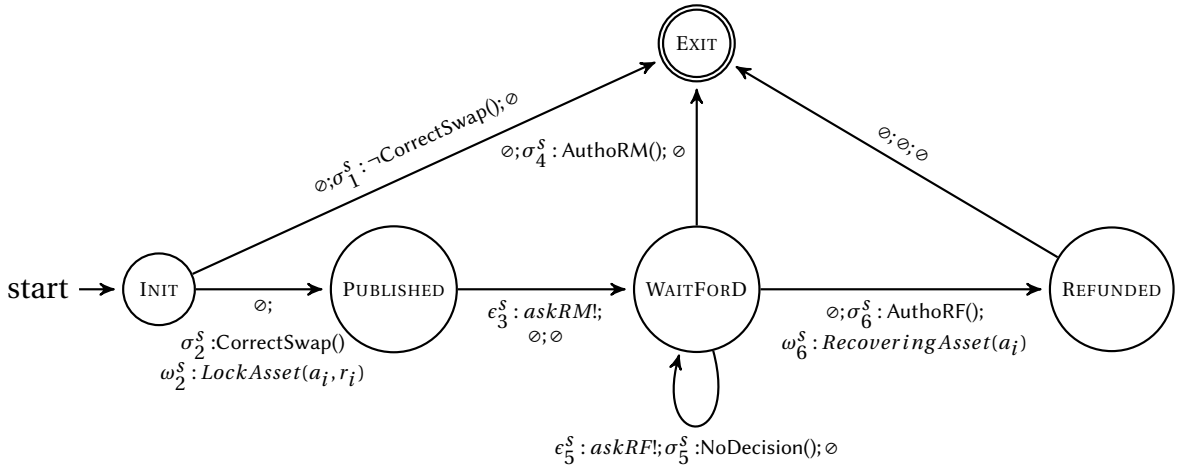
In this part, we describe in detail the different phases of the protocol and the link between *proof-of-actions* and the predicates defined in 3.2.3. Let us recall that *proof-of-actions* allows countering all unacceptable behaviours of the Byzantine participants that may violate the specification.



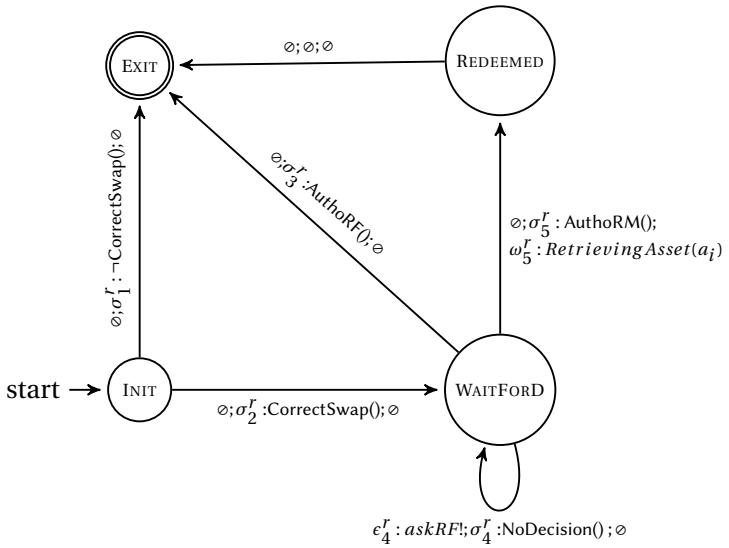
(a) Representation of an asset possible states a_i, with $i \in \{1,...,m\}$ and m the total number of assets



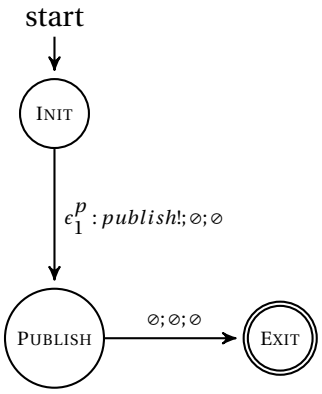
(b) State machine of the coordinator



(c) State machine of a source s_i



(d) State machine of a recipient r_i



(e) State machine of the publisher

Figure 2: Asset representation and participants state machines

The protocol $\mathscr{P}_{swap}$ runs through three phases. Each phase is conditioned by the validation of a *proof-of-action*:

***Phase 1**: proof of graph publication.* In phase 1, participants designate a *publisher* to publish the swap graph to the coordinator (figures 2(e), 2(b)). Each correct participant waits for a *proof-of-action*, let us call it "$Proof_{publish}$", from the coordinator that the graph has been published (figures 2(c), 2(d)). Since all information of the coordinator is public, participants can retrieve "$Proof_{publish}$" and verify its validity; i.e. checking if their local graph and the published one are identical. "$Proof_{publish}$" allows to satisfy CorrectSwap(). The coordinator being public helps prevent a misbehavior from the publisher. If "$Proof_{publish}$" is invalid or the graph has not been published after some time then CorrectSwap() is violated and correct participants will abandon the swap.

***Phase 2**: proof of locking assets.* During phase 2, sources lock their assets. Note that sources need "$Proof_{publish}$" to lock their assets. Indeed, if a source locks an asset before the graph publication, the asset can be locked forever if the Byzantine publisher decides not to publish the graph. Locking operation assigns the new owner of the asset, and only the recipient designated as the new owner can retrieve the asset. Once the asset is locked, each correct source sends a message to the coordinator to request a redeem decision. This request is accompanied by a proof "$Proof_{lock}$", provided by the source, that it successfully computed $LockAsset$. All sources must send a request message accompanied by "$Proof_{lock}$" for each transferred asset; otherwise, the swap cannot be accomplished. The coordinator collects all proofs through the $askRM$! action of all sources and check their validity. If one proof is invalid, then the coordinator aborts the swap by giving a refund decision.

To give a redeem decision, the conditions of the predicate ValidTransfer() are: (1) all sources must request the coordinator to give a redeem decision; (2) all sources' "$Proof_{lock}$" must be valid and verified by the coordinator. If no decision is given after some time, any correct participant can send a refund request.

For example, if a source crashes before sending a redeem request message, any correct participant can ask for a refund decision. A single request message is enough for the coordinator to authorise the refund if no decision has been made previously. Thus, the conditions of the AbortTransfer() predicate are (1) any correct participant asks for a refund decision or (2) at least one "$Proof_{lock}$" is invalid.

***Phase 3**: proof of decision.* In phase 3, participants wait until the coordinator gives any decision. Consequently, if the coordinator gives a redeem decision by updating its state to "OKRM", correct recipients retrieve the proof "$Proof_{redeem}$" that the decision is "redeem". This proof allows to satisfy the predicate AuthoRM(), and correct recipients will be redeemed. If the decision is "refund", correct sources retrieve from the coordinator the proof "$Proof_{refund}$" to be refunded. This proof validates the predicate AuthoRF(). The two proofs are the only way to unlock the assets.

4 TLA+ IMPLEMENTATION

In this section, we prove the protocol $\mathscr{P}_{swap}$ using formal methods. In this paper, we focus on the *model-checking*[4] for proving the liveness properties and the *theorem proving*[17] for the safety property with TLA+[12]. The complete code of the implementation is online[2].

TLA+ Overview. To start, we describe the TLA+ language without going into deep detail. For more information, please refer to [12].

TLA+ is a specification language based on temporal logic. It is used to specify the model of concurrent and distributed systems. The TLA+ language provides a module structure for writing specifications. A system is represented as actions over unprimed variables (old state) and primed variables (new state). Each action states the operations to be carried out and updates the context if required. A system is specified as `Spec = Init /\[][Next]_vars`. The predicate `Init` specifies the possible initial states, `Next` specifies a disjunction of all possible actions of the system and `vars` the tuple of all variables. The expression `[][Next]_vars` means it is always true that either one of the actions defined in `Next` is executed or `vars` is in a state of stuttering. The stuttering is when a variable has the same value in the current and the new states. Consequently, the `Spec` defines a set of infinite sequences of steps, characterising a behaviour, where in each step either an action is true and the state changes or `vars` stutters.

TLA+ toolbox. A model-checker, TLC, is integrated into the TLA+ toolbox and checks the specification by executing every possible behaviour of the system. It builds a finite state model for checking invariance properties, written as an LTL formula. In addition, TLA+ has a proof system TLAPS that mechanically checks proofs of properties.

PlusCal. TLA+ language can become difficult to use if we have no background in TLA+ formalism. In order to make it easy for inexperienced users to use TLA+, PlusCal [13] has been proposed. It is a high-level algorithmic language to generate TLA+ code.

Fair process. A system satisfies a liveness property under fairness assumptions on actions. In a PlusCal algorithm, each label corresponds to an action. An action is enabled iff it can be executed, i.e. a `fair process` cannot stop at that action. Omitting the word `fair` make the process unfair and has no fairness assumptions on its actions (it can behave as a crash process).

4.1 Functions and predicates

In section 2, the swap modelling distinguishes between a participant's operation and action. However, their implementation in the TLA+ language does not make this distinction. An action and an operation are modelled by the definition of a `macro`. Note that the parameters of the functions may vary from 2 because of the TLA+ language. The following functions, written in PlusCal, are actions and operations of participants:

$LockAsset(a_i, r_i)$, source's operation:

```
1 macro lockAsset(self){
2  if (ProofPublish = TRUE /\ self \in Sources /\
3      assets[AofS(self)] = "OwS")
4  assets[AofS(self)]:= "locked"; ProofLock[self]:= TRUE;}
```

`self` is the function caller and `Sources` the set of sources. The primitive `AofS(self)` gives the identifier of `self`'s asset and `assets[]` is a hashtable that maps an asset with its state. The following is the $askRM$! and $askRF$! actions:

```
1 macro askRM(self){
```

[2] https://anonymous.4open.science/r/ICDCN22-D3C5/CrossChain.tla

```
 if (self \in Sources /\ ProofLock[self] = TRUE /\
     coordState = "published") qrm := qrm \union {self};}
macro askRF(self){
 if (coordState = "published") qrf := TRUE;}
```

`coordState` is a variable that describes the coordinator state. `qrm` is a sequence of $askRM$! call function and `qrf` a boolean that is true when a participant calls the $askRF$! function. Below is the $RetrievingAsset(a_i)$ and $RecoveringAsset(a_i)$ operations:

```
macro retrievingAsset(self){
 if (self \in Recipients /\ ProofOkRM = TRUE /\
    assets[AofR(self)] = "locked") assets[AofR(self)] := "OwR";}
macro recoveringAsset(self){
 if (self \in Sources /\ ProofOkRF = TRUE /\
    assets[AofS(self)] ="locked") assets[AofS(self)] := "OwS";}
```

`Recipients` is the set of recipients and `AofR(self)` gives the identifier of the recipient's asset. In the following, we describe additional actions specific to Byzantine:

```
macro otherS(self){
 if (self \in Sources /\ assets[AofS(self)] = "OwS")
 assets[AofS(self)]:= "other";}
macro otherR(self){
 if (self \in Recipients /\ assets[AofR(self)] = "OwR")
 assets[AofR(self)]:= "other";}
macro directToR(self){
 if (self \in Sources /\ assets[AofS(self)] = "OwS")
 assets[AofS(self)]:= "OwR";}
macro directToS(self){
 if (self \in Recipients /\ assets[AofR(self)] = "OwR")
 assets[AofR(self)]:= "OwS";}
```

4.2 Byzantine model

In TLA+, we model Byzantine as unpredictable participants. Hence, we use a non-determinism structure (`[either, or]` statement) in Byzantine processes design. A Byzantine source (resp. recipient) may execute actions and operations of a correct source (resp. recipient) in completely random order. As a result, there exists a run execution of the protocol where Byzantine behaves as a correct participant. The following PlusCal code characterises the process of a Byzantine source. It can execute actions of correct sources lines {5, 6, 7, 8}, and additional actions defined in 4.1, lines {3, 4}.

```
process (BSource \in BSources){
init_bsrc:
either {BdirectToR: directToR(self); goto init_bsrc; }
or {Bother: otherS(self); goto init_bsrc; }
or {BaskRM: askRM(self); goto init_bsrc; }
or {BlockAsset: lockAsset(self); goto init_bsrc; }
or {BSaskRF: askRF(self); goto init_bsrc; }
or {BrecoveringAsset: recoveringAsset(self); goto init_bsrc;};};
```

`BSource` is the process name and `BSources` the set of Byzantine sources.

The following code is the process of a Byzantine recipient. Lines 3 and 4 represent actions of the correct recipient, and lines 6 and 7 represent actions specific to Byzantine recipients.

```
process (BRecipient \in BRecipients){
init_brcp:
either {BRaskRF: askRF(self); goto init_brcp;}
or {BRretrievingAsset: retrievingAsset(self); goto init_brcp;}
or {BRdirectToS: directToS(self); goto init_brcp;}
or {BRother: otherR(self); goto init_brcp;};};
```

`BRecipient` is the process name and `BRecipients` the set of Byzantine recipients. `BRretrievingAsset, BrecoveringAsset, BRother, BRdirectToS, init_bsrc, init_brcp, BdirectToR, BaskRM, BlockAsset, BSaskRF, BRaskRF, Bother` are labels.

As a result, a Byzantine may execute any branch of its code or do nothing, acting like a crashed participant (no `fair` keyword).

A publisher can be Byzantine. What it can do wrong is either publish a wrong graph or do nothing. In both cases, the swap does not take place.

4.3 Proof of the *Safety* Property

The *Consistency* property is a safety property. In the following, we demonstrate the strategy of the proof using TLAPS. The strategy to prove the property is to define an inductive invariant `Inv`. We need to prove that the invariant holds for all states of behaviour. For that, it suffices to prove : (1) The invariant is true in the initial state, (2) if the invariant is true in any state of the behaviour, then it is true in the next state of the behaviour; (3) the *Consistency* is true in all reachable states. The resulting invariant rule is : $\frac{Init \Rightarrow Inv \quad Inv \wedge Next \Rightarrow Inv' \quad Inv \Rightarrow Consistency}{Spec \Rightarrow \Box Consistency}$

The following predicate captures the invariant `Inv`: `TypeOk /\CoordInv`, with `TypeOk` the type correctness invariant and `CoordInv` the predicate that specifies the state of each variable at each coordinator's step. `CoordID` is the identifier of the coordinator.

```
CoordInv ==
/\ pc[CoordID] = "init_c" => init_cInv
/\ pc[CoordID] = "decision" => decisionInv
/\ pc[CoordID] = "decisionValid" => decisionValidInv
/\ pc[CoordID] = "decisionAbort" => decisionAbortInv
/\(pc[CoordID] = "Done" /\ coordState = "okRM") => okRMInv
/\(pc[CoordID] = "Done" /\ coordState = "okRF") => okRFInv
```

`init_c, decision, decisionValid, decisionAbort, Done` are labels of the coordinator process and `pc[]` the program counter variable that tracks which label the process are currently on. `okRMInv, init_cInv, decisionInv, decisionValidInv, decisionAbortInv, okRFInv` are invariants. The following two predicates model the ownership of an asset in TLA+:

```
AvailableS(a) == assets[a] = "OwS" \/
                 (ProofOkRF = TRUE /\ assets[a] = "locked")
AvailableR(a) == assets[a] = "OwR" \/
                 (ProofOkRM = TRUE /\ assets[a] = "locked")
```

`AvailableS(a)` (resp. `AvailableR(a)`) is a predicate that evaluates the asset whether is owned by its source, `assets[a] = "OwS"` (resp. by its recipient, `assets[a] = "OwR"`), or accessible by the source, `ProofOkRF = TRUE /\assets[a] = "locked"` (resp. by the recipient, `ProofOkRM = TRUE /\assets[a] = "locked"`).

Accessible by source or recipient models that any participant that has timeout prematurely will still have the possibility to recover/retrieve its asset asynchronously even if the swap is terminated since the proof of decision will always be available. The TLA+ safety property is:

```
Consistency == \A s \in CSources, r \in CRecipients:
Finish(s,r) => AvailableS(AofS(s)) \/ AvailableR(AofR(r))
```

With `CSources` and `CRecipients` the set of correct sources and correct recipients. The predicate `Finish(s, r)` is true if both `s` and `r` processes have finish their protocol : `Finish(s,r) == pc[s] = "Done" /\pc[r] = "Done"`.

