



HAL
open science

A TLA+ formal proof of a cross-chain swap

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

► **To cite this version:**

Zeinab Nehai, Francois Bobot, Sara Tucci, Carole Delporte-Gallet, Hugues Fauconnier. A TLA+ formal proof of a cross-chain swap. ICDCN '22: 23rd International Conference on Distributed Computing and Networking, ACM, Jan 2022, Delhi, India. pp.148-159, 10.1145/3491003.3491006 . cea-04539632

HAL Id: cea-04539632

<https://cea.hal.science/cea-04539632v1>

Submitted on 9 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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.

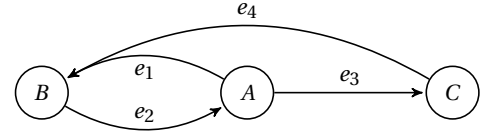


Figure 1: A swap graph S with $\Pi = \{A, B, C\}$ and $m = 4$ assets

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 $E = \{e_1, e_2, \dots, e_m\}$. 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 $i \in \{1, \dots, m\}$, $s \neq r$, 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 follows¹:

- *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.

¹The 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 \mathcal{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 figure 2(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 \mathcal{P}_{swap}

The abstract protocol, \mathcal{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 \mathcal{P}_{swap} . \mathcal{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 \mathcal{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. \mathcal{P}_{swap} interacts with participants of the swap where their behaviour is represented by a state machine structured with the following elements $(\Gamma, Q, \Sigma, \delta, q_0, F)$.

- 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 \emptyset 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 $\epsilon_1^P : \text{publish!}$ 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 \mathcal{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.

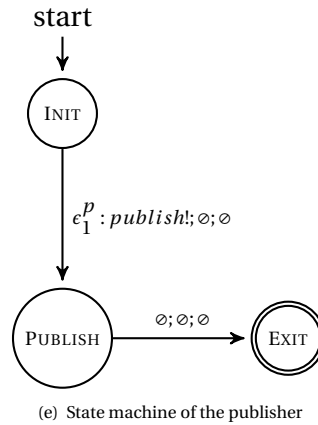
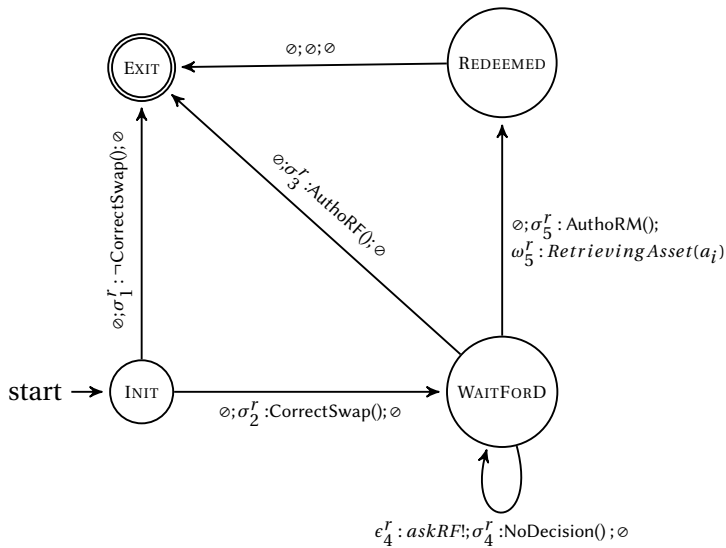
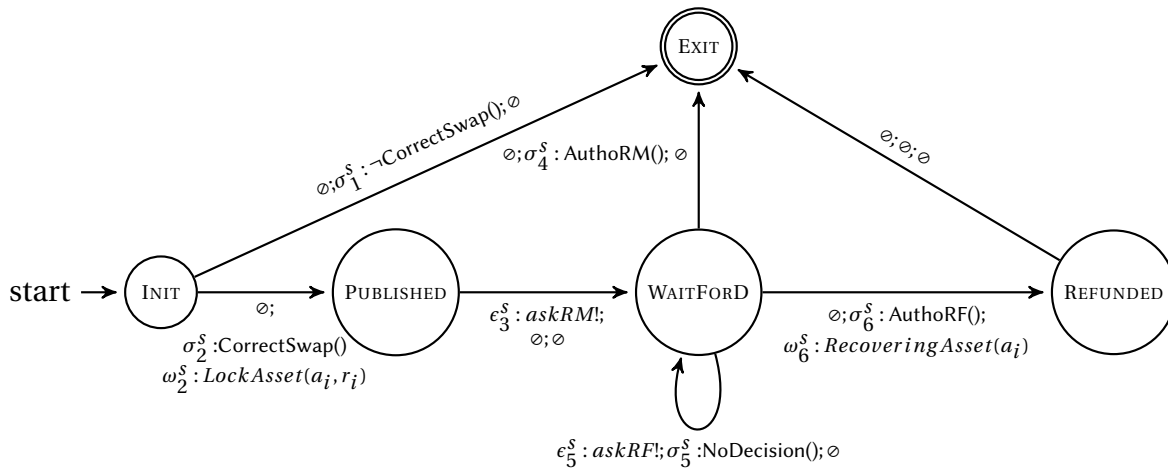
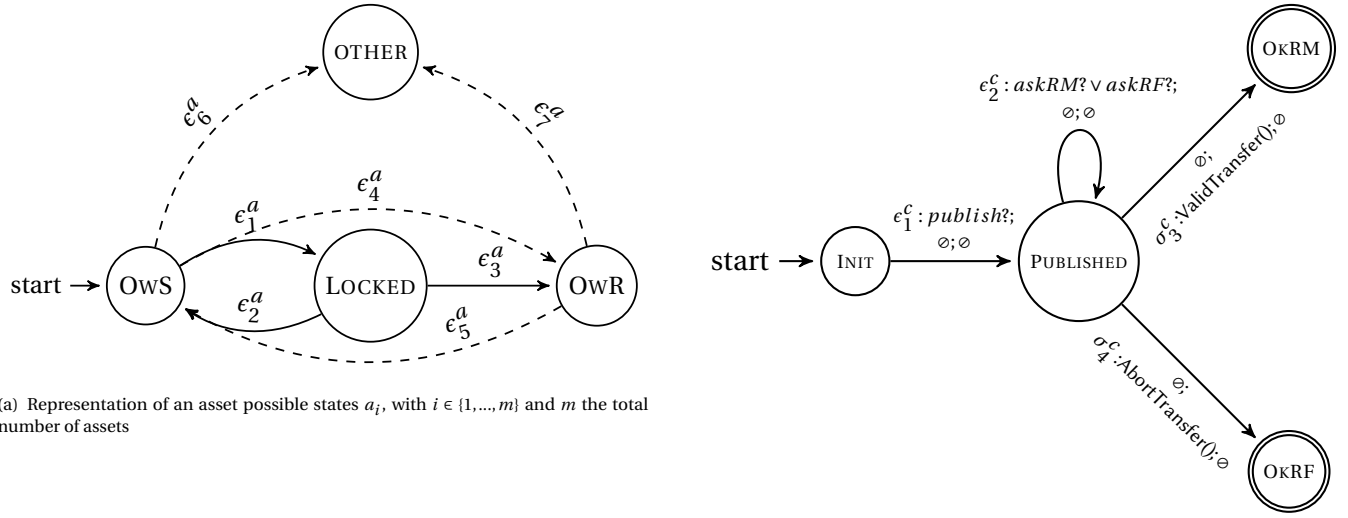


Figure 2: Asset representation and participants state machines

The protocol \mathcal{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 \mathcal{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 online².

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 $Spec = Init \wedge [] [Next]_{vars}$. 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)] = "0wS")
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>

```

2  if (self \in Sources /\ ProofLock[self] = TRUE /\
3    coordState = "published") qrm := qrm \union {self};}
4  macro askRF(self){
5    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:

```

1  macro retrievingAsset(self){
2    if (self \in Recipients /\ ProofOkRM = TRUE /\
3      assets[AofR(self)] = "locked") assets[AofR(self)] := "OwR";}
4  macro recoveringAsset(self){
5    if (self \in Sources /\ ProofOkRF = TRUE /\
6      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:

```

1  macro otherS(self){
2    if (self \in Sources /\ assets[AofS(self)] = "OwS")
3      assets[AofS(self)] := "other";}
4  macro otherR(self){
5    if (self \in Recipients /\ assets[AofR(self)] = "OwR")
6      assets[AofR(self)] := "other";}
7  macro directToR(self){
8    if (self \in Sources /\ assets[AofS(self)] = "OwS")
9      assets[AofS(self)] := "OwR";}
10 macro directToS(self){
11  if (self \in Recipients /\ assets[AofR(self)] = "OwR")
12  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}.

```

1  process BSource \in BSources{
2    init_bsrc:
3    either {BdirectToR: directToR(self); goto init_bsrc; }
4    or {Bother: otherS(self); goto init_bsrc; }
5    or {BaskRM: askRM(self); goto init_bsrc; }
6    or {BblockAsset: lockAsset(self); goto init_bsrc; }
7    or {BSaskRF: askRF(self); goto init_bsrc; }
8    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.

```

1  process BRecipient \in BRecipients{
2    init_brcp:
3    either {BaskRF: askRF(self); goto init_brcp;}
4    or {BRretrievingAsset: retrievingAsset(self); goto init_brcp;}
5    or {BRdirectToS: directToS(self); goto init_brcp;}
6    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 : $\frac{Init \Rightarrow Inv \quad Inv \wedge Next \Rightarrow Inv'}{Inv \Rightarrow Consistency}$

$Spec \Rightarrow \Box Consistency$

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.

```

1  CoordInv ==
2  /\ pc[CoordID] = "init_c" => init_cInv
3  /\ pc[CoordID] = "decision" => decisionInv
4  /\ pc[CoordID] = "decisionValid" => decisionValidInv
5  /\ pc[CoordID] = "decisionAbort" => decisionAbortInv
6  /\ (pc[CoordID] = "Done" /\ coordState = "okRM") => okRMInv
7  /\ (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+:

```

1  AvailableS(a) == assets[a] = "OwS" \/\
2    (ProofOkRF = TRUE /\ assets[a] = "locked")
3  AvailableR(a) == assets[a] = "OwR" \/\
4    (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:

```

1  Consistency == \A s \in CSources, r \in CRecipients:
2  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: Success *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:

- 1 AllParticipantsAreCorrect == (Pi = Pc) /\ swapGraph = "correct"
- 2 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 \leadsto B) is "syntactic sugar" for $[\](A \Rightarrow \langle \rangle B)$, with the temporal operators $[\]$; *always*, $\langle \rangle$; *eventually* and \Rightarrow ; *implies*.

- 1 OwnershipS == $\forall s \ \backslash \text{in AssetsFromCS} : \text{AvailableS}(s)$
- 2 OwnershipR == $\forall r \ \backslash \text{in AssetsForCR} : \text{AvailableR}(r)$
- 3 Ownership == AtLeastOneCorrect \leadsto (OwnershipS \vee OwnershipR)
- 4 Retrieving == AllParticipantsAreCorrect /\ Timeout = FALSE \rightarrow
- 5 $(\forall r \ \backslash \text{in Recipients: assets[AofR}(r)] = \text{"OwR"})$

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 $\mathcal{P}_{\text{swap}}$ for the *Ownership* property on a computer Intel® Core™ i7-8850H CPU @ 2.60GHz \times 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 $\mathcal{P}_{\text{inst}}$

$\mathcal{P}_{\text{inst}}$ protocol is an instantiation of the protocol $\mathcal{P}_{\text{swap}}$. In this section, we instantiate $\mathcal{P}_{\text{swap}}$ 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 $\mathcal{P}_{\text{swap}}$

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 $\mathcal{P}_{\text{swap}}$ 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 \mathcal{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 \mathcal{P}_{swap} , each source has to lock each asset it wants to trade. In \mathcal{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 Instantiation 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.

³the 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 summarised 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]	\mathcal{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 asserts 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 \mathcal{P}_{swap} and [25] specification are completely protocol-agnostic. Another point of comparison is the formal description of the protocol. Unlike \mathcal{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, Iliana 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: Success 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 \mathcal{A}_r be the set of states that implies "available to its recipient" with $\mathcal{A}_r = (loc(a) = "OwR" \vee (Proof_{redeem} \wedge loc(a) = "locked"))$ and \mathcal{A}_s the set of states that implies "available to its source" with $\mathcal{A}_s = (loc(a) = "OwS" \vee (Proof_{refund} \wedge loc(a) = "locked"))$. 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: $loc(c) = "INIT" \implies \forall s \in (P_c \cap \Pi_s) : loc(s) \notin "PUBLISHED" \wedge \forall a \in \Lambda_s : loc(a) = "OWS"$.

PROOF. From figure 2(b) we can see that in the initial state, the coordinator has not triggered e_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 e_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. \square

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

PROOF. When the coordinator is in "PUBLISHED" state, then e_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 e_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. \square

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

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

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 e_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 . \square

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

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

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 e_5^s if the participant is a source or triggering e_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 . \square

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. \square

LEMMA B.6. *If “Proof_{publish}” is valid and at least one participant is correct, then the coordinator eventually makes a decision.*

Formally: $\exists p \in P_c : \text{Proof}_{\text{publish}} \wedge P_c \neq \{\} \implies \diamond(\text{loc}(c) = \text{“OKRF”} \vee \text{loc}(c) = \text{“OKRM”})$

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 “WAITFOR” 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. \square

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

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

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. \square

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

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

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. \square

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. \square

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. \square

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

```

A TLA+ Formal Proof of a Cross-Chain Swap

```

29 \* Pc: the set of correct sources and recipients          98
30 \* CSources: the set of correct sources                  99
31 \* CRecipients: the set of correct recipients           100
32 \* BSources: the set of byzantine sources               101
33 \* BRecipients: the set of byzantine recipients         102
34                                                         103
35 Sources == {3*x-2 : x \in 1..NTxs}                    104
36 Assets  == {3*x-1 : x \in 1..NTxs}                    105
37 Recipients == {3*x : x \in 1..NTxs}                   106
38                                                         107
39 Pi == Sources \cup Recipients                          108
40 Pc == Pi \cap Correct                                  109
41                                                         110
42 CSources == Pc \cap Sources                            111
43 CRecipients == Pc \cap Recipients                     112
44 BSources == Sources \ CSources                        113
45 BRecipients == Recipients \ CRecipients               114
46                                                         115
47 \* AofS(x): function that gives the asset id of the source 'x' 116
48 \* AofR(x): function that gives the asset id of the recipient 'x' 117
49 \* SofA(x): function that gives the source id of the asset 'x' 118
50 \* RofA(x): function that gives the recipient id of the asset 'x' 119
51 \* AssetsFromCS: set of assets initially owned by a correct source 120
52 \* AssetsForCR: set of assets intended for a correct recipient 121
53                                                         122
54 AofS(x) == x + 1                                       123
55 AofR(x) == x - 1                                       124
56 SofA(x) == x - 1                                       125
57 RofA(x) == x + 1                                       126
58                                                         127
59 AssetsFromCS == {AofS(x) : x \in CSources}             128
60 AssetsForCR == {AofR(x) : x \in CRecipients}           129
61                                                         130
62 (*                                                         131
63                                                         132
64 —fair algorithm ACC {                                  133
65                                                         134
66 \* assets: list of all assets initialise to "OwS" state 135
67 \* pState: Publisher state initialises to "init"       136
68 \* coordState: coordinator state initialises to "init" 137
69 \* qrm: sequence of ask redeem call function           138
70 \* qrf: sequence of ask refund call function           139
71 \* swapGraph: swap graph state initialises to "init"  140
72 \* ProofPublish, ProofLock, ProofOkRM and ProofOkRF are 141
73 \* proof-of-action describes in the article           142
74 \* assets, pState, coordState, qrm, qrf, swapGraph and 143
75 \* proof-of-action are VARIABLES for tlaps            144
76                                                         145
77 variable                                              146
78     assets      = [a \in Assets |-> "OwS"],           147
79     pState      = "init",                             148
80     coordState  = "init",                             149
81     qrm = {},                                         150
82     qrf = FALSE,                                     151
83     swapGraph = "init",                              152
84     ProofPublish = FALSE,                             153
85     ProofLock = [c \in Sources |-> FALSE],             154
86     ProofOkRM = FALSE,                               155
87     ProofOkRF = FALSE;                               156
88                                                         157
89 \* the define block can be used in pluscal code and    158
90 \* in properties                                       159
91 \* The operator returns true if the expression is true 160
92 \* ValidTransfer and AbortTransfer corresponds to the  161
93 \* predicates of the same name in the article         162
94 \* qrm = Sources -> all sources has been asked for   163
95 \* a redeem authorisation                             164
96 \* qrf = true -> at least one participant has asked for 165
97 \* a refund authorisation                             166

```

```

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 a 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 a 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
237
238
239
240
241
242
243
244
245
246
247
248
249
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 determinism*)
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
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325

```

A TLA+ Formal Proof of a Cross-Chain Swap

```

304   either { BdirectToR: directToR(self); goto init_bsrc; } 373 ProcSet == {PublisherID} \cup {CoordinatorID} \cup (CSources
305   or { Bother: other(self); goto init_bsrc; } 374   \cup (BSources) \cup (RRecipients) \cup (BRecipients)
306   or {BaskRM: askRM(self); goto init_bsrc; } 375
307   or { BLockAsset: lockAsset(self); goto init_bsrc; } 376 Init == (* Global variables *)
308   or { BSaskRF: askRF(self); goto init_bsrc; } 377   /\ assets = [a \in Assets |-> "0wS"]
309   or {BrecoveringAsset: recoveringAsset(self); goto init_bsrc;}; 378   /\ pState = "init"
310   } 379   /\ coordState = "init"
311 }; 380   /\ qrm = {}
312 381   /\ qrf = FALSE
313 (* ***** 382   /\ swapGraph = "init"
314 (* Recipient: multiprocess of CRecipients processes (correct *) 383   /\ ProofPublish = FALSE
315 (* recipients *) 384   /\ ProofLock = [c \in Sources |-> FALSE]
316 (* init_rcp: either the swap is correct or different *) 385   /\ ProofOkRM = FALSE
317 (* waitForD_rcp: the recipient waits for the coordinator *) 386   /\ ProofOkRF = FALSE
318 (* decision. *) 387   /\ pc = [self \in ProcSet |->
319 (* ***** 388     CASE self = PublisherID -> "init_p"
320 389     [] self = CoordinatorID -> "init_c"
321 fair process (Recipient \in CRecipients) 390     [] self \in CSources -> "init_src"
322 { 391     [] self \in BSources -> "init_bsrc"
323 init_rcp : 392     [] self \in CRecipients -> "init_rcp"
324   either { await swapGraph = "different" \ / TRUE; 393     [] self \in BRecipients -> "init_brcp"}
325     goto Done;} 394
326   or { await ProofPublish = TRUE /\ swapGraph = "correct"; 395 init_p == /\ pc[PublisherID] = "init_p"
327 waitForD_rcp: 396   /\ \ / /\ pState' = "publish"
328   either { await ProofOkRF = TRUE; 397   /\ \ / /\ swapGraph' = "correct"
329     goto Done;} 398   \ / /\ swapGraph' = "different"
330   or { await ProofOkRM = TRUE; 399   \ / /\ TRUE
331     retrievingAsset(self); 400   /\ UNCHANGED <<pState, swapGraph>>
332     goto Done;} 401   /\ pc' = [pc EXCEPT ![PublisherID] = "Done"]
333   or { await coordState = "published"; 402   /\ UNCHANGED << assets, coordState, qrm, qrf,
334     askRF(self); 403     ProofPublish, ProofLock, ProofOkRM, ProofOkRF >>
335     goto waitForD_rcp;}; 404
336 }; 405 Publisher == init_p
337 }; 406
338 407
339 (* ***** 408 init_c == /\ pc[CoordinatorID] = "init_c"
340 (* BRecipient is a multiprocess of BRecipients processes *) 409   /\ pState = "publish" /\ swapGraph # "init"
341 (* (byzantine recipient). As byzantine sources, a byzantine *) 410   /\ coordState' = "published"
342 (* recipient behavior cannot be predicted. a byzantine can *) 411   /\ ProofPublish' = TRUE
343 (* execute the actions of a recipient in a completely random *) 412   /\ pc' = [pc EXCEPT ![CoordinatorID] = "decision"]
344 (* order in addition to the actions. *) 413   /\ UNCHANGED << assets, pState, qrm, qrf, swapGraph,
345 (* As a result, it has the ability to run the protocol *) 414     ProofLock, ProofOkRM, ProofOkRF >>
346 (* correctly and behaves as a correct recipient. The process *) 415
347 (* is unfair, thus we do not add the 'fair' statement before *) 416 decision == /\ pc[CoordinatorID] = "decision"
348 (* process. We assume that the process can crash at anytime *) 417   /\ \ / /\ ValidTransfer
349 (* ***** 418   /\ pc' = [pc EXCEPT ![CoordinatorID] =
350 419     "decisionValid"]
351 process (BRecipient \in BRecipients) 420   \ / /\ AbortTransfer
352 { 421   /\ pc' = [pc EXCEPT ![CoordinatorID] =
353   init_brcp: 422     "decisionAbort"]
354   either {BRaskRF: askRF(self); goto init_brcp;} 423   /\ UNCHANGED << assets, pState, coordState,
355   or {BRretrievingAsset: retrievingAsset(self); goto init_brcp;} 424     qrm, qrf, swapGraph, ProofPublish, ProofLock,
356   or {BRdirectToS: directToS(self); goto init_brcp;} 425     ProofOkRM, ProofOkRF >>
357   or {BROther: otherR(self); goto init_brcp;}; 426
358   }; 427 decisionValid == /\ pc[CoordinatorID] = "decisionValid"
359   }; 428   /\ coordState' = "okRM"
360   *) 429   /\ ProofOkRM' = TRUE
361 \* BEGIN TRANSLATION 430   /\ pc' = [pc EXCEPT ![CoordinatorID] = "Done"]
362 VARIABLES assets, pState, coordState, qrm, qrf, swapGraph, 431   /\ UNCHANGED << assets, pState, qrm, qrf,
363   ProofPublish, ProofLock, ProofOkRM, ProofOkRF, pc 432     swapGraph, ProofPublish, ProofLock, ProofOkRF >>
364 433
365 (* define statement *) 434 decisionAbort == /\ pc[CoordinatorID] = "decisionAbort"
366 ValidTransfer == qrm = Sources 435   /\ coordState' = "okRF"
367 AbortTransfer == qrf = TRUE 436   /\ ProofOkRF' = TRUE
368 437   /\ pc' = [pc EXCEPT ![CoordinatorID] = "Done"]
369 438   /\ UNCHANGED << assets, pState, qrm, qrf,
370 vars == << assets, pState, coordState, qrm, qrf, swapGraph, 439     swapGraph, ProofPublish, ProofLock, ProofOkRM >>
371 ProofPublish, ProofLock, ProofOkRM, ProofOkRF, pc >> 440 Coordinator ==
372 441   init_c \ / decision \ / decisionValid \ / decisionAbort

```