Table 1: Sucess *Ownership*'s model-checking for 6 participants

Rcps \ Srcs	0	1	2	3
0	$11s$ 24 24107	$21s$ 27 216405	$02min17$ 32 1973867	$25min22$ 37 20850284
1	$11s$ 27 46405	$31s$ 28 379715	$04min07$ 32 3483165	$40min27$ 37 39095140
2	$15s$ 30 95475	$50s$ 31 710325	$07min48$ 32 6393275	$01h21min32$ 37 76729100
3	$22s$ 33 247125	$01min22$ 34 1526875	$15min21$ 35 12648125	– – –

4.4 Proof of the *Liveness* Properties

Liveness properties are proven using model-checking. TLAPS is not suited for proving liveness. The properties are the following : *Ownership* and *Retrieving*.

The following predicates are needed to describe the liveness:

```
AllParticipantsAreCorrect == (Pi = Pc) /\ swapGraph = "correct"
AtLeastOneCorrect == Pc # {}
```

`AllParticipantsAreCorrect` is a predicate that is true if all participants (sources, recipients and publisher) are correct. The statement `swapGraph = "correct"` describes that the publisher is correct.

`AtLeastOneCorrect` is a predicate that is true if there is at least one participant correct.

The liveness properties are:

`(A ~> B)` is "syntactic sugar" for `[](A => <>B)`, with the temporal operators `[]`; *always*, `<>`; *eventually* and `=>`; *implies*.

```
OwnershipS == \A s \in AssetsFromCS : AvailableS(s)
OwnershipR == \A r \in AssetsForCR : AvailableR(r)
Ownership == AtLeastOneCorrect ~> (OwnershipS \/ OwnershipR)
Retrieving == AllParticipantsAreCorrect /\ Timeout = FALSE ~>
              (\A r \in Recipients: assets[AofR(r)] = "OwR")
```

With `AssetsFromCS`, the set of assets initially owned by correct sources, and `AssetsForCR` is the set of assets intended for correct recipients.

Table 1 gives some results concerning the model-checking carried out on $\mathscr{P}_{swap}$ for the *Ownership* property on a computer Intel® Core™ i7-8850H CPU @ 2.60GHz × 12. The table gives the proportion of Byzantine recipients and sources in the system. The information in each cell is the time TLC took to model-check, the diameter and the distinct states. The case where all participants are Byzantine (Srcs = 3 and Rcps = 3) has run out of memory with TLC. Though we can trivially deduce that the property holds since the predicate `AtLeastOneCorrect` is `FALSE`.

The *Retrieving* property assumes a synchronous system; hence we suppose that all correct participants will never timeout. This assumption is modelled by `Timeout = FALSE`. The checking is trivial if the predicate on the left side of *Retrieving* holds. A run of TLC with six participants and all correct takes two seconds to succeed.

5 THE INSTANTIATED PROTOCOL $\mathscr{P}_{inst}$

$\mathscr{P}_{inst}$ protocol is an instantiation of the protocol $\mathscr{P}_{swap}$. In this section, we instantiate $\mathscr{P}_{swap}$ in a blockchain environment. The goal is to transfer assets between different participants of different blockchains. Everything that has been described in the abstract protocol applies to the instantiated protocol with the addition of blockchain-specific implementations: smart contracts and consensus mechanisms needed to produce specific *proof-of-actions*.

Smart Contracts. A blockchain system is a distributed ledger that maintains a continuously-growing history of unalterable ordered information organised in a chain of blocks. A smart contract is a self-executing contract running in a blockchain with the terms of the agreement between parties without the need for a central authority. A smart contract is a computer program that contains variables and functions. A smart contract is identified by a unique address. Once a smart contract is published, its location is known to all the blockchain's participants. Each modification of the contract state, e.g. through a function call, generates a transaction. Each created transaction is recorded in the blockchain.

Consensus Mechanisms. The most well-known blockchains, Bitcoin [15] and Ethereum [2], are based on *Proof-of-Work* [15] and *Proof-of-Stake* [20] consensus. Their mechanism for adding blocks to the chain can generate what we call *forks*. The result is that some participants might not have locally the same chain. The rule of the longest chain allows reconciling the blockchain state. However, after reconciliation, some blocks can be revoked. In this case, we say that confirmation is probabilistic.

A committee-based blockchain is a category of blockchain that relies on the BFT consensus mechanism [3]. The block creators are known and clearly defined as the *validators*. For each block, there is a subset of validators, a *committee* that produces that block and sign it. Using deterministic BFT consensus, this category of blockchains offers consistency guarantees that forks will never occur as long as no more than $\frac{1}{3}$ of committee members are Byzantine; hence the blockchain will always have a unique chain. These blockchains guarantee immediate block confirmation, i.e. when a block is added to the chain, it is immediately confirmed. For any decision concerning the validity of a block, a quorum of $2f+1$ validator signatures is needed in the committee. f is the number of participants that can deviate from the protocol. A block signed by quorum is called a certified block. Some examples of such blockchains are HotStuff [23] and Tendermint [11].

5.1 Instantiation of $\mathscr{P}_{swap}$

Forks occurrence is a crucial issue for the adoption of blockchain technologies in critical applications. Thereby, in the instantiated protocol presented in this paper, we have chosen to rely on committee-based blockchains. Thus, all the blockchains involved in the swap are of this category. In the following, we describe how the abstract protocol can be instantiated in a blockchain.

The protocol relies on smart contracts and certified blocks to satisfy the specification of the swap. Throughout the protocol, participants are invoking functions of involved contracts to execute the swap. Therefore, a recorded transaction in a certified block, i.e. which will be signed and validated by at least $2f+1$ validators of the blockchain, constitutes a reliable *proof-of-action* for our protocol; as in [6]. In the following, we detail how smart contracts and certified blocks are used to instantiate $\mathscr{P}_{swap}$ protocol.

5.1.1 Instantiation of Participants. As for the abstract protocol, the set of participants consists of a set of sources, a set of recipients

and a publisher, defined by their state machine in figures 2(c), 2(d), 2(e). A public key address identifies each participant.

The coordinator is represented by a smart contract SC_c implementing the state machine logic introduced in figure 2(b). This contract is used to coordinate the protocol by preventing the occurrence of both *redeem* and *refund* decisions. In $\mathscr{P}_{swap}$, we assume that correct participants can evaluate the coordinator correctness. Delegating the responsibility of the coordinator to a contract makes it possible to satisfy this assumption. Once published on a blockchain, a smart contract can be analysed by everyone. All participants can obtain information from SC_c using its address.

SC_c can be in the following states: "INIT" state, the contract is not yet published. Once the publisher publishes SC_c, its state changes to "PUBLISHED" [3]. This $publish!$ action is a function defined in SC_c. The function inputs are the swap graph and all validators addresses of each blockchain involved in the swap. Moreover, each decision (redeem or refund) is handled by a function contained in SC_c. The function responsible for changing the state of SC_c to "OKRM" has as parameter a proof that the function caller must provide. This proof is the certified block proving the good behaviour of sources. In addition, the function will have as a precondition the predicate ValidTransfer(). Thus, the function can only be executed if the precondition is satisfied. For the function responsible for changing the state to "OKRF" the precondition will be the predicate AbortTransfer(). If the precondition is satisfied, the contract SC_c changes its state to "OKRF".

5.1.2 Instantiation of Locking Asset Operation. A source in the swap is a participant that wants to transfer one or multiple assets. According to $\mathscr{P}_{swap}$, each source has to lock each asset it wants to trade. In $\mathscr{P}_{inst}$, each asset a_i is locked in a unique smart contract SC_{a_i}. For doing this, sources have to publish SC_{a_i} on their corresponding blockchain. The publication of SC_{a_i} sets the new asset's owner r and is accompanied by an additional input detailed in the following. By publishing a contract, sources express their agreement to transfer assets and, it avoids the double spending of the asset. This operation of publishing asset a_i corresponds to the operation $LockAsset(a_i, r)$ defined in section 3.1.

5.1.3 Instatiation of Proof-of-actions. As described in section 3.2, the *proof-of-action* is a mechanism to guard against Byzantine participants. This mechanism can be instantiated in various ways. Our instantiated protocol uses proofs based on certified blocks. Indeed, all blockchains involved in our swap are committee-based blockchains. Thus, a block in the chain is an immediately confirmed block and can no longer be undone; because the block has been validated and signed by a quorum of $2f+1$ validators:

$Proof_{publish}$. Each source needs the block where the proof of SC_c publication can be found to publish its contract and lock its asset. Indeed, the function responsible for the publication of the source contract can be only executed if the proof "$Proof_{publish}$" is valid. The proof must be given as a parameter of the function. The source must wait for SC_c publication before publishing its contract(s) to avoid a forever locking asset. Consequently, if the proof is valid, each asset a_i must be locked in SC_{a_i} by its corresponding source by publishing it.

[3] the choice of the blockchain where SC_c is published is out of the scope.

$Proof_{lock}$. Once the contracts of sources are published, each source requests the redeem decision. This request is a call function from the contract SC_c. The source must give as parameter of the function "$Proof_{lock}$". This proof is the block where SC_{a_i} is located. Notice that if a source transfers multiple assets, it must make for each asset a request. As the coordinator has the list of validators of each blockchain of the swap, it will check if the block has at least $2f+1$ signature of the validators. If this occurs, the redeem decision is given. This condition is embedded in the predicate ValidTransfer(). As a reminder, the conditions are: (1) all sources have requested a redeem decision and (2) all sources' certified block must be valid. An invalid block leads to a refund decision.

$Proof_{redeem,refund}$. Participants wait for SC_c to change state. The block where the change of state is located is used as a proof. Accordingly, if SC_c moves to "OKRM" state, recipients retrieve the proof "$Proof_{redeem}$" and retrieve their asset(s) from the corresponding contract(s). If SC_c moves to "OKRF" state, sources recover their assets from the corresponding contract using the proof "$Proof_{refund}$". Both the retrieve and recover asset operations are possible due to functions in SC_{a_i} that need proof of either redeem or refund decision.

6 STATE OF THE ART

Cross-chain swaps have given rise to several articles on this subject. The one that remains the reference is the atomic cross-chain swap of [5]. In this algorithm, the protocol is based on Hashed Timelocked Contract [21] to transfer assets and, the swap is modelled as a directed graph. [9] proposes a protocol to improve space and local time complexity of [5]'s protocol by using only signatures to set hashed timelocks instead of the graph topology. Both articles guarantee the atomicity property in a synchronous system. As a result, a very slow participant, but following its protocol is considered faulty. Our approach does not allow such result and guarantee that a slow participant will never be worse-off.

[25] have coped with this problem by drawing on a well-known protocol in distributed transactions, namely the two-phase commit [1]. This distributed transaction consists of sub-transactions, and each sub-transaction transfers an asset on some blockchain. The protocol is modelled as a directed graph. Our protocol is strongly inspired by [25], but with significant differences summarises in table 2. They present a solution to the problem of implementing such a swap while aiming to ensure *Atomicity* and *Commitment* properties. They guard against behaviour deviating from the protocol by checking the content of the blockchains participating in the swap. However, the specification does not cover all deviating behaviours. For example, if we consider the swap between *A* and *B*. *A* transfers bitcoins to *B*, which in turn transfers ether to *A*. If the swap is authorised to redeem, *A* safely retrieves the transferred ether. However, imagine that *B* crashes just before being able to retrieve the transferred bitcoins. The sub-transaction that characterises the bitcoin transfer will never occur. Thereby, we face the violation of the *Atomicity* and *Commitment* property

[19] and [6] show that atomicity in a system in the presence of malicious participants cannot be all or nothing. They define an algorithm that can perform cross-chain transfers through intermediaries without asserting atomicity. Both articles define a

Table 2: Comparison between authors

		Zakhary and al.[25]	*Van Glabbeek and al.*[19]	*Herlihy and al.*[6]	$\mathscr{P}_{swap}$
Specification	Protocol-agnostic	✓	✗	✗	✓
	Resilient to Byzantine	✗	✓	✓	✓
	Strong liveness guarantee	✗	✓	✓	✓
Protocol	Formal description	✗	✓	✗	✓
Formal proof	Manual proof	✓	✓	✓	✓
	Semi-automatic proof	✗	✗	✗	✓
Blockchain env	PoW-based blockchain	✓	✓	✓	⚠
	Committee-based blockchain	✗	✗	✓	✓

specification dependent on their protocol. In [6], property 2 assers that no asset belonging to a compliant party is escrowed forever. Although putting assets in escrow is present in most cross-chain protocols, this property makes the specification protocol dependent. The same analysis applies to [19]. Conversely, the $\mathscr{P}_{swap}$ and [25] specification are completely protocol-agnostic. Another point of comparison is the formal description of the protocol. Unlike $\mathscr{P}_{swap}$, [6] and [25] describe the protocol in natural language, without using a formal approach (e.g. by a pseudo-code or an automaton, like in [19]). It is not intuitive to identify the exact behaviour of the protocol participants. In contrast, both articles provide the pseudo-code of the smart contracts involved in the protocol. However, none of the cited articles addresses a semi-automatic proof of their protocol.

Finally, the implementation environment differs from one paper to another. Note that, in our paper, we do not highlight the implementation of the protocol in a PoW-consensus blockchain environment. Although it can be instantiated there, we make a choice not to put it forward. The implementation requires strong assumptions, which in our opinion, do not reflect the reality of the blockchain environment.

7 CONCLUSION

In this paper, we introduce a cross-chain swap that allows the transfer of assets across different distributed ledgers in the presence of Byzantine participants. We propose a protocol sufficiently abstracted to be instantiated in various distributed ledger frameworks. The resulting protocol describes the modelling of the participants in the form of a state machine that eases the proof. A semi-automatic proof of the protocol is given, demonstrating that the abstract protocol satisfies the swap specification. Moreover, we present a way to instantiate the protocol in concrete blockchains systems. At last, the proof of conformity between the abstract protocol and the instantiated protocol are left as future work.

REFERENCES

[1] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems.* Addison-wesley New York.

[2] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).

[3] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*.

[4] E Allen Emerson and Edmund M Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages, and Programming.*

[5] Maurice Herlihy. 2018. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing.*

[6] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2019. Cross-chain deals and adversarial commerce. *Proceedings of the VLDB Endowment* (2019).

[7] Yoichi Hirai. 2018. Blockchains as kripke models: An analysis of atomic cross-chain swap. In *International Symposium on Leveraging Applications of Formal Methods.*

[8] Philipp Hoenisch and Lucas Soriano del Pino. 2021. Atomic Swaps between Bitcoin and Monero. *arXiv:2101.12332* (2021).

[9] Soichiro Imoto, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. 2019. Atomic Cross-Chain Swaps with Improved Space and Local Time Complexity. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems.*

[10] Igor Konnov, Marijana Lazić, Ilina Stoilkovska, and Josef Widder. 2020. Tutorial: Parameterized Verification with Byzantine Model Checker. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems.*

[11] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).

[12] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc.

[13] Leslie Lamport. 2009. The PlusCal algorithm language. In *International Colloquium on Theoretical Aspects of Computing.*

[14] Stephan Merz and Hernán Vanzetto. 2012. Automatic verification of TLA+ proof obligations with SMT solvers. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning.*

[15] Satoshi Nakamoto. 2019. *Bitcoin: A peer-to-peer electronic cash system.* http://www.bitcoin.org/bitcoin.pdf

[16] NomadicLabs. 2021. *Dexter flaw.* https://forum.tezosagora.org/t/dexter-flaw-discovered-funds-are-safe/2742

[17] Alan JA Robinson and Andrei Voronkov. 2001. *Handbook of automated reasoning.* Elsevier and MIT Press.

[18] TierNolan. 2013. *Bitcoin talk forum.* http://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949/

[19] Rob van Glabbeek, Vincent Gramoli, and Pierre Tholoniat. 2019. Cross-Chain Payment Protocols with Success Guarantees. *arXiv:1912.04513* (2019).

[20] Pavel Vasin. 2014. Blackcoin's proof-of-stake protocol v2. *white paper* (2014).

[21] Bitcoin Wiki. 2019. *Hashed timelock contracts.* http://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts

[22] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).

[23] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT consensus in the lens of blockchain. *arXiv:1803.05069* (2018).

[24] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods.*

[25] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2020. Atomic Commitment Across Blockchains. *Proc. VLDB Endow.* (2020).

A RESULTS OF THE *RETRIEVING* MODEL-CHECKING

In this section, we introduce the results of the *Retrieving* model-checking 3. As for the *Ownership* property, the case where all participants are Byzantine (Rcps = 3 and Srcs = 3) runs out of memory. However, since the predicate `AllParticipantsAreCorrect` is false; the property is trivially true.

Table 3: Sucess *Retrieving*'s model-checking results for 6 participants

Rcps \ Srcs	0	1	2	3
0	$02s$ 23 867	$15s$ 30 123494	$01min20$ 34 1406300	$20min20$ 40 18173512
1	$12s$ 28 23010	$25s$ 29 230300	$02min41$ 34 2607290	$39min40$ 39 35349580
2	$14s$ 31 56700	$39s$ 32 477050	$05min30$ 33 5125400	$1h18min56$ 38 72527350
3	$23s$ 34 200500	$01min20$ 35 1240750	$14min31$ 36 11282250	– – –

B MANUAL PROOF

In this section, we will prove manually the swap problem properties defined in section 2. Properties are written as an LTL formula. Let $loc(x)$ be the location state of 'x'. Let $\mathscr{A}_r$ be the set of states that implies "*available to its recipient*" with $\mathscr{A}_r = (loc(a) = \text{"}OwR\text{"} \vee (Proof_{redeem} \wedge loc(a) = \text{"}locked\text{"}))$ and $\mathscr{A}_s$ the set of states that implies "*available to its source*" with $\mathscr{A}_s = (loc(a) = \text{"}OwS\text{"} \vee (Proof_{refund} \wedge loc(a) = \text{"}locked\text{"}))$. Λ is the set of all assets of the swap and Λ_s the set of assets initially owned by correct sources, with $\Lambda_s \subseteq \Lambda$. The set of assets intended for correct recipients is Λ_r, with $\Lambda_r \subseteq \Lambda$. Π_s is the set of sources and Π_r the set of recipients. Let P_c be the set of correct participants with $P_c \subseteq \Pi$. Let us denote 'c' the coordinator and 'p' the publisher.

LEMMA B.1. *When the coordinator is in its initial state, then no correct sources are in published state and, assets initially owned by a correct source are owned by their source.*

Formally: $loc(c) = \text{"INIT"} \implies \forall s \in (P_c \cap \Pi_s) : loc(s) \notin \text{"PUBLISHED"} \wedge \forall a \in \Lambda_s : loc(a) = \text{"OwS"}$.

PROOF. From figure 2(b) we can see that in the initial state, the coordinator has not triggered ϵ_1^c. Hence, no correct sources and correct recipients (figures 2(c) and 2(d)) will have their guard σ_2^s and σ_2^r satisfied. However, σ_1^s and σ_1^r can be satisfied if the publisher takes a long time to trigger ϵ_1^p. Consequently, correct participants can exit the swap. In both scenarios, correct sources would not lock their assets and, these remain owned by their source. □

LEMMA B.2. *When the coordinator is in "PUBLISHED" state, then no assets initially owned by a correct source are available to their recipient.*

Formally: $loc(c) = \text{"PUBLISHED"} \implies \forall a \in \Lambda_s : loc(a) \notin \mathscr{A}_r$.

PROOF. When the coordinator is in "PUBLISHED" state, then ϵ_1^p has been triggered by the publisher in figure 2(e), allowing the coordinator to change its state. Consequently, correct participants will verify "$Proof_{publish}$" and if the proof is valid, then correct sources could lock their assets (executing ω_2^s) and trigger ϵ_1^a from figure 2(a). Since the coordinator is in "PUBLISHED" state, neither σ_2^c nor σ_3^c is satisfied. Thereby, no decision has been taken by the coordinator. It is therefore not possible for an asset to be available to the recipient as long as the coordinator is in the "PUBLISHED" state. □

LEMMA B.3. *When the coordinator gives a redeem decision, then all assets are available to their recipient.*

Formally: $loc(c) = \text{"OKRM"} \implies \forall a \in \Lambda : loc(a) \in \mathscr{A}_r$.

PROOF. For the coordinator to make a redeem decision, σ_3^c from figure 2(b) must be satisfied. ValidTransfer() is satisfied when all sources have executed the action ϵ_3^s from figure 2(c) and "$Proof_{lock}$" provided by the sources to the coordinator are correct and valid. Consequently, satisfying σ_3^c makes all assets accessible to their recipients. Depending on the recipient behaviour, assets can stay in "LOCKED" state or move to "OWR" state by using "$Proof_{redeem}$" to satisfy AuthoRM(). In both cases, the assets are available to their recipient. If the recipient is correct, then its asset will eventually be retrieved by executing ω_5^r. □

LEMMA B.4. *When the coordinator gives a refund decision, then assets initially owned by a correct source are available to their source.*

Formally: $loc(c) = \text{"OKRF"} \implies \forall a \in \Lambda_s : loc(a) \in \mathscr{A}_s$.

PROOF. For the coordinator to make a refund decision, σ_4^c from figure 2(b) must be satisfied. Hence, the conditions for AbortTransfer() predicate are fulfilled. Namely, either a "$Proof_{lock}$" provided by a source has been proven invalid, or there exists a participant who asks for a refund decision (triggering ϵ_5^s if the participant is a source or triggering ϵ_4^r if the participant is a recipient). Consequently, σ_4^c is satisfied, and all assets initially owned by a correct source are now available to their sources. Hence, depending on source behaviour, assets can stay in "LOCKED" state or move to "OWS" state by using "$Proof_{refund}$" to satisfy AuthoRF(). Both cases set the assets available to their source. If the source is correct, then its asset will eventually be recovered by executing ω_6^s. □

THEOREM B.5. *For any correct source s_1 of an edge $e_1 = (s_1, a_1, r_1)$ and correct recipient r_2 of an edge $e_2 = (s_2, a_2, r_2)$, at the end of their execution, either s_1 owns a_1 or r_2 owns a_2.*

PROOF. We have proven from Lemma B.1 that a *correct* source s_1 can timeout and finish its execution before locking its asset a_1. Consequently, a_1 remains in "OWS" state. Lemma B.3 proves that a *correct* recipient r_2 can finish its execution by retrieving its asset a_2. The asset's state changes to "OWR". However, though r_2 can timeout at the beginning of the swap (before the swap graph publication), when the redeem decision is given, a_2 is accessible by the recipients and can retrieve a_2 asynchronously since the decision will always be available. From Lemma B.4, s_1 finishes its execution by recovering its asset. Consequently, a_1's state is "OWS".

We can see that we can extrapolate this result to all *correct* sources and recipients from the swap.

From Lemma B.1, Lemma B.2 and Lemma B.4, we have proven that no assets initially owned by a correct source can be available to their recipient if no redeem authorisation is given. However, an asset can be owned by a recipient if the source of that asset is Byzantine. Indeed, a Byzantine source that behaves arbitrarily can transfer its asset directly to the recipient; without waiting for the coordinator decision. From Lemma B.3, we have proven that the assets may be available to the recipients only when the coordinator

authorises the swap by giving the redeem decision. Moreover, this decision is only possible if all the sources are correct up to the moment of the locking assets.

Therefore, we proved that considering each possible end of execution of s1 and r2; the outcome is that s1 owns its asset or r2 owns its asset. Hence, the *Consistency* property of the swap is proven. □

LEMMA B.6. *If "Proof$_{publish}$" is valid and at least one participant is correct, then the coordinator eventually makes a decision.*

Formally: $\exists p \in P_c : Proof_{publish} \wedge P_c \neq \{\} \implies \Diamond(loc(c) = \text{"OKRF"} \vee loc(c) = \text{"OKRM"})$

PROOF. Once the swap graph is published and validated by the correct participants, the coordinator has only two possibilities of decision, redeem or refund. These two decisions are possible to achieve depending on the actions of the participants. If the coordinator is in "PUBLISHED" state for a while without evolving, then it suffices to have only one correct participant to detect it. Assuming this scenario, the correct participant; whether the source or the recipient will be in "WAITFORD" state. After reaching the participant's timeout, the predicate NoDecision() will be satisfied (σ_5^s if the participant is a source and σ_4^r if the participant is a recipient). Thereby, this allows the participant to request a refund decision from the coordinator (ϵ_5^s or ϵ_4^r). The operation of asking refund satisfies the predicate AbortTransfer() and leads to the decision of the coordinator for a refund authorisation. Moreover, if all participants are correct, then all sources will lock their asset and will give a valid "*Proof $_{lock}$*" to the coordinator. Hence, conditions of ValidTransfer() will be satisfied and lead to the authorisation of the coordinator for a redeem decision.

□

LEMMA B.7. *If the coordinator authorizes the refund, then no asset initially owned by a correct source is ownerless forever.*

Formally: $loc(c) = \text{"OKRF"} \implies \forall a \in \Lambda_s : \Diamond(loc(a) \neq \text{"LOCKED"})$

PROOF. If the coordinator has authorised the refund, then the predicate AbortTransfer() has been satisfied (see Lemma B.4). As a result, assets return to their sources (ϵ_2^a is satisfied); hence all assets initially owned by a correct source become available to their source. A correct source will retrieve from the coordinator "*Proof $_{refund}$*". A valid proof satisfies σ_6^s and, a correct source will be able to recover its assets by executing ω_6^s and become the owner again. If the source is Byzantine, it might never recover its asset, thus leaving the asset ownerless. In addition, the Byzantine source could lock its asset out of the swap with no way to recover it. These two situations are acceptable and satisfy the property. □

LEMMA B.8. *If the coordinator authorises the redeem, then no asset intended for a correct recipient is ownerless forever.*

Formally: $loc(c) = \text{"OKRM"} \implies \forall a \in \Lambda_r : \Diamond(loc(a) \neq \text{"LOCKED"})$

PROOF. If the coordinator has authorised the redeem, then the predicate ValidTransfer() has been satisfied (see Lemma B.3). In the redeeming case, all assets become available to the recipient. A correct recipient will retrieve from the coordinator the proof "*Proof $_{redeem}$*". A valid proof satisfies σ_5^r and, a correct recipient only has to retrieve the asset by executing ω_5^r and making the asset owned by the recipient (ϵ_3^a). However, if a Byzantine recipient decides not to get its asset back, then that asset will be ownerless. It is an acceptable situation and satisfies the property. □

THEOREM B.9. *No asset owned initially by a correct source is ownerless forever or no asset intended to be transferred to a correct recipient is ownerless forever.*

PROOF. From Lemma B.6 we have proven that, if "*Proof $_{publish}$*" is valid, it only takes one correct participant in our system for the coordinator to end up issuing a decision. Moreover, if all participants are Byzantine, then the theorem is satisfied. From Lemma B.7, we have proven that correct sources will not lose their asset. However, no conclusions are possible for assets owned by Byzantine sources. From Lemma B.8 the same assumption has been proven for assets intended for correct recipients. Likewise, no conclusions are possible for assets intended for Byzantine recipients. As a result, we have proven the *Ownership* property of the swap. □

THEOREM B.10. *If all participants are correct, then all recipients will retrieve their intended assets.*

PROOF. If all participants are correct, then they will all execute their protocol within the bounded time limits. The swap graph will be published and be correct (a valid "*Proof $_{publish}$*") and all sources will request the coordinator for a redeem decision, providing a valid "*Proof $_{lock}$*". Consequently, the coordinator will authorise the swap and recipients will eventually be redeemed using "*Proof $_{redeem}$*". Moreover, if a subset of participants is Byzantine, $P_c = \Pi$ will equal false. Since false implies true, the property is still satisfied. □