```

442
443 init_src(self) == /\ pc[self] = "init_src"
444     /\ /\ /\ swapGraph = "different" \/ TRUE
445     /\ pc' = [pc EXCEPT ![self] = "Done"]
446     \/ /\ ProofPublish = TRUE
447     /\ swapGraph = "correct"
448     /\ pc' = [pc EXCEPT ![self] = "lock"]
449     /\ UNCHANGED << assets, pState, coordState,
450     qrm, qrf, swapGraph, ProofPublish, ProofLock,
451     ProofOkRM, ProofOkRF >>
452
453 lock(self) == /\ pc[self] = "lock"
454     /\ IF ProofPublish = TRUE /\ self \in Sources /\
455     assets[AofS(self)] = "OwS"
456     THEN /\ assets' =
457     [assets EXCEPT ![AofS(self)] = "locked"]
458     /\ ProofLock' =
459     [ProofLock EXCEPT ![self] = TRUE]
460     ELSE /\ TRUE
461     /\ UNCHANGED << assets, ProofLock >>
462     /\ pc' = [pc EXCEPT ![self] = "published"]
463     /\ UNCHANGED << pState, coordState, qrm, qrf,
464     swapGraph, ProofPublish, ProofOkRM, ProofOkRF >>
465
466 published(self) == /\ pc[self] = "published"
467     /\ IF self \in Sources
468     /\ ProofLock[self] = TRUE
469     /\ coordState = "published"
470     THEN /\ qrm' = (qrm \union {self})
471     ELSE /\ TRUE
472     /\ qrm' = qrm
473     /\ pc' = [pc EXCEPT ![self] = "waitFord"]
474     /\ UNCHANGED << assets, pState, coordState,
475     qrf, swapGraph, ProofPublish, ProofLock,
476     ProofOkRM, ProofOkRF >>
477
478 waitFord(self) == /\ pc[self] = "waitFord"
479     /\ /\ /\ ProofOkRM = TRUE
480     /\ pc' = [pc EXCEPT ![self] = "Done"]
481     /\ UNCHANGED <<assets, qrf>>
482     \/ /\ ProofOkRF = TRUE
483     /\ IF self \in Sources
484     /\ ProofOkRF = TRUE
485     /\ assets[AofS(self)] = "locked"
486     THEN /\ assets' =
487     [assets EXCEPT ![AofS(self)] = "OwS"]
488     ELSE /\ TRUE
489     /\ UNCHANGED assets
490     /\ pc' = [pc EXCEPT ![self] = "Done"]
491     /\ qrf' = qrf
492     \/ /\ coordState = "published"
493     /\ IF coordState = "published"
494     THEN /\ qrf' = TRUE
495     ELSE /\ TRUE
496     /\ qrf' = qrf
497     /\ pc' = [pc EXCEPT ![self] = "waitFord"]
498     /\ UNCHANGED assets
499     /\ UNCHANGED << pState, coordState, qrm,
500     swapGraph, ProofPublish, ProofLock, ProofOkRM,
501     ProofOkRF >>
502
503 Source(self) == init_src(self) \/ lock(self) \/ published(self)
504     \/ waitFord(self)
505
506 init_bsrc(self) ==
507     /\ pc[self] = "init_bsrc"
508     /\ \/ /\ pc' = [pc EXCEPT ![self] = "BdirectToR"]
509     \/ /\ pc' = [pc EXCEPT ![self] = "Bother"]
510     \/ /\ pc' = [pc EXCEPT ![self] = "BaskRM"]
511     \/ /\ pc' = [pc EXCEPT ![self] = "BlockAsset"]
512     \/ /\ pc' = [pc EXCEPT ![self] = "BSaskRF"]
513     \/ /\ pc' = [pc EXCEPT ![self] = "BrecoveringAsset"]
514     /\ UNCHANGED << assets, pState, coordState, qrm,
515     qrf, swapGraph, ProofPublish, ProofLock,
516     ProofOkRM, ProofOkRF >>
517
518 BdirectToR(self) == /\ pc[self] = "BdirectToR"
519     /\ IF self \in Sources
520     /\ assets[AofS(self)] = "OwS"
521     THEN /\ assets' =
522     [assets EXCEPT ![AofS(self)] = "OwR"]
523     ELSE /\ TRUE
524     /\ UNCHANGED assets
525     /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
526     /\ UNCHANGED << pState, coordState, qrm,
527     qrf, swapGraph, ProofPublish, ProofLock,
528     ProofOkRM, ProofOkRF >>
529
530 Bother(self) == /\ pc[self] = "Bother"
531     /\ IF self \in Sources
532     /\ assets[AofS(self)] = "OwS"
533     THEN /\ assets' =
534     [assets EXCEPT ![AofS(self)] = "other"]
535     ELSE /\ TRUE
536     /\ UNCHANGED assets
537     /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
538     /\ UNCHANGED << pState, coordState, qrm, qrf,
539     swapGraph, ProofPublish, ProofLock, ProofOkRM,
540     ProofOkRF >>
541
542 BaskRM(self) == /\ pc[self] = "BaskRM"
543     /\ IF self \in Sources
544     /\ ProofLock[self] = TRUE
545     /\ coordState = "published"
546     THEN /\ qrm' = (qrm \union {self})
547     ELSE /\ TRUE
548     /\ qrm' = qrm
549     /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
550     /\ UNCHANGED << assets, pState, coordState,
551     qrf, swapGraph, ProofPublish, ProofLock,
552     ProofOkRM, ProofOkRF >>
553
554 BlockAsset(self) == /\ pc[self] = "BlockAsset"
555     /\ IF ProofPublish = TRUE
556     /\ self \in Sources
557     /\ assets[AofS(self)] = "OwS"
558     THEN /\ assets' =
559     [assets EXCEPT ![AofS(self)] = "locked"]
560     /\ ProofLock' =
561     [ProofLock EXCEPT ![self] = TRUE]
562     ELSE /\ TRUE
563     /\ UNCHANGED << assets, ProofLock >>
564     /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
565     /\ UNCHANGED << pState, coordState, qrm,
566     qrf, swapGraph, ProofPublish, ProofOkRM,
567     ProofOkRF >>
568
569 BSaskRF(self) == /\ pc[self] = "BSaskRF"
570     /\ IF coordState = "published"
571     THEN /\ qrf' = TRUE
572     ELSE /\ TRUE
573     /\ qrf' = qrf
574     /\ pc' = [pc EXCEPT ![self] = "init_bsrc"]
575     /\ UNCHANGED << assets, pState, coordState,
576     qrm, swapGraph, ProofPublish, ProofLock,
577     ProofOkRM, ProofOkRF >>
578
579 BrecoveringAsset(self) == /\ pc[self] = "BrecoveringAsset"

```

```

580      /\ IF self \in Sources                                649
581      /\ ProofOkRF = TRUE                                  650
582      /\ assets[AofS(self)] = "locked"                    651
583      THEN /\ assets' =                                    652
584      [assets EXCEPT ![AofS(self)] = "OwS"]            653
585      ELSE /\ TRUE                                        654
586      /\ UNCHANGED assets                                655
587      /\ pc' =                                           656
588      [pc EXCEPT ![self] = "init_bsrc"]                657
589      /\ UNCHANGED << pState, coordState,                658
590      qrm, qrf, swapGraph, ProofPublish,                659
591      ProofLock, ProofOkRM, ProofOkRF >>                660
592
593 BSource(self) == init_bsrc(self) \/ BdirectToR(self)    662
594   \/ Bother(self) \/ BaskRM(self) \/ BlockAsset(self) \/ 663
595   BSaskRF(self) \/ BrecoveringAsset(self)              664
596
597 init_rcp(self) == /\ pc[self] = "init_rcp"              666
598   /\ \/ /\ swapGraph = "different" \/ TRUE             667
599   /\ pc' = [pc EXCEPT ![self] = "Done"]              668
600   \/ /\ ProofPublish = TRUE                             669
601   /\ swapGraph = "correct"                              670
602   /\ pc' =                                              671
603   [pc EXCEPT ![self] = "waitForD_rcp"]               672
604   /\ UNCHANGED << assets, pState, coordState,          673
605   qrm, qrf, swapGraph, ProofPublish, ProofLock,        674
606   ProofOkRM, ProofOkRF >>                              675
607
608 waitForD_rcp(self) == /\ pc[self] = "waitForD_rcp"      677
609   /\ \/ /\ ProofOkRF = TRUE                             678
610   /\ pc' = [pc EXCEPT ![self] = "Done"]              679
611   /\ UNCHANGED <<assets, qrf>>                          680
612   \/ /\ ProofOkRM = TRUE                                681
613   /\ IF self \in Recipients                             682
614   /\ ProofOkRM = TRUE                                   683
615   /\ assets[AofR(self)] = "locked"                     684
616   THEN /\ assets' =                                     685
617   [assets EXCEPT ![AofR(self)] = "OwR"]              686
618   ELSE /\ TRUE                                          687
619   /\ UNCHANGED assets                                  688
620   /\ pc' = [pc EXCEPT ![self] = "Done"]              689
621   /\ qrf' = qrf                                         690
622   \/ /\ coordState = "published"                       691
623   /\ IF coordState = "published"                       692
624   THEN /\ qrf' = TRUE                                   693
625   ELSE /\ TRUE                                          694
626   /\ qrf' = qrf                                         695
627   /\ pc' =                                              696
628   [pc EXCEPT ![self] = "waitForD_rcp"]              697
629   /\ UNCHANGED assets                                  698
630   /\ UNCHANGED << pState, coordState, qrm,             699
631   swapGraph, ProofPublish, ProofLock,                  700
632   ProofOkRM, ProofOkRF >>                              701
633
634 Recipient(self) == init_rcp(self) \/ waitForD_rcp(self) 702
635
636 init_brcp(self) ==                                     703
637   /\ pc[self] = "init_brcp"                             704
638   /\ \/ /\ pc' = [pc EXCEPT ![self] = "BaskRF"]      705
639   \/ /\ pc' = [pc EXCEPT ![self] = "BRretrievingAsset"] 706
640   \/ /\ pc' = [pc EXCEPT ![self] = "BRdirectToS"]    707
641   \/ /\ pc' = [pc EXCEPT ![self] = "BRother"]        708
642   /\ UNCHANGED << assets, pState, coordState, qrm, qrf, 709
643   swapGraph, ProofPublish, ProofLock, ProofOkRM,      710
644   ProofOkRF >>                                         711
645
646 BRaskRF(self) == /\ pc[self] = "BRaskRF"               712
647   /\ IF coordState = "published"                       713
648   THEN /\ qrf' = TRUE                                  714
649
650      ELSE /\ TRUE                                       715
651      /\ qrf' = qrf                                       716
652      /\ pc' = [pc EXCEPT ![self] = "init_brcp"]      717
653      /\ UNCHANGED << assets, pState, coordState, qrm, 718
654      swapGraph, ProofPublish, ProofLock,                719
655      ProofOkRM, ProofOkRF >>                            720
656
657 BRretrievingAsset(self) == /\ pc[self] = "BRretrievingAsset" 721
658   /\ IF self \in Recipients                             722
659   /\ ProofOkRM = TRUE                                   723
660   /\ assets[AofR(self)] = "locked"                     724
661   THEN /\ assets' =                                     725
662   [assets EXCEPT ![AofR(self)] = "OwR"]              726
663   ELSE /\ TRUE                                          727
664   /\ UNCHANGED assets                                  728
665   /\ pc' =                                              729
666   [pc EXCEPT ![self] = "init_brcp"]                  730
667   /\ UNCHANGED << pState, coordState,                  731
668   qrm, qrf, swapGraph, ProofPublish, ProofLock,        732
669   ProofLock, ProofOkRM, ProofOkRF >>                    733
670
671 BRdirectToS(self) == /\ pc[self] = "BRdirectToS"        734
672   /\ IF self \in Recipients                             735
673   /\ assets[AofR(self)] = "OwR"                        736
674   THEN /\ assets' =                                     737
675   [assets EXCEPT ![AofR(self)] = "OwS"]              738
676   ELSE /\ TRUE                                          739
677   /\ UNCHANGED assets                                  740
678   /\ pc' = [pc EXCEPT ![self] = "init_brcp"]        741
679   /\ UNCHANGED << pState, coordState, qrm,             742
680   qrf, swapGraph, ProofPublish, ProofLock,              743
681   ProofOkRM, ProofOkRF >>                              744
682
683 BRother(self) == /\ pc[self] = "BRother"                745
684   /\ IF self \in Recipients                             746
685   /\ assets[AofR(self)] = "OwR"                        747
686   THEN /\ assets' =                                     748
687   [assets EXCEPT ![AofR(self)] = "other"]            749
688   ELSE /\ TRUE                                          750
689   /\ UNCHANGED assets                                  751
690   /\ pc' = [pc EXCEPT ![self] = "init_brcp"]        752
691   /\ UNCHANGED << pState, coordState, qrm,             753
692   qrf, swapGraph, ProofPublish, ProofLock,              754
693   ProofOkRM, ProofOkRF >>                              755
694
695 BRecipient(self) == init_brcp(self) \/ BRaskRF(self)   756
696   \/ BRretrievingAsset(self) \/ BRdirectToS(self)     757
697   \/ BRother(self)                                     758
698
699 (* Allow infinite stuttering to prevent deadlock on termination*)
700 Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
701   /\ UNCHANGED vars
702
703 Next == Publisher \/ Coordinator
704   \/ (\E self \in CSources: Source(self))
705   \/ (\E self \in BSources: BSource(self))
706   \/ (\E self \in CRecipients: Recipient(self))
707   \/ (\E self \in BRecipients: BRecipient(self))
708   \/ Terminating
709
710 Spec == /\ Init /\ [] [Next]_vars
711   /\ WF_vars(Next)
712   /\ WF_vars(Coordinator)
713   /\ \A self \in CSources : WF_vars(Source(self))
714   /\ \A self \in CRecipients : WF_vars(Recipient(self))
715
716 Termination == <<(\A self \in ProcSet: pc[self] = "Done")
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

718
719 (***** Liveness Property***** ) 787
720 (***** Checked using Model-Checking***** ) 788
721 789
722 AvailableS(a) == assets[a] = "OwS" \ / 790
723 (ProofOkRF = TRUE /\ assets[a] = "locked") 791
724 AvailableR(a) == assets[a] = "OwR" \ / 792
725 (ProofOkRM = TRUE /\ assets[a] = "locked") 793
726 AllParticipantsAreCorrect == (Pi = Pc) /\ swapGraph = "correct" 794
727 795
728 AtLeastOneCorrect == Pc # {} /\ (swapGraph \in SwapStates) 796
729 Finish(s,r) == pc[s] = "Done" /\ pc[r] = "Done" 797
730 798
731 OwnershipS == (\A s \in AssetsFromCS: AvailableS(s)) 799
732 OwnershipR == (\A r \in AssetsForCR : AvailableR(r)) 800
733 Ownership == AtLeastOneCorrect => <> (\ / OwnershipS 801
734 \ / OwnershipR) 802
735 803
736 Retrieving == AllParticipantsAreCorrect 804
737 -> (\A r \in Recipients : assets[AofR(r)] = "OwR") 805
738 806
739 (***** ) 807
740 (***** Consistency property***** ) 808
741 809
742 Consistency == \A s \in CSources, r \in CRecipients: 810
743 Finish(s,r) => AvailableS(AoS(s)) \ / AvailableR(AoF(r)) 811
744 812
745 (***** Safety ***** ) 813
746 (***** Proved using TLAPS ***** ) 814
747 815
748 \* the following is a theorem about all sets of the model 816
749 \* that are needed to ensure safety of our algorithm. 817
750 818
751 THEOREM SetsTheorem == 819
752 /\ CoordinatorID # PublisherID 820
753 /\ \A a \in AssetsFromCS : a \in Assets 821
754 /\ \A a \in AssetsForCR : a \in Assets 822
755 /\ \A s \in Sources : s \in Pi 823
756 /\ \A r \in Recipients : r \in Pi 824
757 /\ \A p \in Pc: /\ p \in Pi 825
758 /\ \ / /\ p \in CSources 826
759 /\ p \in Sources 827
760 \ / /\ p \in CRecipients 828
761 /\ p \in Recipients 829
762 /\ p \notin BSources 830
763 /\ p \notin BRecipients 831
764 /\ \A s \in CSources : /\ s \in Sources 832
765 /\ s \in Pi 833
766 /\ s \in Pc 834
767 /\ s \notin BSources 835
768 /\ s # PublisherID 836
769 /\ s # CoordinatorID 837
770 /\ s \notin CRecipients 838
771 /\ s \notin Recipients 839
772 /\ s \notin BRecipients 840
773 /\ AoF(s) \in AssetsFromCS 841
774 /\ \A r \in CRecipients : /\ r \in Recipients 842
775 /\ r \in Pi 843
776 /\ r \in Pc 844
777 /\ r \notin BRecipients 845
778 /\ r # PublisherID 846
779 /\ r # CoordinatorID 847
780 /\ AofR(r) \in AssetsForCR 848
781 /\ SofA(AoF(r)) \in Sources 849
782 /\ \A bs \in BSources : /\ bs \in Pi 850
783 /\ bs \in Sources 851
784 /\ bs \notin CSources 852
785 /\ bs \notin Pc 853
786 /\ bs # PublisherID 854

```