C TLA+ CODE

```
1  ---------------- MODULE FaultyCrossChain ----------------
2  EXTENDS TLAPS, Integers, Sequences, TLC, FiniteSets
3  Integers   == Nat \ {0}
4  CONSTANT NTxs, Correct
5
6  \* NTxs is the number of transactions (number of traded assets)
7  \* Correct is the number of correct participants
8
9  \* AStates: asset's states
10 \* CStates: coordinator's states
11 \* PStates: Publisher's states
12 \* SwapStates: swap graph states
13 \* AStates, AvailableR, CStates, PStates, SwapStates
14 \* are DEFINITION for tlaps
15 \* PublisherID is the identifier of the publisher
16 \* CoordinatorID is the identifier of the coordinator
17
18 AStates       == { "OwS", "OwR", "locked", "other" }
19 CStates       == { "init", "published", "okRM", "okRF" }
20 PStates       == { "init", "publish" }
21 SwapStates    == { "init", "correct", "different" }
22 PublisherID   == -1
23 CoordinatorID == 0
24
25 \* Sources: the set of source's id
26 \* Assets: the set of asset's id
27 \* Recipients: the set of recipient's id
28 \* Pi: the set of all sources and recipients
```

```
\* Pc: the set of correct sources and recipients
\* CSources: the set of correct sources
\* CRecipients: the set of correct recipients
\* BSources: the set of byzantine sources
\* BRecipients: the set of byzantine recipients

Sources          == {3*x-2   : x \in 1..NTxs}
Assets           == {3*x-1   : x \in 1..NTxs}
Recipients       == {3*x     : x \in 1..NTxs}

Pi               == Sources \cup Recipients
Pc               == Pi \cap Correct

CSources         == Pc \cap Sources
CRecipients      == Pc \cap Recipients
BSources         == Sources \ CSources
BRecipients      == Recipients \ CRecipients

\* AofS(x): function that gives the asset id of the source 'x'
\* AofR(x): function that gives the asset id of the recipient 'x'
\* SofA(x): function that gives the source id of the asset 'x'
\* RofA(x): function that gives the recipient id of the asset 'x'
\* AssetsFromCS: set of assets initially owned by a correct source
\* AssetsForCR: set of assets intended for a correct recipient

AofS(x) == x + 1
AofR(x) == x - 1
SofA(x) == x - 1
RofA(x) == x + 1

AssetsFromCS     == {AofS(x) : x \in CSources}
AssetsForCR      == {AofR(x) : x \in CRecipients}

(*

--fair algorithm ACC {

\* assets: list of all assets initialise to "OwS" state
\* pState: Publisher state initialises to "init"
\* coordState: coordinator state initialises to "init"
\* qrm: sequence of ask redeem call function
\* qrf: sequence of ask refund call function
\* swapGraph: swap graph state initialises to "init"
\* ProofPublish, ProofLock, ProofOkRM and ProofOkRF are
\* proof-of-action describes in the article
\* assets, pState, coordState, qrm, qrf, swapGraph and
\* proof-of-action are VARIABLES for tlaps

variable
        assets          = [a \in Assets |-> "OwS"],
        pState          = "init",
        coordState      = "init",
        qrm = {},
        qrf = FALSE,
        swapGraph = "init",
        ProofPublish = FALSE,
        ProofLock =[c \in Sources |-> FALSE],
        ProofOkRM = FALSE,
        ProofOkRF = FALSE;

\* the define block can be used in pluscal code and
\* in properties
\* The operator returns true if the expression is true
\* ValidTransfer and AbortTransfer corresponds to the
\* predicates of the same name in the article
\* qrm = Sources -> all sources has been asked for
\* a redeem authorisation
\* qrf = true -> at least one participant has asked for
\* a refund authorisation

define
{
    ValidTransfer == qrm = Sources
    AbortTransfer == qrf = TRUE
}


(***************************************************************)
(* Macros are the general catch-all code inliner. *)
(* lockAsset(self): locks the asset of the source 'self'*)
(* and set to true the proof of action ProofLock[self] *)
(***************************************************************)

macro lockAsset(self) {
 if (ProofPublish = TRUE /\ self \in Sources /\
     assets[AofS(self)] = "OwS")
   assets[AofS(self)] := "locked";
   ProofLock[self] := TRUE;}

(***************************************************************)
(* askRM(self): When a source asks for a redeem *)
(* authorisation, its lock proof has to be valid and *)
(* the coordinator has to be in the published state *)
(***************************************************************)

macro askRM(self) {
 if (self \in Sources /\ ProofLock[self] = TRUE /\
     coordState = "published")
   qrm := qrm \union {self};}

(***************************************************************)
(* retrievingAsset(self): the macro updates the assets' *)
(* state to OwR *)
(***************************************************************)

macro retrievingAsset(self) {
  if (self \in Recipients /\ ProofOkRM = TRUE /\
      assets[AofR(self)] = "locked")
    assets[AofR(self)] := "OwR"}

(***************************************************************)
(* askRF(self): any participant can ask for a refund *)
(* authorisation *)
(***************************************************************)

macro askRF(self) {
  if (coordState = "published")
    qrf := TRUE;}

(***************************************************************)
(* recoveringAsset(self): the macro updates the assets' *)
(* state to OwS *)
(***************************************************************)

macro recoveringAsset(self) {
 if (self \in Sources /\ ProofOkRF = TRUE /\
     assets[AofS(self)] ="locked")
   assets[AofS(self)] := "OwS"}

(***************************************************************)
(* other(self) and directToR(self) are actions that *)
(* byzantine can do *)
(* other(self) describes the behavior of a byzantine *)
(* where it can do anything with its asset. *)
(* directToR(self) describes the direct send of its asset*)
(* to the recipient without waiting for the coordinator's*)
(* decision *)
(***************************************************************)
```

```
167
168 macro other(self) {
169  if (self \in Sources /\ assets[AofS(self)] = "OwS")
170     assets[AofS(self)]:= "other";}
171
172 macro otherR(self) {
173  if (self \in Recipients /\ assets[AofR(self)] = "OwR")
174     assets[AofR(self)]:= "other";}
175
176 macro directToR(self) {
177  if (self \in Sources /\ assets[AofS(self)] = "OwS")
178  assets[AofS(self)]:= "OwR"}
179
180 macro directToS(self) {
181  if (self \in Recipients /\ assets[AofR(self)] = "OwR")
182  assets[AofR(self)]:= "OwS"}
183
184 (*************************************************************)
185 (* We define the following processes: (1) the Publisher. *)
186 (* (2) the coordinator (3) correct sources, *)
187 (* (4) byzantine sources, (5) correct recipients and *)
188 (* (6) byzantine recipients *)
189 (* The Publisher has -1 as an random identifier,random *)
190 (* it publish the swap graph, by changing its state to *)
191 (* "publish". We assume that the publisher can be *)
192 (* byzantine. Hence, the graph is either correct or *)
193 (* different from the graph constructed by the *)
194 (* participants of the swap. *)
195 (* The publisher can halt, even if an action is enabled *)
196 (* stay in "init_p" forever and stutters. A process that *)
197 (* crashes is modelled by having stuttering steps *)
198 (*************************************************************)
199
200       process (Publisher = PublisherID)
201     {
202         init_p :
203                either {
204                     pState := "publish";
205                     either swapGraph := "correct";
206                     or swapGraph := "different";
207                       }
208                or skip;
209     };
210
211 (*************************************************************)
212 (* The coordinator has 0 as an random identifier *)
213 (* init_c : The coordinator waits for the Publisher to *)
214 (* publish the graph to updates its state. When the *)
215 (* state is updated, the proof of published is set to true*)
216 (* decision: the decision is either redeem (if *)
217 (* ValidTransfer is true) or the decision is refund (if *)
218 (* AbortTransfer is true). *)
219 (* decisionValid: the coordinator updates to okRM state *)
220 (* and the ProofOkRM is available for recipients to *)
221 (* retrieve their assets *)
222 (* decisionAbort: the coordinator updates to okRF state *)
223 (* and the ProofOkRF is available for sources to recover *)
224 (* their assets. *)
225 (* the coordinator is a correct entity. We add to the *)
226 (* process a fairness condition that the process cannot *)
227 (* stop at a non-blocking action. *)
228 (*************************************************************)
229
230   fair   process (Coordinator = CoordinatorID)
231     {
232 init_c:
233            await pState = "publish" /\ swapGraph # "init";
234            coordState := "published";
235            ProofPublish := TRUE;
236 decision:
237            either {
238                await ValidTransfer;
239                decisionValid:
240                coordState := "okRM";
241                ProofOkRM := TRUE;
242                goto Done;
243                   }
244            or {
245                await AbortTransfer;
246                decisionAbort:
247                coordState := "okRF";
248                ProofOkRF := TRUE;
249                goto Done;
250               };
251     };
252
253 (*************************************************************)
254 (* Source: multiprocess of CSources processes (correct sources*)
255 (* init_src: sources waits for the swap graph to be published *)
256 (* either the swap is different (in this case the source *)
257 (* leaves the swap) or the graph is correct and the *)
258 (* ProofPublish of the coordinator is valid *)
259 (* if the graph is correct, correct sources can lock their *)
260 (* asset (lock) and asks for a redeem decision (published). *)
261 (* waitForD: source waits for the coordinator decision *)
262 (* exit_src: when the decision is given, sources exit the swap*)
263 (*************************************************************)
264
265  fair  process (Source \in CSources )
266 {
267 init_src :
268   either { await swapGraph = "different" \/ TRUE;
269                    goto Done;}
270   or     { await ProofPublish = TRUE /\ swapGraph = "correct";
271 lock:              lockAsset(self);
272
273 published:         askRM(self );
274
275 waitForD:          either { await ProofOkRM = TRUE;
276                             goto Done;}
277                    or { await ProofOkRF = TRUE;
278                         recoveringAsset(self);
279                         goto Done;}
280                    or {\* the case where NoDecision is true
281                        await coordState = "published";
282                        askRF(self);
283                        goto waitForD;};
284                     };
285 };
286
287 (*************************************************************)
288 (* BSource is a multiprocess of BSources processes (byzantine *)
289 (* sources). Since a byzantine behavior cannot be predicted, *)
290 (* we use the either statement to express the non determinisme*)
291 (* of a byzantine. a byzantine can execute the actions of a *)
292 (* source in a completely random order in addition to the *)
293 (* actions directToR and other. *)
294 (* As a result, it has the ability to run the protocol *)
295 (* correctly and behaves as a correct source. The process is *)
296 (* unfair, thus we do not add the 'fair' statement before *)
297 (* process. We assume that the process can crash at anytime *)
298 (*************************************************************)
299
300 process (BSource \in BSources)
301 {
302 init_bsrc:
303   {
```

```
  either { BdirectToR: directToR(self); goto init_bsrc; }
  or { Bother: other(self); goto init_bsrc; }
  or {BaskRM: askRM(self ); goto init_bsrc; }
  or { BlockAsset: lockAsset(self); goto init_bsrc; }
  or { BSaskRF: askRF(self); goto init_bsrc; }
  or {BrecoveringAsset: recoveringAsset(self); goto init_bsrc;};
  }
};

(*************************************************************)
(* Recipient: multiprocess of CRecipients processes (correct *)
(* recipients) *)
(* init_rcp: either the swap is correct or different *)
(* waitForD_rcp: the recipient waits for the coordinator *)
(* decision. *)
(*************************************************************)

 fair process (Recipient \in CRecipients)
{
init_rcp :
       either { await swapGraph = "different" \/ TRUE;
                goto Done;}
       or { await ProofPublish = TRUE /\ swapGraph = "correct";
waitForD_rcp:
              either { await ProofOkRF = TRUE;
                       goto Done;}
              or { await ProofOkRM = TRUE;
                   retrievingAsset(self);
                   goto Done;}
              or { await coordState = "published";
                   askRF(self);
                   goto waitForD_rcp;};
};
};

(*************************************************************)
(* BRecipient is a multiprocess of BRecipients processes *)
(* (byzantine recipient). As byzantine sources, a byzantine *)
(* recipient behavior cannot be predicted. a byzantine can *)
(* execute the actions of a recipient in a completely random *)
(* order in addition to the actions. *)
(* As a result, it has the ability to run the protocol *)
(* correctly and behaves as a correct recipient. The process *)
(* is unfair,thus we do not add the 'fair' statement before *)
(* process. We assume that the process can crash at anytime *)
(*************************************************************)

process (BRecipient \in BRecipients)
{
init_brcp:
either {BRaskRF: askRF(self); goto init_brcp;}
or {BRretrievingAsset: retrievingAsset(self); goto init_brcp;}
or {BRdirectToS: directToS(self); goto init_brcp;}
or {BRother: otherR(self); goto init_brcp;};
};
};
*)
\* BEGIN TRANSLATION
VARIABLES assets, pState, coordState, qrm, qrf, swapGraph,
 ProofPublish, ProofLock, ProofOkRM, ProofOkRF, pc

(* define statement *)
ValidTransfer == qrm = Sources
AbortTransfer == qrf = TRUE


vars == << assets, pState, coordState, qrm, qrf, swapGraph,
ProofPublish, ProofLock, ProofOkRM, ProofOkRF, pc >>

ProcSet == {PublisherID} \cup {CoordinatorID} \cup (CSources)
           \cup (BSources) \cup (CRecipients) \cup (BRecipients)

Init == (* Global variables *)
        /\ assets = [a \in Assets |-> "OwS"]
        /\ pState = "init"
        /\ coordState = "init"
        /\ qrm = {}
        /\ qrf = FALSE
        /\ swapGraph = "init"
        /\ ProofPublish = FALSE
        /\ ProofLock = [c \in Sources |-> FALSE]
        /\ ProofOkRM = FALSE
        /\ ProofOkRF = FALSE
        /\ pc = [self \in ProcSet |->
            CASE self = PublisherID -> "init_p"
                 [] self = CoordinatorID -> "init_c"
                 [] self \in CSources -> "init_src"
                 [] self \in BSources -> "init_bsrc"
                 [] self \in CRecipients -> "init_rcp"
                 [] self \in BRecipients -> "init_brcp"]

init_p == /\ pc[PublisherID] = "init_p"
          /\ \/ /\ pState' = "publish"
                /\ \/ /\ swapGraph' = "correct"
                      \/ /\ swapGraph' = "different"
             \/ /\ TRUE
                /\ UNCHANGED <<pState, swapGraph>>
          /\ pc' = [pc EXCEPT ![PublisherID] = "Done"]
          /\ UNCHANGED << assets, coordState, qrm, qrf,
              ProofPublish, ProofLock, ProofOkRM, ProofOkRF >>

Publisher == init_p

init_c == /\ pc[CoordinatorID] = "init_c"
          /\ pState = "publish" /\ swapGraph # "init"
          /\ coordState' = "published"
          /\ ProofPublish' = TRUE
          /\ pc' = [pc EXCEPT ![CoordinatorID] = "decision"]
          /\ UNCHANGED << assets, pState, qrm, qrf, swapGraph,
               ProofLock, ProofOkRM, ProofOkRF >>

decision == /\ pc[CoordinatorID] = "decision"
            /\ \/ /\ ValidTransfer
                  /\ pc' = [pc EXCEPT ![CoordinatorID] =
                                      "decisionValid"]
               \/ /\ AbortTransfer
                  /\ pc' = [pc EXCEPT ![CoordinatorID] =
                                      "decisionAbort"]
            /\ UNCHANGED << assets, pState, coordState,
               qrm, qrf, swapGraph, ProofPublish, ProofLock,
               ProofOkRM, ProofOkRF >>

decisionValid == /\ pc[CoordinatorID] = "decisionValid"
                 /\ coordState' = "okRM"
                 /\ ProofOkRM' = TRUE
                 /\ pc' = [pc EXCEPT ![CoordinatorID] = "Done"]
                 /\ UNCHANGED << assets, pState, qrm, qrf,
                swapGraph, ProofPublish, ProofLock, ProofOkRF >>

decisionAbort == /\ pc[CoordinatorID] = "decisionAbort"
                 /\ coordState' = "okRF"
                 /\ ProofOkRF' = TRUE
                 /\ pc' = [pc EXCEPT ![CoordinatorID] = "Done"]
                 /\ UNCHANGED << assets, pState, qrm, qrf,
                 swapGraph, ProofPublish, ProofLock, ProofOkRM >>

Coordinator ==
  init_c \/ decision \/ decisionValid \/ decisionAbort
```

```
init_src(self) == /\ pc[self] = "init_src"
                  /\ \/ /\ swapGraph = "different" \/ TRUE
                        /\ pc' = [pc EXCEPT ![self] = "Done"]
                     \/ /\ ProofPublish = TRUE
                        /\ swapGraph = "correct"
                        /\ pc' = [pc EXCEPT ![self] = "lock"]
                  /\ UNCHANGED << assets, pState, coordState,
                  qrm, qrf, swapGraph, ProofPublish, ProofLock,
                            ProofOkRM, ProofOkRF >>