```

787 /\ bs # CoordinatorID
788 /\ AoF(bs) \notin AssetsFromCS
789 /\ \A br \in BRecipients: /\ br \in Recipients
790 /\ br \in Pi
791 /\ br \notin Pc
792 /\ br \notin CRecipients
793 /\ br # PublisherID
794 /\ br # CoordinatorID
795 /\ AofR(br) \notin AssetsForCR
796 /\ ProcSet = {PublisherID} \cup {CoordinatorID} \cup
797 (CSources) \cup (BSources) \cup (CRecipients) \cup
798 (BRecipients)
799 /\ Pi = Sources \cup Recipients
800 /\ Pc = Pi \cap Correct
801 /\ CSources = Pc \cap Sources
802 /\ CRecipients = Pc \cap Recipients
803 /\ BSources = Sources \ CSources
804 /\ BRecipients = Recipients \ CRecipients
805 /\ BSources \cap CSources = {}
806 /\ BRecipients \cap CRecipients = {}
807 /\ AStates = {"OwS", "OwR", "locked", "other"}
808 /\ CStates = {"init", "published", "okRM", "okRF"}
809 /\ PStates = {"init", "publish"}
810 /\ SwapStates = {"init", "correct", "different"}
811 /\ \A s \in Sources: SofA(AoS(s)) = s
812 /\ \A s \in Recipients: RofA(AoF(s)) = s
813 /\ \A s \in Assets: AoS(SofA(s)) = s
814 /\ \A s \in Assets: AofR(RofA(s)) = s
815 /\ \A s \in Sources: AoS(s) \in Assets
816 /\ \A a \in Assets: SofA(a) \in Sources
817 BY DEF ProcSet, CSources, CRecipients, Sources, Recipients,
818 AssetsFromCS, Assets, AssetsForCR, AoF, AofR, SofA, RofA,
819 Pi, Pc, BSources, BRecipients, PublisherID, CoordinatorID,
820 AStates, CStates, PStates, SwapStates
821
822 \* the following predicate is a type correctness invariant
823
824 TypeOk == /\ assets \in [Assets -> AStates]
825 /\ pState \in PStates
826 /\ coordState \in CStates
827 /\ ProofLock \in [Sources -> \in BOOLEAN]
828 /\ ProofPublish \in \in BOOLEAN
829 /\ ProofOkRM \in BOOLEAN
830 /\ ProofOkRF \in BOOLEAN
831 /\ qrm \subseteqq Sources
832 /\ qrf \in \in BOOLEAN
833 /\ swapGraph \in SwapStates
834 /\ pc[CoordinatorID] \in
835 { "init_c", "decision", "decisionValid",
836 "decisionAbort", "Done" }
837 /\ pc[CoordinatorID] = "Done" =>
838 coordState \in { "okRM", "okRF" }
839 /\ pc \in [ ProcSet -> { "init_c", "decision",
840 "decisionValid", "decisionAbort", "Done",
841 "init_p", "init_src", "lock", "published",
842 "waitForD", "refunded", "Done", "init_bsrc",
843 "BdirectToR", "Bother", "BaskRM", "BlockAsset",
844 "BSaskRF", "BrecoveringAsset", "init_rcp",
845 "waitForD_rcp", "redeemed", "exit_rcp", "Done",
846 "init_brcp", "BRaskRF", "BRretrievingAsset",
847 "BRdirectToS", "BROther" } ]
848
849 \* the following predicates are needed to define the
850 \* coordinator invariant
851
852 init_cInv ==
853 /\ coordState = "init"
854 /\ ProofOkRM = FALSE

```

```

856      /\ Proof0kRF = FALSE
857      /\ ProofPublish = FALSE
858      /\ qrf = FALSE
859      /\ qrm = {}
860      /\ \A s \in Sources: ProofLock[s] = FALSE
861      /\ \A s \in CSources :
862          /\ pc[s] \in {"init_src", "Done"}
863          /\ ProofLock[s] = FALSE
864          /\ assets[AofS(s)] = "OwS"
865      /\ \A r \in CRecipients :
866          pc[r] \in {"init_rcp", "Done"}
867      /\ swapGraph = "init" => pState = "init"
868      /\ swapGraph = "correct" => pState = "publish"
869      /\ swapGraph = "different" => pState = "publish"
870      /\ \A a \in AssetsFromCS: assets[a] = "OwS"
871      /\ pState = "publish" => pc[PublisherID] = "Done"
872
873 decisionInv ==
874     /\ coordState = "published"
875     /\ pState \in {"publish", "Done"}
876     /\ Proof0kRM = FALSE
877     /\ Proof0kRF = FALSE
878     /\ ProofPublish = TRUE
879     /\ pc[PublisherID] = "Done"
880     /\ \A s \in Sources:
881         /\ s \in qrm => ProofLock[s] = TRUE
882         /\ ProofLock[s] = TRUE =>
883             assets[AofS(s)] = "locked"
884     /\ \A s \in CSources :
885         /\ pc[s] \in
886             {"published", "waitForD", "init_src", "lock", "Done"}
887         /\ pc[s] \in {"published", "waitForD"}
888             => /\ ProofLock[s] = TRUE
889                 /\ assets[AofS(s)] = "locked"
890         /\ pc[s] \in {"init_src", "lock", "Done"}
891             => /\ ProofLock[s] = FALSE
892                 /\ assets[AofS(s)] = "OwS"
893         /\ pc[s] \in
894             {"init_src", "lock", "Done", "published"}
895             => s \notin qrm
896         /\ s \in qrm => pc[s] = "waitForD"
897     /\ \A r \in CRecipients : pc[r] \in
898         {"init_rcp", "waitForD_rcp", "Done"}
899     /\ \A a \in AssetsFromCS: assets[a] \in
900         {"locked", "OwS"}
901
902 decisionValidInv ==
903     /\ coordState = "published"
904     /\ pState \in {"publish", "Done"}
905     /\ Proof0kRM = FALSE
906     /\ Proof0kRF = FALSE
907     /\ ProofPublish = TRUE
908     /\ pc[PublisherID] = "Done"
909     /\ qrm = Sources
910     /\ \A s \in Sources:
911         /\ ProofLock[s] = TRUE
912         /\ assets[AofS(s)] = "locked"
913     /\ \A s \in CSources: pc[s] \in {"waitForD"}
914     /\ \A r \in CRecipients :
915         /\ pc[r] \in
916             {"init_rcp", "waitForD_rcp", "Done"}
917         /\ assets[AofR(r)] = "locked"
918         /\ pc[r] = "init_src"
919             => assets[AofR(r)] = "locked"
920     /\ qrm = Sources
921     => \A a \in Assets : assets[a] = "locked"
922
923 decisionAbortInv ==
924     /\ coordState = "published"
925     /\ pState \in {"publish", "Done"}
926     /\ Proof0kRM = FALSE
927     /\ Proof0kRF = FALSE
928     /\ ProofPublish = TRUE
929     /\ pc[PublisherID] = "Done"
930     /\ qrf = TRUE
931     /\ \A s \in CSources :
932         /\ assets[AofS(s)] \in {"locked", "OwS"}
933         /\ pc[s] \in
934             {"init_src", "lock", "published", "waitForD", "Done"}
935         /\ pc[s] = "Done" => assets[AofS(s)] = "OwS"
936         /\ pc[s] = "init_src"
937             => assets[AofS(s)] = "OwS"
938     /\ \A a \in AssetsFromCS:
939         assets[a] \in {"locked", "OwS"}
940
941 okRMInv ==
942     /\ coordState = "okRM"
943     /\ Proof0kRM = TRUE
944     /\ Proof0kRF = FALSE
945     /\ ProofPublish = TRUE
946     /\ qrm = Sources
947     /\ pc[PublisherID] = "Done"
948     /\ \A s \in CSources:
949         pc[s] \in {"waitForD", "Done"}
950     /\ \A r \in CRecipients :
951         /\ pc[r] \in
952             {"init_rcp", "waitForD_rcp", "Done"}
953         /\ assets[AofR(r)] \in {"locked", "OwR"}
954         /\ pc[r] = "Done"
955             => assets[AofR(r)] \in {"OwR", "locked"}
956         /\ pc[r] \in {"init_rcp", "waitForD_rcp"}
957             => assets[AofR(r)] = "locked"
958         /\ pc[r] = "init_src"
959             => assets[AofR(r)] = "locked"
960     /\ qrm = Sources
961     => \A a \in AssetsForCR :
962         assets[a] \in {"locked", "OwR"}
963
964 okRFInv ==
965     /\ Proof0kRM = FALSE
966     /\ Proof0kRF = TRUE
967     /\ ProofPublish = TRUE
968     /\ pc[PublisherID] = "Done"
969     /\ qrf = TRUE
970     /\ \A s \in CSources :
971         /\ assets[AofS(s)] \in {"locked", "OwS"}
972         /\ pc[s] = "Done" => assets[AofS(s)] = "OwS"
973         /\ pc[s] = "init_src" => assets[AofS(s)] = "OwS"
974     /\ \A a \in AssetsFromCS:
975         assets[a] \in {"locked", "OwS"}
976
977 CoordInv2 ==
978     /\ pc[CoordinatorID] = "init_c" => init_cInv
979     /\ pc[CoordinatorID] = "decision" => decisionInv
980     /\ pc[CoordinatorID] = "decisionValid" => decisionValidInv
981     /\ pc[CoordinatorID] = "decisionAbort" => decisionAbortInv
982     /\ (pc[CoordinatorID] = "Done" /\ coordState = "okRM" )
983         => okRMInv
984     /\ (pc[CoordinatorID] = "Done" /\ coordState = "okRF" )
985         => okRFInv
986
987 (* the inductive invariant for proving the safety property*)
988 Inv == Type0k /\ CoordInv2
989
990 THEOREM InitImpliesType0k ==
991     ASSUME Init
992     PROVE Type0k
993 <1>1. assets \in [Assets -> AStates]

```

994	BY SetsTheorem DEF TypeOk, Init	1063	<2> SUFFICES ASSUME NEW self \in CSources,
995	<1>2. pState \in PStates	1064	Source(self)
996	BY SetsTheorem DEF TypeOk, Init	1065	PROVE TypeOk'
997	<1>3. coordState \in CStates	1066	BY <1>3
998	BY SetsTheorem DEF TypeOk, Init	1067	<2>1. CASE init_src(self)
999	<1>4. ProofLock \in [Sources → { TRUE, FALSE }]	1068	BY <2>1, <1>3, SetsTheorem DEF TypeOk, init_src
1000	BY SetsTheorem DEF TypeOk, Init	1069	<2>2. CASE lock(self)
1001	<1>5. ProofOkRM \in { TRUE, FALSE }	1070	BY <2>2, <1>3, SetsTheorem DEF TypeOk, lock
1002	BY SetsTheorem DEF TypeOk, Init	1071	<2>3. CASE published(self)
1003	<1>6. ProofOkRM \in BOOLEAN	1072	BY <2>3, <1>3, SetsTheorem DEF TypeOk, published
1004	BY SetsTheorem DEF TypeOk, Init	1073	<2>4. CASE waitForD(self)
1005	<1>7. ProofOkRF \in BOOLEAN	1074	BY <2>4, <1>3, SetsTheorem DEF TypeOk, waitForD
1006	BY SetsTheorem DEF TypeOk, Init	1075	<2>7. QED
1007	<1>8. qrm \subseteqq Sources	1076	BY <1>3, <2>1, <2>2, <2>3, <2>4 DEF Source
1008	BY SetsTheorem DEF TypeOk, Init	1077	
1009	<1>9. qrf \in { TRUE, FALSE }	1078	<1>4. CASE \E self \in BSources: BSource(self)
1010	BY SetsTheorem DEF TypeOk, Init	1079	<2> SUFFICES ASSUME NEW self \in BSources,
1011	<1>10. swapGraph \in SwapStates	1080	BSource(self)
1012	BY SetsTheorem DEF TypeOk, Init	1081	PROVE TypeOk'
1013	<1>11. pc[CoordinatorID] = "Done"	1082	BY <1>4
1014	=> coordState \in { "okRM", "okRF" }	1083	<2>1. CASE init_bsrc(self)
1015	BY SetsTheorem DEF TypeOk, Init	1084	BY <2>1, <1>4, SetsTheorem DEF TypeOk, init_bsrc
1016	<1>12. pc[CoordinatorID] \in { "init_c", "decision",	1085	<2>2. CASE BdirectToR(self)
1017	"decisionValid", "decisionAbort", "Done" }	1086	BY <2>2, <1>4, SetsTheorem DEF TypeOk, BdirectToR
1018	BY SetsTheorem DEF TypeOk, Init	1087	<2>3. CASE Bother(self)
1019	<1>13. pc \in [ProcSet → { "init_c", "decision",	1088	BY <2>3, <1>4, SetsTheorem DEF TypeOk, Bother
1020	"decisionValid", "decisionAbort", "Done",	1089	<2>4. CASE BaskRM(self)
1021	"init_p", "init_src", "lock", "published", "waitForD",	1090	BY <2>4, <1>4, SetsTheorem DEF TypeOk, BaskRM
1022	"refunded", "Done", "init_bsrc", "BdirectToR", "Bother",	1091	<2>5. CASE BblockAsset(self)
1023	"BaskRM", "BblockAsset", "BSaskRF", "BrecoveringAsset",	1092	BY <2>5, <1>4, SetsTheorem DEF TypeOk, BblockAsset
1024	"init_rcp", "waitForD_rcp", "redeemed", "exit_rcp",	1093	<2>6. CASE BSaskRF(self)
1025	"Done", "init_brcp", "BRaskRF", "BRretrievingAsset",	1094	BY <2>6, <1>4, SetsTheorem DEF TypeOk, BSaskRF
1026	"BRdirectToS", "BROther" }]	1095	<2>7. CASE BrecoveringAsset(self)
1027	BY SetsTheorem DEF TypeOk, Init	1096	BY <2>7, <1>4, SetsTheorem DEF TypeOk, BrecoveringAsset
1028	<1>14. QED	1097	<2>8. QED
1029	BY <1>1, <1>10, <1>11, <1>12, <1>13, <1>2, <1>3,	1098	BY <1>4, <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7
1030	<1>4, <1>5, <1>6, <1>7, <1>8, <1>9 DEF TypeOk	1099	DEF BSource
1031		1100	
1032	THEOREM InitImpliesCoord ==	1101	<1>5. CASE \E self \in CRecipients: Recipient(self)
1033	ASSUME TypeOk,	1102	<2> SUFFICES ASSUME NEW self \in CRecipients,
1034	Init	1103	Recipient(self)
1035	PROVE CoordInv2	1104	PROVE TypeOk'
1036	BY SetsTheorem DEF TypeOk, Init, CoordInv2, init_cInv	1105	BY <1>5
1037		1106	<2>1. CASE init_rcp(self)
1038	THEOREM InitImpliesInv ==	1107	BY <2>1, <1>5, SetsTheorem DEF TypeOk, init_rcp
1039	ASSUME Init	1108	<2>2. CASE waitForD_rcp(self)
1040	PROVE TypeOk /\ CoordInv2	1109	BY <2>2, <1>5, SetsTheorem DEF TypeOk, waitForD_rcp
1041	BY InitImpliesCoord, InitImpliesTypeOk	1110	<2>5. QED
1042		1111	BY <1>5, <2>1, <2>2 DEF Recipient
1043	THEOREM TypeOkInvariant ==	1112	
1044	ASSUME TypeOk,	1113	<1>6. CASE \E self \in BRecipients: BRecipient(self)
1045	Next	1114	<2> SUFFICES ASSUME NEW self \in BRecipients,
1046	PROVE TypeOk'	1115	BRecipient(self)
1047	<1>1. CASE Publisher	1116	PROVE TypeOk'
1048	BY <1>1, SetsTheorem DEF TypeOk, Publisher, init_p	1117	BY <1>6
1049	<1>2. CASE Coordinator	1118	<2>1. CASE init_brcp(self)
1050	<2>1. CASE init_c	1119	BY <2>1, <1>6, SetsTheorem DEF TypeOk, init_brcp
1051	BY <2>1, <1>2, SetsTheorem DEF TypeOk, init_c	1120	<2>2. CASE BRaskRF(self)
1052	<2>2. CASE decision	1121	BY <2>2, <1>6, SetsTheorem DEF TypeOk, BRaskRF
1053	BY <2>2, <1>2, SetsTheorem DEF TypeOk, decision	1122	<2>3. CASE BRretrievingAsset(self)
1054		1123	BY <2>3, <1>6, SetsTheorem DEF TypeOk, BRretrievingAsset
1055	<2>3. CASE decisionValid	1124	<2>4. CASE BRdirectToS(self)
1056	BY <2>3, <1>2, SetsTheorem DEF TypeOk, decisionValid	1125	BY <2>4, <1>6, SetsTheorem DEF TypeOk, BRdirectToS
1057	<2>4. CASE decisionAbort	1126	<2>5. CASE BROther(self)
1058	BY <2>4, <1>2, SetsTheorem DEF TypeOk, decisionAbort	1127	BY <2>5, <1>6, SetsTheorem DEF TypeOk, BROther
1059	<2>7. QED	1128	<2>6. QED
1060	BY <1>2, <2>1, <2>2, <2>3, <2>4 DEF Coordinator	1129	BY <1>6, <2>1, <2>2, <2>3, <2>4, <2>5 DEF BRecipient
1061		1130	
1062	<1>3. CASE \E self \in CSources: Source(self)	1131	<1>7. CASE Terminating

```

1132 BY <1>7 DEF TypeOk, Terminating, vars 1201
1133 <1>8. QED 1202
1134 BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6, <1>7 DEF Next 1203
1135 1204
1136 1205
1137 THEOREM CoordInvariant == 1206
1138 ASSUME CoordInv2, TypeOk, TypeOk', 1207
1139 Next 1208
1140 PROVE CoordInv2' 1209
1141 <1>1. CASE pc[CoordinatorID] = "init_c" 1210
1142 <2>0. init_cInv 1211
1143 BY <1>1 DEF CoordInv2 1212
1144 <2>1. CASE Publisher 1213
1145 BY <1>1, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher, 1214
1146 init_p, init_cInv, CoordInv2 1215
1147 <2>2. CASE Coordinator 1216
1148 <3>1. CASE init_c 1217
1149 <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)' 1218
1150 BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, 1219
1151 init_c, init_cInv 1220
1152 <4>2. (pc[CoordinatorID] = "decision" => decisionInv)' 1221
1153 BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, 1222
1154 init_c, init_cInv, decisionInv 1223
1155 <4>3. (pc[CoordinatorID] = "decisionValid" => 1224
1156 decisionValidInv)' 1225
1157 BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, 1226
1158 init_c, init_cInv 1227
1159 <4>4. (pc[CoordinatorID] = "decisionAbort" => 1228
1160 decisionAbortInv)' 1229
1161 BY <2>0, <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, 1230
1162 init_c, init_cInv 1231
1163 <4>5. ((pc[CoordinatorID] = "Done" /\ coordState="okRM") 1232
1164 => okRMInv)' 1233
1165 BY <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c, 1234
1166 init_cInv, CoordInv2 1235
1167 <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF") 1236
1168 => okRFInv)' 1237
1169 BY <1>1, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c, 1238
1170 init_cInv, CoordInv2 1239
1171 <4>7. QED 1240
1172 BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2 1241
1173 <3>2. CASE decision 1242
1174 BY <1>1, <3>2, <2>2, SetsTheorem DEF TypeOk, decision 1243
1175 <3>3. CASE decisionValid 1244
1176 BY <1>1, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionVal 1245
1177 <3>4. CASE decisionAbort 1246
1178 BY <1>1, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbor 1247
1179 <3>7. QED 1248
1180 BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator 1249
1181 1250
1182 <2>3. CASE \E self \in CSources: Source(self) 1251
1183 <3> SUFFICES ASSUME NEW self \in CSources, 1252
1184 Source(self) 1253
1185 PROVE CoordInv2' 1254
1186 BY <2>3 1255
1187 <3>1. CASE init_src(self) 1256
1188 BY <3>1, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk, 1257
1189 init_src, init_cInv, CoordInv2 1258
1190 <3>2. CASE lock(self) 1259
1191 BY <3>2, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk, 1260
1192 lock, init_cInv, CoordInv2 1261
1193 <3>3. CASE published(self) 1262
1194 BY <3>3, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk, 1263
1195 published, init_cInv, CoordInv2 1264
1196 <3>4. CASE waitForD(self) 1265
1197 BY <3>4, <2>0, <1>1, <2>3, SetsTheorem DEF TypeOk, 1266
1198 waitForD, init_cInv, CoordInv2 1267
1199 <3>7. QED 1268
1200 BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source 1269

```