lock(self) == /\ pc[self] = "lock"
              /\ IF ProofPublish = TRUE /\ self \in Sources /\
                    assets[AofS(self)] = "OwS"
                 THEN /\ assets' =
                         [assets EXCEPT ![AofS(self)] = "locked"]
                      /\ ProofLock' =
                         [ProofLock EXCEPT ![self] = TRUE]
                 ELSE /\ TRUE
                      /\ UNCHANGED << assets, ProofLock >>
              /\ pc' = [pc EXCEPT ![self] = "published"]
              /\ UNCHANGED << pState, coordState, qrm, qrf,
              swapGraph, ProofPublish, ProofOkRM, ProofOkRF >>

published(self) == /\ pc[self] = "published"
                   /\ IF self \in Sources
                      /\ ProofLock[self] = TRUE
                      /\ coordState = "published"
                         THEN /\ qrm' = (qrm \union {self})
                         ELSE /\ TRUE
                              /\ qrm' = qrm
                   /\ pc' = [pc EXCEPT ![self] = "waitForD"]
                   /\ UNCHANGED << assets, pState, coordState,
                   qrf, swapGraph, ProofPublish, ProofLock,
                   ProofOkRM, ProofOkRF >>

waitForD(self) == /\ pc[self] = "waitForD"
                  /\ \/ /\ ProofOkRM = TRUE
                        /\ pc' = [pc EXCEPT ![self] = "Done"]
                        /\ UNCHANGED <<assets, qrf>>
                     \/ /\ ProofOkRF = TRUE
                        /\ IF self \in Sources
                           /\ ProofOkRF = TRUE
                           /\ assets[AofS(self)] ="locked"
                              THEN /\ assets' =
                                      [assets EXCEPT ![AofS(self)] = "OwS"]
                              ELSE /\ TRUE
                                   /\ UNCHANGED assets
                        /\ pc' = [pc EXCEPT ![self] = "Done"]
                        /\ qrf' = qrf
                     \/ /\ coordState = "published"
                        /\ IF coordState = "published"
                              THEN /\ qrf' = TRUE
                              ELSE /\ TRUE
                                   /\ qrf' = qrf
                        /\ pc' = [pc EXCEPT ![self] = "waitForD"]
                        /\ UNCHANGED assets
                  /\ UNCHANGED << pState, coordState, qrm,
                  swapGraph, ProofPublish, ProofLock, ProofOkRM,
                                  ProofOkRF >>

Source(self) == init_src(self) \/ lock(self) \/ published(self)
                 \/ waitForD(self)

init_bsrc(self) ==
            /\ pc[self] = "init_bsrc"
            /\ \/ /\ pc' = [pc EXCEPT ![self] = "BdirectToR"]
               \/ /\ pc' = [pc EXCEPT ![self] = "Bother"]
               \/ /\ pc' = [pc EXCEPT ![self] = "BaskRM"]
               \/ /\ pc' = [pc EXCEPT ![self] = "BlockAsset"]
               \/ /\ pc' = [pc EXCEPT ![self] = "BSaskRF"]
               \/ /\ pc' = [pc EXCEPT ![self] = "BrecoveringAsset"]
            /\ UNCHANGED << assets, pState, coordState, qrm,
            qrf, swapGraph, ProofPublish, ProofLock,
                                  ProofOkRM, ProofOkRF >>

BdirectToR(self) == /\ pc[self] = "BdirectToR"
                    /\ IF self \in Sources
                    /\ assets[AofS(self)] = "OwS"
                          THEN /\ assets' =
                          [assets EXCEPT ![AofS(self)] = "OwR"]
                          ELSE /\ TRUE
                               /\ UNCHANGED assets
                    /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
                    /\ UNCHANGED << pState, coordState, qrm,
                   qrf, swapGraph, ProofPublish, ProofLock,
                   ProofOkRM, ProofOkRF >>

Bother(self) == /\ pc[self] = "Bother"
                /\ IF self \in Sources
                /\ assets[AofS(self)] = "OwS"
                      THEN /\ assets' =
                      [assets EXCEPT ![AofS(self)] = "other"]
                      ELSE /\ TRUE
                           /\ UNCHANGED assets
                /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
                /\ UNCHANGED << pState, coordState, qrm, qrf,
                swapGraph, ProofPublish, ProofLock, ProofOkRM,
                 ProofOkRF >>

BaskRM(self) == /\ pc[self] = "BaskRM"
                /\ IF self \in Sources
                /\ ProofLock[self] = TRUE
                /\ coordState = "published"
                      THEN /\ qrm' = (qrm \union {self})
                      ELSE /\ TRUE
                           /\ qrm' = qrm
                /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
                /\ UNCHANGED << assets, pState, coordState,
                qrf, swapGraph, ProofPublish, ProofLock,
                ProofOkRM, ProofOkRF >>

BlockAsset(self) == /\ pc[self] = "BlockAsset"
                    /\ IF ProofPublish = TRUE
                    /\ self \in Sources
                    /\ assets[AofS(self)] = "OwS"
                        THEN /\ assets' =
                        [assets EXCEPT ![AofS(self)] = "locked"]
                             /\ ProofLock' =
                        [ProofLock EXCEPT ![self] = TRUE]
                        ELSE /\ TRUE
                             /\ UNCHANGED << assets, ProofLock >>
                    /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
                    /\ UNCHANGED << pState, coordState, qrm,
                    qrf, swapGraph, ProofPublish, ProofOkRM,
                     ProofOkRF >>

BSaskRF(self) == /\ pc[self] = "BSaskRF"
                 /\ IF coordState = "published"
                       THEN /\ qrf' = TRUE
                       ELSE /\ TRUE
                            /\ qrf' = qrf
                 /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
                 /\ UNCHANGED << assets, pState, coordState,
                 qrm, swapGraph, ProofPublish, ProofLock,
                 ProofOkRM, ProofOkRF >>

BrecoveringAsset(self) == /\ pc[self] = "BrecoveringAsset"
```

```
                          /\ IF self \in Sources
                          /\ ProofOkRF = TRUE
                          /\ assets[AofS(self)] ="locked"
                              THEN /\ assets' =
                               [assets EXCEPT ![AofS(self)] = "OwS"]
                              ELSE /\ TRUE
                                   /\ UNCHANGED assets
                          /\ pc' =
                          [pc EXCEPT ![self] = "init_bsrc"]
                          /\ UNCHANGED << pState, coordState,
                         qrm, qrf, swapGraph, ProofPublish,
                         ProofLock, ProofOkRM, ProofOkRF >>

BSource(self) == init_bsrc(self) \/ BdirectToR(self)
   \/ Bother(self) \/ BaskRM(self) \/ BlockAsset(self) \/
 BSaskRF(self) \/ BrecoveringAsset(self)

init_rcp(self) == /\ pc[self] = "init_rcp"
                  /\ \/ /\ swapGraph = "different" \/ TRUE
                        /\ pc' = [pc EXCEPT ![self] = "Done"]
                     \/ /\ ProofPublish = TRUE
                        /\ swapGraph = "correct"
                        /\ pc' =
                        [pc EXCEPT ![self] = "waitForD_rcp"]
                  /\ UNCHANGED << assets, pState, coordState,
                   qrm, qrf, swapGraph, ProofPublish, ProofLock,
                                ProofOkRM, ProofOkRF >>

waitForD_rcp(self) == /\ pc[self] = "waitForD_rcp"
                      /\ \/ /\ ProofOkRF = TRUE
                            /\ pc' = [pc EXCEPT ![self] = "Done"]
                            /\ UNCHANGED <<assets, qrf>>
                         \/ /\ ProofOkRM = TRUE
                            /\ IF self \in Recipients
                            /\ ProofOkRM = TRUE
                            /\ assets[AofR(self)] = "locked"
                                  THEN /\ assets' =
                                 [assets EXCEPT ![AofR(self)] = "OwR"]
                                  ELSE /\ TRUE
                                       /\ UNCHANGED assets
                            /\ pc' = [pc EXCEPT ![self] = "Done"]
                            /\ qrf' = qrf
                         \/ /\ coordState = "published"
                            /\ IF coordState = "published"
                                  THEN /\ qrf' = TRUE
                                  ELSE /\ TRUE
                                       /\ qrf' = qrf
                            /\ pc' =
                           [pc EXCEPT ![self] = "waitForD_rcp"]
                            /\ UNCHANGED assets
                      /\ UNCHANGED << pState, coordState, qrm,
                      swapGraph, ProofPublish, ProofLock,
                      ProofOkRM, ProofOkRF >>

Recipient(self) == init_rcp(self) \/ waitForD_rcp(self)

init_brcp(self) ==
     /\ pc[self] = "init_brcp"
     /\ \/ /\ pc' = [pc EXCEPT ![self] = "BRaskRF"]
        \/ /\ pc' = [pc EXCEPT ![self] = "BRretrievingAsset"]
        \/ /\ pc' = [pc EXCEPT ![self] = "BRdirectToS"]
        \/ /\ pc' = [pc EXCEPT ![self] = "BRother"]
     /\ UNCHANGED << assets, pState, coordState, qrm, qrf,
        swapGraph, ProofPublish, ProofLock, ProofOkRM,
        ProofOkRF >>

BRaskRF(self) == /\ pc[self] = "BRaskRF"
                 /\ IF coordState = "published"
                       THEN /\ qrf' = TRUE
                       ELSE /\ TRUE
                            /\ qrf' = qrf
                 /\ pc' = [pc EXCEPT ![self] = "init_brcp"]
                 /\ UNCHANGED << assets, pState, coordState, qrm,
                  swapGraph, ProofPublish, ProofLock,
                  ProofOkRM, ProofOkRF >>

BRretrievingAsset(self) == /\ pc[self] = "BRretrievingAsset"
                           /\ IF self \in Recipients
                           /\ ProofOkRM = TRUE
                           /\ assets[AofR(self)] = "locked"
                                 THEN /\ assets' =
                                [assets EXCEPT ![AofR(self)] = "OwR"]
                                 ELSE /\ TRUE
                                      /\ UNCHANGED assets
                           /\ pc' =
                          [pc EXCEPT ![self] = "init_brcp"]
                           /\ UNCHANGED << pState, coordState,
                          qrm, qrf, swapGraph, ProofPublish,
                           ProofLock,ProofOkRM, ProofOkRF >>

BRdirectToS(self) == /\ pc[self] = "BRdirectToS"
                     /\ IF self \in Recipients
                     /\ assets[AofR(self)] = "OwR"
                           THEN /\ assets' =
                         [assets EXCEPT ![AofR(self)] = "OwS"]
                           ELSE /\ TRUE
                                /\ UNCHANGED assets
                     /\ pc' = [pc EXCEPT ![self] = "init_brcp"]
                     /\ UNCHANGED << pState, coordState, qrm,
                     qrf, swapGraph, ProofPublish, ProofLock,
                      ProofOkRM, ProofOkRF >>

BRother(self) == /\ pc[self] = "BRother"
                 /\ IF self \in Recipients
                 /\ assets[AofR(self)] = "OwR"
                       THEN /\ assets' =
                     [assets EXCEPT ![AofR(self)] = "other"]
                       ELSE /\ TRUE
                            /\ UNCHANGED assets
                 /\ pc' = [pc EXCEPT ![self] = "init_brcp"]
                 /\ UNCHANGED << pState, coordState, qrm,
                 qrf, swapGraph, ProofPublish, ProofLock,
                 ProofOkRM, ProofOkRF >>

BRecipient(self) == init_brcp(self) \/ BRaskRF(self)
        \/ BRretrievingAsset(self) \/ BRdirectToS(self)
                    \/ BRother(self)

(* Allow infinite stuttering to prevent deadlock on termination*)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == Publisher \/ Coordinator
          \/ (\E self \in CSources: Source(self))
          \/ (\E self \in BSources: BSource(self))
          \/ (\E self \in CRecipients: Recipient(self))
          \/ (\E self \in BRecipients: BRecipient(self))
          \/ Terminating

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)
        /\ WF_vars(Coordinator)
        /\ \A self \in CSources : WF_vars(Source(self))
        /\ \A self \in CRecipients : WF_vars(Recipient(self))

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION
```

```
(*************** Liveness Property**************************)
(*************** Checked using Model-Checking***************)

AvailableS(a) == assets[a] = "OwS" \/
                 (ProofOkRF = TRUE /\ assets[a] = "locked")
AvailableR(a) == assets[a] = "OwR" \/
                 (ProofOkRM = TRUE /\ assets[a] = "locked")
AllParticipantsAreCorrect == (Pi = Pc) /\ swapGraph = "correct"

AtLeastOneCorrect == Pc # {} /\ (swapGraph \in SwapStates)
Finish(s,r) == pc[s] = "Done" /\ pc[r] = "Done"

OwnershipS ==  (\A s \in AssetsFromCS:  AvailableS(s))
OwnershipR ==  (\A r \in AssetsForCR :  AvailableR(r))
Ownership ==   AtLeastOneCorrect => <>  (\/ OwnershipS
                                         \/ OwnershipR)

Retrieving == AllParticipantsAreCorrect
              ~> (\A r \in Recipients : assets[AofR(r)]= "OwR")

(*********************************************************)
(**************** Consistency property********************)

Consistency == \A s \in CSources, r \in CRecipients:
Finish(s,r) => AvailableS(AofS(s)) \/ AvailableR(AofR(r))

(*************** Safety ****************************)
(***************** Proved using TLAPS ******************)

\* the following is a theorem about all sets of the model
\* that are needed to ensure safety of our algorithm.

THEOREM SetsTheorem ==
      /\ CoordinatorID # PublisherID
      /\ \A a \in AssetsFromCS : a \in Assets
      /\ \A a \in AssetsForCR : a \in  Assets
      /\ \A s \in Sources : s \in Pi
      /\ \A r \in Recipients : r \in Pi
      /\ \A p \in Pc: /\ p \in Pi
                      /\ \/ /\ p \in CSources
                            /\ p \in Sources
                         \/ /\ p \in CRecipients
                            /\ p \in Recipients
                      /\ p \notin BSources
                      /\ p \notin BRecipients
      /\ \A s \in CSources : /\ s \in Sources
                             /\ s \in Pi
                             /\ s \in Pc
                             /\ s \notin BSources
                             /\ s # PublisherID
                             /\ s # CoordinatorID
                             /\ s \notin CRecipients
                             /\ s \notin Recipients
                             /\ s \notin BRecipients
                             /\ AofS(s) \in AssetsFromCS
      /\ \A r \in CRecipients : /\ r \in Recipients
                                /\ r \in Pi
                                /\ r \in Pc
                                /\ r \notin BRecipients
                                /\ r # PublisherID
                                /\ r # CoordinatorID
                                /\ AofR(r) \in AssetsForCR
                                /\ SofA(AofR(r)) \in Sources
      /\ \A bs \in BSources : /\ bs \in Pi
                              /\ bs \in Sources
                              /\ bs \notin CSources
                              /\ bs \notin Pc
                              /\ bs # PublisherID
                              /\ bs # CoordinatorID
                              /\ AofS(bs) \notin AssetsFromCS
      /\ \A br \in BRecipients: /\ br \in Recipients
                                /\ br \in Pi
                                /\ br \notin Pc
                                /\ br \notin CRecipients
                                /\ br # PublisherID
                                /\ br # CoordinatorID
                                /\ AofR(br) \notin AssetsForCR
      /\ ProcSet = {PublisherID} \cup {CoordinatorID} \cup
       (CSources) \cup (BSources) \cup (CRecipients) \cup
        (BRecipients)
      /\ Pi = Sources \cup Recipients
      /\ Pc = Pi \cap Correct
      /\ CSources = Pc \cap Sources
      /\ CRecipients  = Pc \cap Recipients
      /\ BSources  = Sources \ CSources
      /\ BRecipients  = Recipients \ CRecipients
      /\ BSources \cap CSources = {}
      /\ BRecipients \cap CRecipients = {}
      /\ AStates = {"OwS", "OwR", "locked", "other"}
      /\ CStates   = {"init", "published", "okRM", "okRF"}
      /\ PStates   = {"init", "publish"}
      /\ SwapStates  = {"init", "correct", "different"}
      /\ \A s \in Sources: SofA(AofS(s)) = s
      /\ \A s \in Recipients: RofA(AofR(s)) = s
      /\ \A s \in Assets: AofS(SofA(s)) = s
      /\ \A s \in Assets: AofR(RofA(s)) = s
      /\ \A s \in Sources: AofS(s) \in Assets
      /\ \A a \in Assets: SofA(a) \in Sources
BY  DEF ProcSet, CSources, CRecipients, Sources, Recipients,
AssetsFromCS, Assets, AssetsForCR, AofS, AofR, SofA, RofA,
Pi, Pc, BSources, BRecipients, PublisherID, CoordinatorID,
AStates, CStates, PStates, SwapStates


\* the following predicate is a type correctness invariant

TypeOk == /\ assets \in [Assets -> AStates]
          /\ pState \in PStates
          /\ coordState \in CStates
          /\ ProofLock \in [Sources -> \in BOOLEAN]
          /\ ProofPublish \in \in BOOLEAN
          /\ ProofOkRM \in BOOLEAN
          /\ ProofOkRF \in BOOLEAN
          /\ qrm \subseteq Sources
          /\ qrf \in \in BOOLEAN
          /\ swapGraph \in SwapStates
          /\ pc[CoordinatorID] \in
        { "init_c", "decision", "decisionValid",
        "decisionAbort", "Done" }
          /\ pc[CoordinatorID] = "Done" =>
             coordState \in { "okRM", "okRF" }
          /\ pc \in [ ProcSet -> { "init_c", "decision",