```

<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 BOther(self)
BY <3>5, <1>1, <2>0, <2>6, SetsTheorem DEF TypeOk,
BOther, 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

```

1270	BY <1>1, <2>0, <2>7, SetsTheorem DEF TypeOk, Terminating,	1339	<4>4. (pc[CoordinatorID] = "decisionAbort" =>
1271	CoordInv2	1340	decisionAbortInv)'
1272	<2>8. QED	1341	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1273	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next	1342	lock, decisionInv, decisionValidInv
1274		1343	<4>5. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
1275	<1>2. CASE pc[CoordinatorID] = "decision"	1344	=> okRMInv)'
1276	<2>0. decisionInv	1345	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1277	BY <1>2 DEF CoordInv2	1346	lock, decisionInv
1278	<2>1. CASE Publisher	1347	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
1279	BY <1>2, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,	1348	=> okRFInv)'
1280	init_p, decisionInv, CoordInv2	1349	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1281	<2>2. CASE Coordinator	1350	lock, decisionInv
1282	<3>1. CASE init_c	1351	<4>7. QED
1283	BY <1>2, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c	1352	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1284	<3>2. CASE decision	1353	
1285	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'	1354	<3>3. CASE published(self)
1286	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1355	BY <3>3, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1287	decision	1356	published, decisionInv, CoordInv2
1288	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'	1357	<3>4. CASE waitForD(self)
1289	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1358	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
1290	decision, decisionInv	1359	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1291	<4>3. (pc[CoordinatorID] = "decisionValid" =>	1360	waitForD, decisionInv
1292	decisionValidInv)'	1361	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
1293	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1362	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1294	decision, decisionInv, ValidTransfer, decisionValidInv	1363	waitForD, decisionInv
1295	<4>4. (pc[CoordinatorID] = "decisionAbort" =>	1364	<4>3. (pc[CoordinatorID] = "decisionValid" =>
1296	decisionAbortInv)'	1365	decisionValidInv)'
1297	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1366	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1298	decision, decisionInv, AbortTransfer, decisionAbortInv	1367	waitForD, decisionInv, decisionValidInv
1299	<4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM")	1368	<4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")
1300	=> okRMInv)'	1369	=> okRMInv)'
1301	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1370	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1302	decision	1371	waitForD, decisionInv
1303	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF")	1372	<4>5. (pc[CoordinatorID] = "decisionAbort" =>
1304	=> okRFInv)'	1373	decisionAbortInv)'
1305	BY <2>0, <1>2, <3>2, <2>2, SetsTheorem DEF TypeOk,	1374	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1306	decision	1375	waitForD, decisionInv, decisionValidInv
1307	<4>7. QED	1376	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")
1308	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2	1377	=> okRFInv)'
1309		1378	BY <3>4, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,
1310	<3>3. CASE decisionValid	1379	waitForD, decisionInv
1311	BY <1>2, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid	1380	<4>7. QED
1312	<3>4. CASE decisionAbort	1381	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1313	BY <1>2, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort	1382	<3>7. QED
1314	<3>7. QED	1383	BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source
1315	BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator	1384	
1316		1385	<2>4. CASE \E self \in BSources: BSource(self)
1317	<2>3. CASE \E self \in CSources: Source(self)	1386	<3> SUFFICES ASSUME NEW self \in BSources,
1318	<3> SUFFICES ASSUME NEW self \in CSources,	1387	BSource(self)
1319	Source(self)	1388	PROVE CoordInv2'
1320	PROVE CoordInv2'	1389	
1321	BY <2>3	1390	BY <2>4
1322	<3>1. CASE init_src(self)	1391	<3>1. CASE init_bsrc(self)
1323	BY <3>1, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,	1392	BY <3>1, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,
1324	init_src, decisionInv, CoordInv2	1393	init_bsrc, decisionInv, CoordInv2
1325	<3>2. CASE lock(self)	1394	<3>2. CASE BdirectToR(self)
1326	<4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]	1395	<8>1. CASE assets[AofS(self)] = "OwS"
1327	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,	1396	<9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]
1328	lock, decisionInv	1397	BY <8>1, <3>2, <2>0, <1>2, <2>4, SetsTheorem
1329	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'	1398	DEF TypeOk, BdirectToR, decisionInv, CoordInv2
1330	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,	1399	<9>2. QED
1331	lock, decisionInv	1400	BY <9>1, <8>1, <3>2, <2>0, <1>2, <2>4, SetsTheorem
1332	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'	1401	DEF TypeOk, BdirectToR, decisionInv, CoordInv2
1333	BY <3>2, <2>0, <1>2, <2>3, <4>0, SetsTheorem	1402	<8>2. CASE assets[AofS(self)] # "OwS"
1334	DEF TypeOk, lock, decisionInv	1403	BY <8>2, <3>2, <2>0, <1>2, <2>4, SetsTheorem
1335	<4>3. (pc[CoordinatorID] = "decisionValid" =>	1404	DEF TypeOk, BdirectToR, decisionInv, CoordInv2
1336	decisionValidInv)'	1405	<8>3. QED
1337	BY <3>2, <2>0, <1>2, <2>3, SetsTheorem DEF TypeOk,	1406	BY <8>1, <8>2
1338	lock, decisionInv, decisionValidInv	1407	<3>3. CASE Bother(self)
			<8>1. CASE assets[AofS(self)] = "OwS"

1408	<9>1. assets' = [assets EXCEPT ![AofS(self)] = "other"	1477	<3>4. CASE BRdirectToS(self)
1409	BY <8>1, <3>3, <2>0, <1>2, <2>4, SetsTheorem	1478	BY <3>4, <1>2, <2>0, <2>6, SetsTheorem DEF TypeOk,
1410	DEF TypeOk, Bother, decisionInv, CoordInv2	1479	BRdirectToS, decisionInv, CoordInv2
1411	<9>2. QED	1480	<3>5. CASE BRouter(self)
1412	BY <9>1, <8>1, <3>3, <2>0, <1>2, <2>4, SetsTheorem	1481	BY <3>5, <1>2, <2>0, <2>6, SetsTheorem DEF TypeOk,
1413	DEF TypeOk, Bother, decisionInv, CoordInv2	1482	BRouter, decisionInv, CoordInv2
1414	<8>2. CASE assets[AofS(self)] # "OwS"	1483	<3>6. QED
1415	BY <8>2, <3>3, <2>0, <1>2, <2>4, SetsTheorem	1484	BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient
1416	DEF TypeOk, Bother, decisionInv, CoordInv2	1485	
1417	<8>3. QED	1486	<2>7. CASE Terminating
1418	BY <8>1, <8>2	1487	BY <1>2, <2>0, <2>7, SetsTheorem DEF TypeOk,
1419	<3>4. CASE BaskRM(self)	1488	Terminating, CoordInv2
1420	BY <3>4, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,	1489	<2>8. QED
1421	BaskRM, decisionInv, CoordInv2	1490	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next
1422		1491	
1423	<3>5. CASE BlockAsset(self)	1492	<1>4. CASE pc[CoordinatorID] = "decisionValid"
1424	<8>1. CASE (ProofPublish = TRUE /\	1493	<2>0. decisionValidInv
1425	assets[AofS(self)] = "OwS")	1494	BY <1>4 DEF CoordInv2
1426	<9>1. assets' =	1495	<2>1. CASE Publisher
1427	[assets EXCEPT ![AofS(self)] = "locked"]	1496	BY <1>4, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,
1428	BY <8>1, <3>5, <2>0, <1>2, <2>4, SetsTheorem	1497	init_p, decisionValidInv, CoordInv2
1429	DEF TypeOk, BlockAsset, decisionInv, CoordInv2	1498	<2>2. CASE Coordinator
1430	<9>2. QED	1499	<3>1. CASE init_c
1431	BY <9>1, <8>1, <3>5, <2>0, <1>2, <2>4, SetsTheorem	1500	BY <1>4, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c
1432	DEF TypeOk, BlockAsset, decisionInv, CoordInv2	1501	<3>2. CASE decision
1433	<8>2. CASE ~(ProofPublish = TRUE /\	1502	BY <1>4, <3>2, <2>2, SetsTheorem DEF TypeOk, decision
1434	assets[AofS(self)] = "OwS")	1503	
1435	BY <8>2, <3>5, <2>0, <1>2, <2>4, SetsTheorem	1504	<3>3. CASE decisionValid
1436	DEF TypeOk, BlockAsset, decisionInv, CoordInv2	1505	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
1437	<8>3. QED	1506	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1438	BY <8>1, <8>2	1507	decisionValid
1439	<3>6. CASE BSaskRF(self)	1508	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
1440	BY <3>6, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,	1509	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1441	BSaskRF, decisionInv, CoordInv2	1510	decisionValid
1442	<3>7. CASE BrecoveringAsset(self)	1511	<4>3. (pc[CoordinatorID] = "decisionValid" =>
1443	BY <3>7, <2>0, <1>2, <2>4, SetsTheorem DEF TypeOk,	1512	decisionValidInv)'
1444	BrecoveringAsset, decisionInv, CoordInv2	1513	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1445	<3>8. QED	1514	decisionValid, decisionValidInv
1446	BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7	1515	<4>4. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM"
1447	DEF BSource	1516	=> okRMInv)'
1448		1517	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1449	<2>5. CASE \E self \in CRecipients: Recipient(self)	1518	decisionValid, decisionValidInv, okRMInv
1450	<3> SUFFICES ASSUME NEW self \in CRecipients,	1519	<4>5. (pc[CoordinatorID] = "decisionAbort" =>
1451	Recipient(self)	1520	decisionAbortInv)'
1452	PROVE CoordInv2'	1521	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1453	BY <2>5	1522	decisionValid, decisionValidInv
1454	<3>1. CASE init_rcp(self)	1523	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF"
1455	BY <3>1, <1>2, <2>0, <2>5, SetsTheorem DEF TypeOk,	1524	=> okRFInv)'
1456	init_rcp, decisionInv, CoordInv2	1525	BY <2>0, <1>4, <3>3, <2>2, SetsTheorem DEF TypeOk,
1457	<3>2. CASE waitForD_rcp(self)	1526	decisionValid, decisionValidInv, okRMInv
1458	BY <3>2, <1>2, <2>0, <2>5, SetsTheorem DEF TypeOk,	1527	<4>7. QED
1459	waitForD_rcp, decisionInv, CoordInv2	1528	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1460	<3>5. QED	1529	<3>4. CASE decisionAbort
1461	BY <2>5, <3>1, <3>2 DEF Recipient	1530	BY <1>4, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort
1462		1531	<3>7. QED
1463	<2>6. CASE \E self \in BRecipients: BRecipient(self)	1532	BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator
1464	<3> SUFFICES ASSUME NEW self \in BRecipients,	1533	
1465	BRecipient(self)	1534	<2>3. CASE \E self \in CSources: Source(self)
1466	PROVE CoordInv2'	1535	<3> SUFFICES ASSUME NEW self \in CSources,
1467	BY <2>6	1536	Source(self)
1468	<3>1. CASE init_brpc(self)	1537	PROVE CoordInv2'
1469	BY <3>1, <1>2, <2>0, <2>6, SetsTheorem DEF TypeOk,	1538	BY <2>3
1470	init_brpc, decisionInv, CoordInv2	1539	<3>1. CASE init_src(self)
1471	<3>2. CASE BRaskRF(self)	1540	BY <3>1, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
1472	BY <3>2, <1>2, <2>0, <2>6, SetsTheorem DEF TypeOk,	1541	init_src, decisionValidInv, CoordInv2
1473	BRaskRF, decisionInv, CoordInv2	1542	<3>2. CASE lock(self)
1474	<3>3. CASE BRretrievingAsset(self)	1543	<4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]
1475	BY <3>3, <1>2, <2>0, <2>6, SetsTheorem DEF TypeOk,	1544	BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk,
1476	BRretrievingAsset, decisionInv, CoordInv2	1545	lock, decisionValidInv