           "decisionValid", "decisionAbort", "Done",
           "init_p", "init_src" , "lock", "published",
           "waitForD", "refunded",  "Done", "init_bsrc",
           "BdirectToR", "Bother", "BaskRM", "BlockAsset",
           "BSaskRF", "BrecoveringAsset", "init_rcp",
           "waitForD_rcp", "redeemed", "exit_rcp", "Done",
           "init_brcp", "BRaskRF", "BRretrievingAsset",
           "BRdirectToS", "BRother" } ]

\* the following predicates are needed to define the
\* coordinator invariant

init_cInv ==
            /\ coordState = "init"
            /\ ProofOkRM = FALSE
```

```
            /\ ProofOkRF = FALSE
            /\ ProofPublish = FALSE
            /\ qrf = FALSE
            /\ qrm = {}
            /\ \A s \in Sources: ProofLock[s] = FALSE
            /\ \A s \in CSources  :
                    /\ pc[s] \in {"init_src",  "Done"}
                    /\ ProofLock[s] = FALSE
                    /\ assets[AofS(s)] = "OwS"
            /\ \A r \in CRecipients  :
                   pc[r] \in {"init_rcp", "Done"}
            /\ swapGraph = "init" => pState = "init"
            /\ swapGraph ="correct" => pState = "publish"
            /\ swapGraph = "different" => pState = "publish"
            /\ \A a \in AssetsFromCS: assets[a] = "OwS"
            /\ pState = "publish" => pc[PublisherID] = "Done"

decisionInv ==
            /\ coordState = "published"
            /\ pState \in {"publish", "Done"}
            /\ ProofOkRM = FALSE
            /\ ProofOkRF = FALSE
            /\ ProofPublish = TRUE
            /\ pc[PublisherID] = "Done"
            /\ \A s \in Sources:
                /\ s \in qrm => ProofLock[s] = TRUE
                /\ ProofLock[s] = TRUE =>
                  assets[AofS(s)] = "locked"
            /\ \A s \in CSources :
                 /\ pc[s] \in
         {"published", "waitForD", "init_src", "lock", "Done"}
                 /\ pc[s] \in {"published", "waitForD"}
                    => /\ ProofLock[s] = TRUE
                       /\ assets[AofS(s)] = "locked"
                 /\ pc[s] \in {"init_src", "lock", "Done"}
                    => /\ ProofLock[s] = FALSE
                       /\ assets[AofS(s)] = "OwS"
                 /\ pc[s] \in
         {"init_src", "lock", "Done", "published" }
                    => s \notin qrm
                 /\ s \in qrm => pc[s] = "waitForD"
            /\ \A r \in CRecipients : pc[r] \in
               {"init_rcp", "waitForD_rcp", "Done"}
            /\ \A a \in AssetsFromCS: assets[a] \in
               {"locked", "OwS"}

decisionValidInv ==
            /\ coordState = "published"
            /\ pState \in {"publish", "Done"}
            /\ ProofOkRM = FALSE
            /\ ProofOkRF = FALSE
            /\ ProofPublish = TRUE
            /\ pc[PublisherID] = "Done"
            /\ qrm = Sources
            /\ \A s \in Sources:
                    /\ ProofLock[s] = TRUE
                    /\ assets[AofS(s)] = "locked"
            /\ \A s \in CSources: pc[s] \in {"waitForD"}
            /\ \A r \in CRecipients :
                    /\ pc[r] \in
              {"init_rcp", "waitForD_rcp", "Done"}
                    /\ assets[AofR(r)] = "locked"
                    /\ pc[r] = "init_src"
                       =>  assets[AofR(r)] = "locked"
            /\ qrm = Sources
               => \A a \in Assets : assets[a] = "locked"

decisionAbortInv ==
            /\ coordState = "published"
            /\ pState \in {"publish", "Done"}
            /\ ProofOkRM = FALSE
            /\ ProofOkRF = FALSE
            /\ ProofPublish = TRUE
            /\ pc[PublisherID] = "Done"
            /\ qrf = TRUE
            /\ \A s \in CSources :
                 /\ assets[AofS(s)] \in { "locked", "OwS" }
                 /\ pc[s] \in
          {"init_src", "lock", "published", "waitForD", "Done"}
                 /\ pc[s] = "Done" => assets[AofS(s)] = "OwS"
                 /\ pc[s] = "init_src"
                    => assets[AofS(s)] = "OwS"
            /\ \A a \in AssetsFromCS:
                 assets[a] \in {"locked", "OwS"}

okRMInv ==
            /\ coordState = "okRM"
            /\ ProofOkRM = TRUE
            /\ ProofOkRF = FALSE
            /\ ProofPublish = TRUE
            /\ qrm = Sources
            /\ pc[PublisherID] = "Done"
            /\ \A s \in CSources:
                 pc[s] \in {"waitForD", "Done"}
            /\ \A r \in CRecipients :
                 /\ pc[r] \in
            {"init_rcp", "waitForD_rcp", "Done"}
                 /\ assets[AofR(r)] \in { "locked", "OwR" }
                 /\ pc[r] = "Done"
                    => assets[AofR(r)] \in {"OwR", "locked"}
                 /\ pc[r] \in { "init_rcp", "waitForD_rcp"}
                    => assets[AofR(r)] = "locked"
                 /\ pc[r] = "init_src"
                    => assets[AofR(r)] = "locked"
            /\ qrm = Sources
               => \A a \in AssetsForCR :
                 assets[a] \in {"locked", "OwR"}

okRFInv ==
            /\ ProofOkRM = FALSE
            /\ ProofOkRF = TRUE
            /\ ProofPublish = TRUE
            /\ pc[PublisherID] = "Done"
            /\ qrf = TRUE
            /\ \A s \in CSources :
               /\ assets[AofS(s)] \in { "locked", "OwS" }
               /\ pc[s] = "Done" => assets[AofS(s)] = "OwS"
               /\ pc[s] = "init_src" => assets[AofS(s)] = "OwS"
               /\ \A a \in AssetsFromCS:
                 assets[a] \in { "locked", "OwS" }

CoordInv2 ==
    /\ pc[CoordinatorID] = "init_c" => init_cInv
    /\ pc[CoordinatorID] = "decision" => decisionInv
    /\ pc[CoordinatorID] = "decisionValid" => decisionValidInv
    /\ pc[CoordinatorID] = "decisionAbort" => decisionAbortInv
    /\ (pc[CoordinatorID] = "Done" /\ coordState = "okRM" )
       => okRMInv
    /\ (pc[CoordinatorID] = "Done" /\ coordState = "okRF" )
       => okRFInv

(* the inductive invariant for proving the safety property*)
Inv == TypeOk  /\ CoordInv2

THEOREM InitImpliesTypeOk ==
    ASSUME Init
    PROVE TypeOk
  <1>1. assets \in [Assets -> AStates]
```

```
    BY SetsTheorem DEF TypeOk, Init
  <1>2. pState \in PStates
    BY SetsTheorem DEF TypeOk, Init
  <1>3. coordState \in CStates
    BY SetsTheorem DEF TypeOk, Init
  <1>4. ProofLock \in [Sources -> { TRUE, FALSE } ]
    BY SetsTheorem DEF TypeOk, Init
  <1>5. ProofPublish \in { TRUE, FALSE }
    BY SetsTheorem DEF TypeOk, Init
  <1>6. ProofOkRM \in BOOLEAN
    BY SetsTheorem DEF TypeOk, Init
  <1>7. ProofOkRF \in BOOLEAN
    BY SetsTheorem DEF TypeOk, Init
  <1>8. qrm \subseteq Sources
    BY SetsTheorem DEF TypeOk, Init
  <1>9. qrf \in { TRUE, FALSE }
    BY SetsTheorem DEF TypeOk, Init
  <1>10. swapGraph \in SwapStates
    BY SetsTheorem DEF TypeOk, Init
  <1>11. pc[CoordinatorID] = "Done"
  => coordState \in { "okRM", "okRF" }
    BY SetsTheorem DEF TypeOk, Init
  <1>12. pc[CoordinatorID] \in { "init_c", "decision",
   "decisionValid", "decisionAbort", "Done" }
    BY SetsTheorem DEF TypeOk, Init
  <1>13. pc \in [ ProcSet -> { "init_c", "decision",
   "decisionValid", "decisionAbort", "Done",
    "init_p", "init_src" , "lock", "published", "waitForD",
     "refunded",  "Done", "init_bsrc" , "BdirectToR", "Bother",
      "BaskRM", "BlockAsset", "BSaskRF", "BrecoveringAsset",
    "init_rcp" , "waitForD_rcp", "redeemed", "exit_rcp",
    "Done", "init_brcp", "BRaskRF", "BRretrievingAsset",
     "BRdirectToS", "BRother" } ]
    BY SetsTheorem DEF TypeOk, Init
  <1>14. QED
    BY <1>1, <1>10, <1>11, <1>12, <1>13, <1>2, <1>3,
    <1>4, <1>5, <1>6, <1>7, <1>8, <1>9 DEF TypeOk

THEOREM InitImpliesCoord ==
    ASSUME TypeOk,
           Init
    PROVE CoordInv2
  BY SetsTheorem DEF TypeOk, Init, CoordInv2, init_cInv

THEOREM InitImpliesInv ==
    ASSUME Init
    PROVE TypeOk /\ CoordInv2
BY InitImpliesCoord, InitImpliesTypeOk

THEOREM TypeOkInvariant ==
    ASSUME TypeOk,
           Next
    PROVE TypeOk'
  <1>1. CASE Publisher
    BY <1>1, SetsTheorem DEF TypeOk, Publisher, init_p
  <1>2. CASE Coordinator
    <2>1. CASE init_c
      BY <2>1, <1>2, SetsTheorem DEF TypeOk, init_c
    <2>2. CASE decision
      BY <2>2, <1>2, SetsTheorem DEF TypeOk, decision

    <2>3. CASE decisionValid
      BY <2>3, <1>2, SetsTheorem DEF TypeOk, decisionValid
    <2>4. CASE decisionAbort
      BY <2>4, <1>2, SetsTheorem DEF TypeOk, decisionAbort
    <2>7. QED
      BY <1>2, <2>1, <2>2, <2>3, <2>4 DEF Coordinator

  <1>3. CASE \E self \in CSources: Source(self)
    <2> SUFFICES ASSUME NEW self \in CSources,
                        Source(self)
                 PROVE  TypeOk'
      BY <1>3
    <2>1. CASE init_src(self)
      BY <2>1, <1>3, SetsTheorem DEF TypeOk, init_src
    <2>2. CASE lock(self)
      BY <2>2, <1>3, SetsTheorem DEF TypeOk, lock
    <2>3. CASE published(self)
      BY <2>3, <1>3, SetsTheorem DEF TypeOk, published
    <2>4. CASE waitForD(self)
      BY <2>4, <1>3, SetsTheorem DEF TypeOk, waitForD
    <2>7. QED
      BY <1>3, <2>1, <2>2, <2>3, <2>4 DEF Source

  <1>4. CASE \E self \in BSources: BSource(self)
    <2> SUFFICES ASSUME NEW self \in BSources,
                        BSource(self)
                 PROVE  TypeOk'
      BY <1>4
    <2>1. CASE init_bsrc(self)
      BY <2>1, <1>4, SetsTheorem DEF TypeOk, init_bsrc
    <2>2. CASE BdirectToR(self)
      BY <2>2, <1>4, SetsTheorem DEF TypeOk, BdirectToR
    <2>3. CASE Bother(self)
      BY <2>3, <1>4, SetsTheorem DEF TypeOk, Bother
    <2>4. CASE BaskRM(self)
      BY <2>4, <1>4, SetsTheorem DEF TypeOk, BaskRM
    <2>5. CASE BlockAsset(self)
      BY <2>5, <1>4, SetsTheorem DEF TypeOk, BlockAsset
    <2>6. CASE BSaskRF(self)
      BY <2>6, <1>4, SetsTheorem DEF TypeOk, BSaskRF
    <2>7. CASE BrecoveringAsset(self)
      BY <2>7, <1>4, SetsTheorem DEF TypeOk, BrecoveringAsset
    <2>8. QED
      BY <1>4, <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7
      DEF BSource

  <1>5. CASE \E self \in CRecipients: Recipient(self)
    <2> SUFFICES ASSUME NEW self \in CRecipients,
                        Recipient(self)
                 PROVE  TypeOk'
      BY <1>5
    <2>1. CASE init_rcp(self)
      BY <2>1, <1>5, SetsTheorem DEF TypeOk, init_rcp
    <2>2. CASE waitForD_rcp(self)
      BY <2>2, <1>5, SetsTheorem DEF TypeOk, waitForD_rcp
    <2>5. QED
      BY <1>5, <2>1, <2>2 DEF Recipient

  <1>6. CASE \E self \in BRecipients: BRecipient(self)
    <2> SUFFICES ASSUME NEW self \in BRecipients,
                        BRecipient(self)
                 PROVE  TypeOk'
      BY <1>6
    <2>1. CASE init_brcp(self)
      BY <2>1, <1>6, SetsTheorem DEF TypeOk, init_brcp
    <2>2. CASE BRaskRF(self)
      BY <2>2, <1>6, SetsTheorem DEF TypeOk, BRaskRF
    <2>3. CASE BRretrievingAsset(self)
      BY <2>3, <1>6, SetsTheorem DEF TypeOk, BRretrievingAsset
    <2>4. CASE BRdirectToS(self)
      BY <2>4, <1>6, SetsTheorem DEF TypeOk, BRdirectToS
    <2>5. CASE BRother(self)
      BY <2>5, <1>6, SetsTheorem DEF TypeOk, BRother
    <2>6. QED
      BY <1>6, <2>1, <2>2, <2>3, <2>4, <2>5 DEF BRecipient

  <1>7. CASE Terminating
```

```
    BY <1>7 DEF TypeOk, Terminating, vars
  <1>8. QED
    BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6, <1>7 DEF Next

THEOREM CoordInvariant ==
    ASSUME CoordInv2, TypeOk, TypeOk',
           Next
    PROVE CoordInv2'
  <1>1. CASE pc[CoordinatorID] = "init_c"
    <2>0. init_cInv
      BY <1>1 DEF CoordInv2
    <2>1. CASE Publisher
      BY <1>1, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
       init_p, init_cInv, CoordInv2
    <2>2. CASE Coordinator
      <3>1. CASE init_c
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk,
          init_c, init_cInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk,
          init_c, init_cInv, decisionInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk,
          init_c, init_cInv
        <4>4. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk,
           init_c, init_cInv
        <4>5. ((pc[CoordinatorID] = "Done" /\  coordState="okRM")
            => okRMInv)'
          BY <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c,
           init_cInv, CoordInv2
        <4>6. ((pc[CoordinatorID] = "Done" /\  coordState="okRF")
           => okRFInv)'
          BY <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c,
           init_cInv, CoordInv2
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
      <3>2. CASE decision
        BY <1>1, <3>2, <2>2, SetsTheorem DEF TypeOk, decision
      <3>3. CASE decisionValid
        BY <1>1, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid
      <3>4. CASE decisionAbort
        BY <1>1, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
      <3>7. QED
        BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

    <2>3. CASE \E self \in CSources: Source(self)
      <3> SUFFICES ASSUME NEW self \in CSources,
                          Source(self)
                   PROVE  CoordInv2'
        BY <2>3
      <3>1. CASE init_src(self)
        BY <3>1, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk,
        init_src, init_cInv, CoordInv2
      <3>2. CASE lock(self)
        BY <3>2, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk,
        lock, init_cInv, CoordInv2
      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk,
         published, init_cInv, CoordInv2
      <3>4. CASE waitForD(self)
        BY <3>4, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk,
        waitForD, init_cInv, CoordInv2
      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, init_cInv, CoordInv2
      <3>2. CASE BdirectToR(self)
        BY <3>2, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
         BdirectToR, init_cInv, CoordInv2
      <3>3. CASE Bother(self)
        BY <3>3, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
        Bother, init_cInv, CoordInv2
      <3>4. CASE BaskRM(self)
        BY <3>4, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
        BaskRM, init_cInv, CoordInv2
      <3>5. CASE BlockAsset(self)
        BY <3>5, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
         BlockAsset, init_cInv, CoordInv2
      <3>6. CASE BSaskRF(self)
        BY <3>6, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
        BSaskRF, init_cInv, CoordInv2
      <3>7. CASE BrecoveringAsset(self)
        BY <3>7, <2>0, <1>1, <2>4, SetsTheorem DEF TypeOk,
         BrecoveringAsset, init_cInv, CoordInv2
      <3>8. QED
        BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
        DEF BSource

    <2>5. CASE \E self \in CRecipients: Recipient(self)
      <3> SUFFICES ASSUME NEW self \in CRecipients,
                          Recipient(self)
                   PROVE  CoordInv2'
        BY <2>5
      <3>1. CASE init_rcp(self)
        BY <3>1, <1>1, <2>0,<2>5, SetsTheorem DEF TypeOk,
        init_rcp, init_cInv, CoordInv2
      <3>2. CASE waitForD_rcp(self)
        BY <3>2, <1>1, <2>0,<2>5, SetsTheorem DEF TypeOk,
         waitForD_rcp, init_cInv, CoordInv2
      <3>5. QED
        BY <2>5, <3>1, <3>2 DEF Recipient

    <2>6. CASE \E self \in BRecipients: BRecipient(self)
      <3> SUFFICES ASSUME NEW self \in BRecipients,
                          BRecipient(self)
                   PROVE  CoordInv2'
        BY <2>6
      <3>1. CASE init_brcp(self)
        BY <3>1, <1>1, <2>0,<2>6, SetsTheorem DEF TypeOk,
        init_brcp, init_cInv, CoordInv2
      <3>2. CASE BRaskRF(self)
        BY <3>2, <1>1, <2>0,<2>6, SetsTheorem DEF TypeOk,
        BRaskRF, init_cInv, CoordInv2
      <3>3. CASE BRretrievingAsset(self)
        BY <3>3, <1>1, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRretrievingAsset, init_cInv, CoordInv2
      <3>4. CASE BRdirectToS(self)
        BY <3>4, <1>1, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRdirectToS, init_cInv, CoordInv2
      <3>5. CASE BRother(self)
        BY <3>5, <1>1, <2>0,<2>6, SetsTheorem DEF TypeOk,
        BRother, init_cInv, CoordInv2
      <3>6. QED
        BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

    <2>7. CASE Terminating
```

```
      BY <1>1, <2>0,<2>7, SetsTheorem DEF TypeOk, Terminating,
       CoordInv2
    <2>8. QED
      BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

  <1>2. CASE pc[CoordinatorID] = "decision"
    <2>0. decisionInv
      BY <1>2 DEF CoordInv2
    <2>1. CASE Publisher
      BY <1>2, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
       init_p, decisionInv, CoordInv2
    <2>2. CASE Coordinator
      <3>1. CASE init_c
        BY <1>2, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
      <3>2. CASE decision
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision, decisionInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision, decisionInv, ValidTransfer, decisionValidInv
        <4>4. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision, decisionInv, AbortTransfer, decisionAbortInv
        <4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM")
         => okRMInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF")
          => okRFInv)'
          BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,
           decision
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>3. CASE decisionValid
        BY <1>2, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid
      <3>4. CASE decisionAbort
        BY <1>2, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
      <3>7. QED
        BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

    <2>3. CASE \E self \in CSources: Source(self)
      <3> SUFFICES ASSUME NEW self \in CSources,
                          Source(self)
                   PROVE  CoordInv2'
        BY <2>3
      <3>1. CASE init_src(self)
        BY <3>1, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
        init_src, decisionInv, CoordInv2
      <3>2. CASE lock(self)
        <4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]
            BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
            lock, decisionInv
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>2, <2>0, <1>2, <2>3, <4>0, SetsTheorem
          DEF TypeOk, lock, decisionInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
          lock, decisionInv, decisionValidInv
        <4>4. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
          lock, decisionInv, decisionValidInv
        <4>5. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
        => okRMInv)'
          BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
         => okRFInv)'
          BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionInv
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
         published, decisionInv, CoordInv2
      <3>4. CASE waitForD(self)
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv, decisionValidInv
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
        => okRMInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv, decisionValidInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
         => okRFInv)'
          BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionInv
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, decisionInv, CoordInv2
      <3>2. CASE BdirectToR(self)
        <8>1. CASE assets[AofS(self)] = "OwS"
          <9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]
          BY <8>1, <3>2, <2>0, <1>2, <2>4, SetsTheorem
          DEF TypeOk, BdirectToR, decisionInv, CoordInv2
          <9>2. QED
          BY <9>1, <8>1, <3>2, <2>0, <1>2, <2>4, SetsTheorem
          DEF TypeOk, BdirectToR, decisionInv, CoordInv2
        <8>2. CASE assets[AofS(self)] # "OwS"
        BY <8>2, <3>2, <2>0, <1>2, <2>4, SetsTheorem
        DEF TypeOk, BdirectToR, decisionInv, CoordInv2
        <8>3. QED
        BY <8>1, <8>2
      <3>3. CASE Bother(self)
        <8>1. CASE assets[AofS(self)] = "OwS"
```

```
            <9>1. assets' = [assets EXCEPT ![AofS(self)] = "otherL"]
            BY <8>1, <3>3, <2>0, <1>2, <2>4, SetsTheorem
            DEF TypeOk, Bother, decisionInv, CoordInv2
            <9>2. QED
            BY <9>1, <8>1, <3>3, <2>0, <1>2, <2>4, SetsTheorem
            DEF TypeOk, Bother, decisionInv, CoordInv2
          <8>2. CASE assets[AofS(self)] # "OwS"
          BY <8>2, <3>3, <2>0, <1>2, <2>4, SetsTheorem
          DEF TypeOk, Bother, decisionInv, CoordInv2
          <8>3. QED
          BY <8>1, <8>2
        <3>4. CASE BaskRM(self)
          BY <3>4, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,
          BaskRM, decisionInv, CoordInv2

        <3>5. CASE BlockAsset(self)
          <8>1. CASE (ProofPublish = TRUE /\
                      assets[AofS(self)] = "OwS")
            <9>1. assets' =
                  [assets EXCEPT ![AofS(self)] = "locked"]
            BY <8>1, <3>5, <2>0, <1>2, <2>4, SetsTheorem
            DEF TypeOk, BlockAsset, decisionInv, CoordInv2
            <9>2. QED
            BY <9>1, <8>1, <3>5, <2>0, <1>2, <2>4, SetsTheorem
            DEF TypeOk, BlockAsset, decisionInv, CoordInv2
          <8>2. CASE ~(ProofPublish = TRUE /\
                       assets[AofS(self)] = "OwS")
          BY <8>2, <3>5, <2>0, <1>2, <2>4, SetsTheorem
          DEF TypeOk, BlockAsset, decisionInv, CoordInv2
          <8>3. QED
          BY <8>1, <8>2
        <3>6. CASE BSaskRF(self)
          BY <3>6, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,
          BSaskRF, decisionInv, CoordInv2
        <3>7. CASE BrecoveringAsset(self)
          BY <3>7, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,
          BrecoveringAsset, decisionInv, CoordInv2
        <3>8. QED
          BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
          DEF BSource

      <2>5. CASE \E self \in CRecipients: Recipient(self)
        <3> SUFFICES ASSUME NEW self \in CRecipients,
                            Recipient(self)
                     PROVE  CoordInv2'
          BY <2>5
        <3>1. CASE init_rcp(self)
          BY <3>1, <1>2, <2>0,<2>5, SetsTheorem DEF TypeOk,
          init_rcp, decisionInv, CoordInv2
        <3>2. CASE waitForD_rcp(self)
          BY <3>2, <1>2, <2>0,<2>5, SetsTheorem DEF TypeOk,
          waitForD_rcp, decisionInv, CoordInv2
        <3>5. QED
          BY <2>5, <3>1, <3>2 DEF Recipient

      <2>6. CASE \E self \in BRecipients: BRecipient(self)
        <3> SUFFICES ASSUME NEW self \in BRecipients,
                            BRecipient(self)
                     PROVE  CoordInv2'
          BY <2>6
        <3>1. CASE init_brcp(self)
          BY <3>1, <1>2, <2>0,<2>6, SetsTheorem DEF TypeOk,
          init_brcp, decisionInv, CoordInv2
        <3>2. CASE BRaskRF(self)
          BY <3>2, <1>2, <2>0,<2>6, SetsTheorem DEF TypeOk,
          BRaskRF, decisionInv, CoordInv2
        <3>3. CASE BRretrievingAsset(self)
          BY <3>3, <1>2, <2>0,<2>6, SetsTheorem DEF TypeOk,
          BRretrievingAsset, decisionInv, CoordInv2
        <3>4. CASE BRdirectToS(self)
          BY <3>4, <1>2, <2>0,<2>6, SetsTheorem DEF TypeOk,
          BRdirectToS, decisionInv, CoordInv2
        <3>5. CASE BRother(self)
          BY <3>5, <1>2, <2>0,<2>6, SetsTheorem DEF TypeOk,
          BRother, decisionInv, CoordInv2
        <3>6. QED
          BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

      <2>7. CASE Terminating
        BY <1>2, <2>0,<2>7, SetsTheorem DEF TypeOk,
        Terminating, CoordInv2
      <2>8. QED
        BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

    <1>4. CASE pc[CoordinatorID] = "decisionValid"
      <2>0. decisionValidInv
        BY <1>4 DEF CoordInv2
      <2>1. CASE Publisher
        BY <1>4, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
        init_p, decisionValidInv, CoordInv2
      <2>2. CASE Coordinator
        <3>1. CASE init_c
          BY <1>4, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
        <3>2. CASE decision
          BY <1>4, <3>2, <2>2, SetsTheorem DEF TypeOk, decision

        <3>3. CASE decisionValid
          <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid
          <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid
          <4>3. (pc[CoordinatorID] = "decisionValid" =>
           decisionValidInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid, decisionValidInv
          <4>4. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM" )
           => okRMInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid, decisionValidInv, okRMInv
          <4>5. (pc[CoordinatorID] = "decisionAbort" =>
           decisionAbortInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid, decisionValidInv
          <4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF" )
           => okRFInv)'
            BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
            decisionValid, decisionValidInv, okRMInv
          <4>7. QED
            BY <4>1, <4>2, <4>3, <4>4,<4>5,<4>6 DEF CoordInv2
        <3>4. CASE decisionAbort
          BY <1>4, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
        <3>7. QED
          BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

      <2>3. CASE \E self \in CSources: Source(self)
        <3> SUFFICES ASSUME NEW self \in CSources,
                            Source(self)
                     PROVE  CoordInv2'
          BY <2>3
        <3>1. CASE init_src(self)
          BY <3>1, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
          init_src, decisionValidInv, CoordInv2
        <3>2. CASE lock(self)
          <4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]
              BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
              lock, decisionValidInv
```

```
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionValidInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>2, <2>0, <1>4, <2>3, <4>0, SetsTheorem
          DEF TypeOk, lock, decisionValidInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionValidInv, decisionValidInv
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
         => okRMInv)'
          BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionValidInv
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionValidInv, decisionValidInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
        => okRFInv)'
          BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionValidInv
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
         published, decisionValidInv, CoordInv2
      <3>4. CASE waitForD(self)
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv, decisionValidInv
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
         => okRMInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv, decisionValidInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
         => okRFInv)'
          BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
           waitForD, decisionValidInv
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, decisionValidInv, CoordInv2
      <3>2. CASE BdirectToR(self)
         <8>1. CASE assets[AofS(self)] = "OwS"
           <9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]
           BY <8>1, <3>2, <2>0, <1>4, <2>4, SetsTheorem
           DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2
           <9>2. QED
           BY <9>1, <8>1, <3>2, <2>0, <1>4, <2>4, SetsTheorem
           DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2
         <8>2. CASE assets[AofS(self)] # "OwS"
         BY <8>2, <3>2, <2>0, <1>4, <2>4, SetsTheorem
         DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2
         <8>3. QED
         BY <8>1, <8>2
      <3>3. CASE Bother(self)
        <8>1. CASE assets[AofS(self)] = "OwS"
          <9>1. assets' = [assets EXCEPT ![AofS(self)] = "other"]
          BY <8>1, <3>3, <2>0, <1>4, <2>4, SetsTheorem
          DEF TypeOk, Bother, decisionValidInv, CoordInv2
          <9>2. QED
          BY <9>1, <8>1, <3>3, <2>0, <1>4, <2>4, SetsTheorem
          DEF TypeOk, Bother, decisionValidInv, CoordInv2
        <8>2. CASE assets[AofS(self)] # "OwS"
        BY <8>2, <3>3, <2>0, <1>4, <2>4, SetsTheorem
        DEF TypeOk, Bother, decisionValidInv, CoordInv2
        <8>3. QED
        BY <8>1, <8>2
      <3>4. CASE BaskRM(self)
        <4>1. qrm = qrm \union { self }
        BY <3>4, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
         decisionValidInv
        <4>2. qrm' = qrm
        BY <4>1 , <3>4 DEF BaskRM
        <4>3. QED
        BY <4>2, <3>4, <2>0, <1>4, <2>4, SetsTheorem
        DEF TypeOk, BaskRM, decisionValidInv, CoordInv2

      <3>5. CASE BlockAsset(self)
        <8>1. CASE (ProofPublish = TRUE /\
                    assets[AofS(self)] = "OwS")
          <9>1. assets' =
                [assets EXCEPT ![AofS(self)] = "locked"]
          BY <8>1, <3>5, <2>0, <1>4, <2>4, SetsTheorem
          DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
          <9>2. QED
          BY <9>1, <8>1, <3>5, <2>0, <1>4, <2>4, SetsTheorem
          DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
        <8>2. CASE ~(ProofPublish = TRUE /\
                     assets[AofS(self)] = "OwS")
        BY <8>2, <3>5, <2>0, <1>4, <2>4, SetsTheorem
        DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
        <8>3. QED
        BY <8>1, <8>2
      <3>6. CASE BSaskRF(self)
        BY <3>6, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
        BSaskRF, decisionValidInv, CoordInv2
      <3>7. CASE BrecoveringAsset(self)
        BY <3>7, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
        BrecoveringAsset, decisionValidInv, CoordInv2
      <3>8. QED
        BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
        DEF BSource

    <2>5. CASE \E self \in CRecipients: Recipient(self)
      <3> SUFFICES ASSUME NEW self \in CRecipients,
                          Recipient(self)
                   PROVE  CoordInv2'
        BY <2>5
      <3>1. CASE init_rcp(self)
        BY <3>1, <1>4, <2>0,<2>5, SetsTheorem DEF TypeOk,
        init_rcp, decisionValidInv, CoordInv2
      <3>2. CASE waitForD_rcp(self)
        BY <3>2, <1>4, <2>0,<2>5, SetsTheorem DEF TypeOk,
         waitForD_rcp, decisionValidInv, CoordInv2
      <3>5. QED
```

```
        BY <2>5, <3>1, <3>2 DEF Recipient

    <2>6. CASE \E self \in BRecipients: BRecipient(self)
      <3> SUFFICES ASSUME NEW self \in BRecipients,
                          BRecipient(self)
                   PROVE  CoordInv2'
        BY <2>6
      <3>1. CASE init_brcp(self)
        BY <3>1, <1>4, <2>0,<2>6, SetsTheorem DEF TypeOk,
        init_brcp, decisionValidInv, CoordInv2
      <3>2. CASE BRaskRF(self)
        BY <3>2, <1>4, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRaskRF, decisionValidInv, CoordInv2
      <3>3. CASE BRretrievingAsset(self)
        BY <3>3, <1>4, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRretrievingAsset, decisionValidInv, CoordInv2
      <3>4. CASE BRdirectToS(self)
        BY <3>4, <1>4, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRdirectToS, decisionValidInv, CoordInv2
      <3>5. CASE BRother(self)
        BY <3>5, <1>4, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRother, decisionValidInv, CoordInv2
      <3>6. QED
        BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

    <2>7. CASE Terminating
      BY <1>4, <2>0,<2>7, SetsTheorem DEF TypeOk,
      Terminating, CoordInv2
    <2>8. QED
      BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

  <1>6. CASE pc[CoordinatorID] = "decisionAbort"
    <2>0. decisionAbortInv
      BY <1>6 DEF CoordInv2
    <2>1. CASE Publisher
      BY <1>6, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
       init_p, decisionAbortInv, CoordInv2
    <2>2. CASE Coordinator
      <3>1. CASE init_c
        BY <1>6, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
      <3>2. CASE decision
        BY <1>6, <3>2, <2>2, SetsTheorem DEF TypeOk, decision

      <3>3. CASE decisionAbort
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort, decisionAbortInv
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM")
          => okRMInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort, decisionAbortInv, okRMInv
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort, decisionAbortInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF")
         => okRFInv)'
          BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,
           decisionAbort, okRFInv, decisionAbortInv

        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4,<4>5,<4>6 DEF CoordInv2

      <3>4. CASE decisionValid
        BY <1>6, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionValid
      <3>7. QED
        BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

    <2>3. CASE \E self \in CSources: Source(self)
      <3> SUFFICES ASSUME NEW self \in CSources,
                          Source(self)
                   PROVE  CoordInv2'
        BY <2>3
      <3>1. CASE init_src(self)
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>4. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM" )
          => okRMInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF" )
          => okRFInv)'
          BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           init_src, decisionAbortInv, CoordInv2
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>2. CASE lock(self)
        <4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]
            BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
             lock, decisionAbortInv
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionAbortInv
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>2, <2>0, <1>6, <2>3, <4>0, SetsTheorem
          DEF TypeOk, lock, decisionAbortInv
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionAbortInv, decisionAbortInv
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
         => okRMInv)'
          BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
          lock, decisionAbortInv
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionAbortInv, decisionAbortInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
         => okRFInv)'
          BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           lock, decisionAbortInv
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
         published, decisionAbortInv, CoordInv2
      <3>4. CASE waitForD(self)
```

```
        <4>0. pc'[CoordinatorID] = pc[CoordinatorID] /\
         pc'[PublisherID] = pc[PublisherID]
          BY <3>4, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
           waitForD
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <4>0, <1>6
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <4>0, <1>6
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <4>0, <1>6
        <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
         => okRMInv)'
          BY <4>0, <1>6
        <4>5. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <4>0, <3>4, <2>0, <1>6, <2>3, SetsTheorem
          DEF TypeOk, waitForD, decisionAbortInv
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
         => okRFInv)'
          BY <4>0, <1>6
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, decisionAbortInv, CoordInv2
      <3>2. CASE BdirectToR(self)
         <8>1. CASE assets[AofS(self)] = "OwS"
           <9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]
           BY <8>1, <3>2, <2>0, <1>6, <2>4, SetsTheorem
           DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2
           <9>2. QED
           BY <9>1, <8>1, <3>2, <2>0, <1>6, <2>4, SetsTheorem
            DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2
         <8>2. CASE assets[AofS(self)] # "OwS"
         BY <8>2, <3>2, <2>0, <1>6, <2>4, SetsTheorem
         DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2
         <8>3. QED
         BY <8>1, <8>2
      <3>3. CASE Bother(self)
         <8>1. CASE assets[AofS(self)] = "OwS"
           <9>1. assets' = [assets EXCEPT ![AofS(self)] = "other"]
           BY <8>1, <3>3, <2>0, <1>6, <2>4, SetsTheorem
           DEF TypeOk, Bother, decisionAbortInv, CoordInv2
           <9>2. QED
           BY <9>1, <8>1, <3>3, <2>0, <1>6, <2>4, SetsTheorem
            DEF TypeOk, Bother, decisionAbortInv, CoordInv2
         <8>2. CASE assets[AofS(self)] # "OwS"
         BY <8>2, <3>3, <2>0, <1>6, <2>4, SetsTheorem
         DEF TypeOk, Bother, decisionAbortInv, CoordInv2
         <8>3. QED
         BY <8>1, <8>2
      <3>4. CASE BaskRM(self)
        BY <3>4, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
        BaskRM, decisionAbortInv, CoordInv2

      <3>5. CASE BlockAsset(self)
         <8>1. CASE (ProofPublish = TRUE /\
                     assets[AofS(self)] = "OwS")
           <9>1. assets' =
                [assets EXCEPT ![AofS(self)] = "locked"]
           BY <8>1, <3>5, <2>0, <1>6, <2>4, SetsTheorem
           DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
           <9>2. QED
           BY <9>1, <8>1, <3>5, <2>0, <1>6, <2>4, SetsTheorem
           DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
         <8>2. CASE ~(ProofPublish = TRUE /\
                      assets[AofS(self)] = "OwS")
        BY <8>2, <3>5, <2>0, <1>6, <2>4, SetsTheorem
        DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
        <8>3. QED
        BY <8>1, <8>2
      <3>6. CASE BSaskRF(self)
        BY <3>6, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
        BSaskRF, decisionAbortInv, CoordInv2
      <3>7. CASE BrecoveringAsset(self)
        BY <3>7, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
         BrecoveringAsset, decisionAbortInv, CoordInv2
      <3>8. QED
        BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
        DEF BSource

    <2>5. CASE \E self \in CRecipients: Recipient(self)
      <3> SUFFICES ASSUME NEW self \in CRecipients,
                          Recipient(self)
                   PROVE  CoordInv2'
        BY <2>5
      <3>1. CASE init_rcp(self)
        BY <3>1, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
         init_rcp, decisionAbortInv, CoordInv2
      <3>2. CASE waitForD_rcp(self)
        <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>3. (pc[CoordinatorID] = "decisionValid" =>
         decisionValidInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>4. (pc[CoordinatorID] = "decisionAbort" =>
         decisionAbortInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM" )
         => okRMInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF" )
         => okRFInv)'
          BY <3>2, <1>6, <2>0,<2>5, SetsTheorem DEF TypeOk,
           waitForD_rcp, decisionAbortInv, CoordInv2
        <4>7. QED
          BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2

      <3>5. QED
        BY <2>5, <3>1, <3>2 DEF Recipient

    <2>6. CASE \E self \in BRecipients: BRecipient(self)
      <3> SUFFICES ASSUME NEW self \in BRecipients,
                          BRecipient(self)
                   PROVE  CoordInv2'
        BY <2>6
      <3>1. CASE init_brcp(self)
        BY <3>1, <1>6, <2>0,<2>6, SetsTheorem DEF TypeOk,
         init_brcp, decisionAbortInv, CoordInv2
      <3>2. CASE BRaskRF(self)
        BY <3>2, <1>6, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRaskRF, decisionAbortInv, CoordInv2
```

```
      <3>3. CASE BRretrievingAsset(self)
        BY <3>3, <1>6, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRretrievingAsset, decisionAbortInv, CoordInv2
      <3>4. CASE BRdirectToS(self)
        BY <3>4, <1>6, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRdirectToS, decisionAbortInv, CoordInv2
      <3>5. CASE BRother(self)
        BY <3>5, <1>6, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRother, decisionAbortInv, CoordInv2
      <3>6. QED
        BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

    <2>7. CASE Terminating
      BY <1>6, <2>0,<2>7, SetsTheorem DEF TypeOk,
      Terminating, CoordInv2
    <2>8. QED
      BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

  <1>3. CASE pc[CoordinatorID] = "Done" /\ coordState = "okRM"
    <2>0. okRMInv
      BY <1>3 DEF CoordInv2
    <2>1. CASE Publisher
      BY <1>3, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
       init_p, okRMInv, CoordInv2
    <2>2. CASE Coordinator
      <3>1. CASE init_c
        BY <1>3, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
      <3>2. CASE decision
        BY <1>3, <3>2, <2>2, SetsTheorem DEF TypeOk, decision,
         CoordInv2
      <3>3. CASE decisionValid
        BY <1>3, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid
      <3>4. CASE decisionAbort
        BY <1>3, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
      <3>7. QED
        BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

    <2>3. CASE \E self \in CSources: Source(self)
      <3> SUFFICES ASSUME NEW self \in CSources,
                          Source(self)
                   PROVE  CoordInv2'
        BY <2>3
      <3>1. CASE init_src(self)
        BY <3>1, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,
         init_src, okRMInv, CoordInv2
      <3>2. CASE lock(self)
        BY <3>2, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,
         lock, okRMInv, CoordInv2

      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,
         published, okRMInv, CoordInv2
      <3>4. CASE waitForD(self)
        BY <3>4, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,
         waitForD, okRMInv, CoordInv2

      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, okRMInv, CoordInv2
      <3>2. CASE BdirectToR(self)
        BY <3>2, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         BdirectToR, okRMInv, CoordInv2
      <3>3. CASE Bother(self)
        BY <3>3, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
        Bother, okRMInv, CoordInv2
      <3>4. CASE BaskRM(self)
        BY <3>4, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         BaskRM, okRMInv, CoordInv2

      <3>5. CASE BlockAsset(self)
        BY <3>5, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         BlockAsset, okRMInv, CoordInv2
      <3>6. CASE BSaskRF(self)
        BY <3>6, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         BSaskRF, okRMInv, CoordInv2
      <3>7. CASE BrecoveringAsset(self)
        BY <3>7, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
         BrecoveringAsset, okRMInv, CoordInv2
      <3>8. QED
        BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
        DEF BSource

    <2>5. CASE \E self \in CRecipients: Recipient(self)
      <3> SUFFICES ASSUME NEW self \in CRecipients,
                          Recipient(self)
                   PROVE  CoordInv2'
        BY <2>5
      <3>1. CASE init_rcp(self)
        BY <3>1, <1>3, <2>0,<2>5, SetsTheorem DEF TypeOk,
        init_rcp, okRMInv, CoordInv2
      <3>2. CASE waitForD_rcp(self)
        BY <3>2, <1>3, <2>0,<2>5, SetsTheorem DEF TypeOk,
         waitForD_rcp, okRMInv, CoordInv2
      <3>5. QED
        BY <2>5, <3>1, <3>2 DEF Recipient

    <2>6. CASE \E self \in BRecipients: BRecipient(self)
      <3> SUFFICES ASSUME NEW self \in BRecipients,
                          BRecipient(self)
                   PROVE  CoordInv2'
        BY <2>6
      <3>1. CASE init_brcp(self)
        BY <3>1, <1>3, <2>0,<2>6, SetsTheorem DEF TypeOk,
         init_brcp, okRMInv, CoordInv2
      <3>2. CASE BRaskRF(self)
        BY <3>2, <1>3, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRaskRF, okRMInv, CoordInv2
      <3>3. CASE BRretrievingAsset(self)
        BY <3>3, <1>3, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRretrievingAsset, okRMInv, CoordInv2
      <3>4. CASE BRdirectToS(self)
        BY <3>4, <1>3, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRdirectToS, okRMInv, CoordInv2
      <3>5. CASE BRother(self)
        BY <3>5,<1>3, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRother, okRMInv, CoordInv2
      <3>6. QED
        BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

    <2>7. CASE Terminating
      BY <1>3, <2>0,<2>7, SetsTheorem DEF TypeOk,
      Terminating, vars, CoordInv2, init_cInv, decisionInv,
           okRMInv, decisionValidInv
    <2>8. QED
      BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

  <1>5. CASE (pc[CoordinatorID] = "Done" /\ coordState = "okRF")
    <2>0. okRFInv
      BY <1>5 DEF CoordInv2
    <2>1. CASE Publisher
```

```
      BY <1>5, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
       init_p, okRFInv, CoordInv2
    <2>2. CASE Coordinator
      <3>1. CASE init_c
        BY <1>5, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
      <3>2. CASE decision
        BY <1>5, <3>2, <2>2, SetsTheorem DEF TypeOk, decision,
         CoordInv2
      <3>3. CASE decisionValid
        BY <1>5, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid
      <3>4. CASE decisionAbort
        BY <1>5, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
      <3>7. QED
        BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator

    <2>3. CASE \E self \in CSources: Source(self)
      <3> SUFFICES ASSUME NEW self \in CSources,
                          Source(self)
                   PROVE  CoordInv2'
        BY <2>3
      <3>1. CASE init_src(self)
        BY <3>1, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,
         init_src, okRFInv, CoordInv2
      <3>2. CASE lock(self)
        BY <3>2, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,
         lock, okRFInv, CoordInv2
      <3>3. CASE published(self)
        BY <3>3, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,
         published, okRFInv, CoordInv2
      <3>4. CASE waitForD(self)
        BY <3>4, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,
         waitForD, okRFInv, CoordInv2

      <3>7. QED
        BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source

    <2>4. CASE \E self \in BSources: BSource(self)
      <3> SUFFICES ASSUME NEW self \in BSources,
                          BSource(self)
                   PROVE  CoordInv2'
        BY <2>4
      <3>1. CASE init_bsrc(self)
        BY <3>1, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         init_bsrc, okRFInv, CoordInv2
      <3>2. CASE BdirectToR(self)
        BY <3>2, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         BdirectToR, okRFInv, CoordInv2
      <3>3. CASE Bother(self)
        BY <3>3, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         Bother, okRFInv, CoordInv2
      <3>4. CASE BaskRM(self)
        BY <3>4, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
        BaskRM, okRFInv, CoordInv2

      <3>5. CASE BlockAsset(self)
        BY <3>5, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         BlockAsset, okRFInv, CoordInv2
      <3>6. CASE BSaskRF(self)
        BY <3>6, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         BSaskRF, okRFInv, CoordInv2
      <3>7. CASE BrecoveringAsset(self)
        BY <3>7, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,
         BrecoveringAsset, okRFInv, CoordInv2
      <3>8. QED
        BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
        DEF BSource

    <2>5. CASE \E self \in CRecipients: Recipient(self)
      <3> SUFFICES ASSUME NEW self \in CRecipients,
                          Recipient(self)
                   PROVE  CoordInv2'
        BY <2>5
      <3>1. CASE init_rcp(self)
        BY <3>1, <1>5, <2>0,<2>5, SetsTheorem DEF TypeOk,
         init_rcp, okRFInv, CoordInv2
      <3>2. CASE waitForD_rcp(self)
        BY <3>2, <1>5, <2>0,<2>5, SetsTheorem DEF TypeOk,
         waitForD_rcp, okRFInv, CoordInv2
      <3>5. QED
        BY <2>5, <3>1, <3>2 DEF Recipient

    <2>6. CASE \E self \in BRecipients: BRecipient(self)
      <3> SUFFICES ASSUME NEW self \in BRecipients,
                          BRecipient(self)
                   PROVE  CoordInv2'
        BY <2>6
      <3>1. CASE init_brcp(self)
        BY <3>1, <1>5, <2>0,<2>6, SetsTheorem DEF TypeOk,
         init_brcp, okRFInv, CoordInv2
      <3>2. CASE BRaskRF(self)
        BY <3>2, <1>5, <2>0,<2>6, SetsTheorem DEF TypeOk,
        BRaskRF, okRFInv, CoordInv2
      <3>3. CASE BRretrievingAsset(self)
        BY <3>3, <1>5, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRretrievingAsset, okRFInv, CoordInv2
      <3>4. CASE BRdirectToS(self)
        BY <3>4, <1>5, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRdirectToS, okRFInv, CoordInv2
      <3>5. CASE BRother(self)
        BY <3>5, <1>5, <2>0,<2>6, SetsTheorem DEF TypeOk,
         BRother, okRFInv, CoordInv2
      <3>6. QED
        BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient

    <2>7. CASE Terminating
      BY <1>5, <2>0,<2>7, SetsTheorem DEF TypeOk,
      Terminating, vars, CoordInv2, init_cInv, decisionInv,
           okRFInv, decisionValidInv
    <2>8. QED
      BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next

  <1>7. QED
    BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6 DEF TypeOk, CStates



THEOREM InvInvariant ==
    ASSUME Inv, Next
    PROVE Inv'
BY TypeOkInvariant, CoordInvariant DEF Inv, TypeOk, CoordInv2, Next


THEOREM InvImpliesConsistency ==
    ASSUME TypeOk /\ CoordInv2
    PROVE Consistency
  <1> USE DEF Finish, AvailableS, AvailableR
  <1>1. CASE pc[CoordinatorID] = "init_c"
  BY <1>1, SetsTheorem DEF CoordInv2, init_cInv, Consistency
  <1>2. CASE pc[CoordinatorID] = "decision"
    <2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "OwS"}
    BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
    <2>2. \A s \in CSources: assets[AofS(s)] \in  {"locked", "OwS"}
    BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
    <2>3. \A s \in CSources: pc[s] = "Done" =>
                                    assets[AofS(s)] = "OwS"
    BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
    <2>4. QED
    BY <2>1, <2>2, <2>3 DEF Consistency
```

```
  <1>3. CASE pc[CoordinatorID] = "decisionValid"
    <2>1. \A a \in Assets: assets[a] =  "locked"
       BY <1>3, SetsTheorem DEF CoordInv2, decisionValidInv,
                                Consistency
    <2>2. \A a \in AssetsFromCS: assets[a] = "locked"
       BY <1>3, SetsTheorem DEF CoordInv2, decisionValidInv,
                                Consistency
    <2>3. \A s \in CSources: pc[s] # "Done"
       BY <1>3, SetsTheorem DEF  CoordInv2, decisionValidInv,
                                 Consistency
    <2>4. QED
    BY <2>1, <2>2, <2>3 DEF Consistency
  <1>4. CASE  pc[CoordinatorID] = "decisionAbort"
   <2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "OwS"}
    BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
                             Consistency
   <2>2. \A s \in CSources: assets[AofS(s)] \in
                            { "locked", "OwS"}
    BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
                             Consistency
   <2>3. \A s \in CSources: pc[s] = "Done" =>
                               assets[AofS(s)] = "OwS"
    BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
                             Consistency
   <2>4 QED
    BY <2>1, <2>2, <2>3 DEF Consistency
  <1>5. CASE (pc[CoordinatorID] ="Done" /\ coordState ="okRM")
    <2>1. ProofOkRM = TRUE  => \A a \in AssetsForCR: assets[a]
                                  \in {"locked", "OwR"}
    BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
    <2>2. \A r \in CRecipients : pc[r] = "Done" =>
                        assets[AofR(r)] \in {"OwR", "locked"}
    BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
    <2>3. ProofOkRM = TRUE
    BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
   <2>4 QED
    BY <2>1, <2>2, <2>3 DEF Consistency
  <1>6. CASE (pc[CoordinatorID] = "Done" /\ coordState ="okRF")
   <2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "OwS"}
    BY <1>6, SetsTheorem DEF CoordInv2, okRFInv, Consistency
   <2>2. \A s \in CSources: pc[s] = "Done"  =>
                                  assets[AofS(s)] = "OwS"
    BY <1>6, SetsTheorem DEF CoordInv2, okRFInv, Consistency
   <2>3 QED
    BY <2>1, <2>2 DEF Consistency
  <1>7. QED
  BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6 DEF TypeOk

THEOREM Safety2 == Spec => [] Consistency
<1>1. Init => Inv
  BY InitImpliesInv, SMT DEF Inv
<1>2. Inv /\ [Next]_vars => Inv'
  <2> SUFFICES ASSUME Inv,
                      [Next]_vars
               PROVE  Inv'
    OBVIOUS
  <2>1. CASE Next
    BY <2>1, SMTT(60), InvInvariant DEF vars
  <2>2. CASE UNCHANGED vars
    BY <2>2, SMTT(60) DEF vars, Inv, TypeOk, CoordInv2, okRFInv,
    okRMInv, decisionAbortInv, decisionValidInv, init_cInv,
     decisionInv
  <2>3. QED
    BY <2>1, <2>2

<1>3. Inv => Consistency
  BY SMT, InvImpliesConsistency DEFS Consistency, Inv
<1>4. QED
  BY ONLY <1>1,<1>2,<1>3,PTL DEF Spec

=============================================================
```