```

1546 <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)' 1615
1547 BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1616
1548 lock, decisionValidInv 1617
1549 <4>2. (pc[CoordinatorID] = "decision" => decisionInv)' 1618
1550 BY <3>2, <2>0, <1>4, <2>3, <4>0, SetsTheorem 1619
1551 DEF TypeOk, lock, decisionValidInv 1620
1552 <4>3. (pc[CoordinatorID] = "decisionValid" => 1621
1553 decisionValidInv)' 1622
1554 BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1623
1555 lock, decisionValidInv, decisionValidInv 1624
1556 <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")1625
1557 => okRMInv)' 1626
1558 BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1627
1559 lock, decisionValidInv 1628
1560 <4>5. (pc[CoordinatorID] = "decisionAbort" => 1629
1561 decisionAbortInv)' 1630
1562 BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1631
1563 lock, decisionValidInv, decisionValidInv 1632
1564 <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")1633
1565 => okRFInv)' 1634
1566 BY <3>2, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1635
1567 lock, decisionValidInv 1636
1568 <4>7. QED 1637
1569 BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2 1638
1570 1639
1571 <3>3. CASE published(self) 1640
1572 BY <3>3, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1641
1573 published, decisionValidInv, CoordInv2 1642
1574 <3>4. CASE waitForD(self) 1643
1575 <4>1. (pc[CoordinatorID] = "init_c" => init_cInv)' 1644
1576 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1645
1577 waitForD, decisionValidInv 1646
1578 <4>2. (pc[CoordinatorID] = "decision" => decisionInv)' 1647
1579 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1648
1580 waitForD, decisionValidInv 1649
1581 <4>3. (pc[CoordinatorID] = "decisionValid" => 1650
1582 decisionValidInv)' 1651
1583 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1652
1584 waitForD, decisionValidInv, decisionValidInv 1653
1585 <4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")1654
1586 => okRMInv)' 1655
1587 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1656
1588 waitForD, decisionValidInv 1657
1589 <4>5. (pc[CoordinatorID] = "decisionAbort" => 1658
1590 decisionAbortInv)' 1659
1591 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1660
1592 waitForD, decisionValidInv, decisionValidInv 1661
1593 <4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")1662
1594 => okRFInv)' 1663
1595 BY <3>4, <2>0, <1>4, <2>3, SetsTheorem DEF TypeOk, 1664
1596 waitForD, decisionValidInv 1665
1597 <4>7. QED 1666
1598 BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2 1667
1599 <3>7. QED 1668
1600 BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source 1669
1601 1670
1602 <2>4. CASE \E self \in BSources: BSource(self) 1671
1603 <3> SUFFICES ASSUME NEW self \in BSources, 1672
1604 BSource(self) 1673
1605 PROVE CoordInv2' 1674
1606 BY <2>4 1675
1607 <3>1. CASE init_bsrc(self) 1676
1608 BY <3>1, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk, 1677
1609 init_bsrc, decisionValidInv, CoordInv2 1678
1610 <3>2. CASE BdirectToR(self) 1679
1611 <8>1. CASE assets[AofS(self)] = "OwS" 1680
1612 <9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]1681
1613 BY <8>1, <3>2, <2>0, <1>4, <2>4, SetsTheorem 1682
1614 DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2 1683
1615 <9>2. QED
1616 BY <9>1, <8>1, <3>2, <2>0, <1>4, <2>4, SetsTheorem
1617 DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2
1618 <8>2. CASE assets[AofS(self)] # "OwS"
1619 BY <8>2, <3>2, <2>0, <1>4, <2>4, SetsTheorem
1620 DEF TypeOk, BdirectToR, decisionValidInv, CoordInv2
1621 <8>3. QED
1622 BY <8>1, <8>2
1623 <3>3. CASE Bother(self)
1624 <8>1. CASE assets[AofS(self)] = "OwS"
1625 <9>1. assets' = [assets EXCEPT ![AofS(self)] = "other"]
1626 BY <8>1, <3>3, <2>0, <1>4, <2>4, SetsTheorem
1627 DEF TypeOk, Bother, decisionValidInv, CoordInv2
1628 <9>2. QED
1629 BY <9>1, <8>1, <3>3, <2>0, <1>4, <2>4, SetsTheorem
1630 DEF TypeOk, Bother, decisionValidInv, CoordInv2
1631 <8>2. CASE assets[AofS(self)] # "OwS"
1632 BY <8>2, <3>3, <2>0, <1>4, <2>4, SetsTheorem
1633 DEF TypeOk, Bother, decisionValidInv, CoordInv2
1634 <8>3. QED
1635 BY <8>1, <8>2
1636 <3>4. CASE BaskRM(self)
1637 <4>1. qrm = qrm \union { self }
1638 BY <3>4, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
1639 decisionValidInv
1640 <4>2. qrm' = qrm
1641 BY <4>1, <3>4 DEF BaskRM
1642 <4>3. QED
1643 BY <4>2, <3>4, <2>0, <1>4, <2>4, SetsTheorem
1644 DEF TypeOk, BaskRM, decisionValidInv, CoordInv2
1645 <3>5. CASE BlockAsset(self)
1646 <8>1. CASE (ProofPublish = TRUE /\
1647 assets[AofS(self)] = "OwS")
1648 <9>1. assets' =
1649 [assets EXCEPT ![AofS(self)] = "locked"]
1650 BY <8>1, <3>5, <2>0, <1>4, <2>4, SetsTheorem
1651 DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
1652 <9>2. QED
1653 BY <9>1, <8>1, <3>5, <2>0, <1>4, <2>4, SetsTheorem
1654 DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
1655 <8>2. CASE ~(ProofPublish = TRUE /\
1656 assets[AofS(self)] = "OwS")
1657 BY <8>2, <3>5, <2>0, <1>4, <2>4, SetsTheorem
1658 DEF TypeOk, BlockAsset, decisionValidInv, CoordInv2
1659 <8>3. QED
1660 BY <8>1, <8>2
1661 <3>6. CASE BSaskRF(self)
1662 BY <3>6, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
1663 BSaskRF, decisionValidInv, CoordInv2
1664 <3>7. CASE BrecoveringAsset(self)
1665 BY <3>7, <2>0, <1>4, <2>4, SetsTheorem DEF TypeOk,
1666 BrecoveringAsset, decisionValidInv, CoordInv2
1667 <3>8. QED
1668 BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
1669 DEF BSource
1670 <2>5. CASE \E self \in CRecipients: Recipient(self)
1671 <3> SUFFICES ASSUME NEW self \in CRecipients,
1672 Recipient(self)
1673 PROVE CoordInv2'
1674 BY <2>5
1675 <3>1. CASE init_rcp(self)
1676 BY <3>1, <1>4, <2>0, <2>5, SetsTheorem DEF TypeOk,
1677 init_rcp, decisionValidInv, CoordInv2
1678 <3>2. CASE waitForD_rcp(self)
1679 BY <3>2, <1>4, <2>0, <2>5, SetsTheorem DEF TypeOk,
1680 waitForD_rcp, decisionValidInv, CoordInv2
1681 <3>5. QED

```

1684	BY <2>5, <3>1, <3>2 DEF Recipient	1753	<3>4. CASE decisionValid
1685		1754	BY <1>6, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionValid
1686	<2>6. CASE \E self \in BRecipients: BRecipient(self)	1755	<3>7. QED
1687	<3> SUFFICES ASSUME NEW self \in BRecipients,	1756	BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator
1688	BRecipient(self)	1757	
1689	PROVE CoordInv2'	1758	<2>3. CASE \E self \in CSources: Source(self)
1690	BY <2>6	1759	<3> SUFFICES ASSUME NEW self \in CSources,
1691	<3>1. CASE init_brcp(self)	1760	Source(self)
1692	BY <3>1, <1>4, <2>0, <2>6, SetsTheorem DEF TypeOk,	1761	PROVE CoordInv2'
1693	init_brcp, decisionValidInv, CoordInv2	1762	BY <2>3
1694	<3>2. CASE BRaskRF(self)	1763	<3>1. CASE init_src(self)
1695	BY <3>2, <1>4, <2>0, <2>6, SetsTheorem DEF TypeOk,	1764	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
1696	BRaskRF, decisionValidInv, CoordInv2	1765	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1697	<3>3. CASE BRretrievingAsset(self)	1766	init_src, decisionAbortInv, CoordInv2
1698	BY <3>3, <1>4, <2>0, <2>6, SetsTheorem DEF TypeOk,	1767	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
1699	BRretrievingAsset, decisionValidInv, CoordInv2	1768	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1700	<3>4. CASE BRdirectToS(self)	1769	init_src, decisionAbortInv, CoordInv2
1701	BY <3>4, <1>4, <2>0, <2>6, SetsTheorem DEF TypeOk,	1770	<4>3. (pc[CoordinatorID] = "decisionValid" =>
1702	BRdirectToS, decisionValidInv, CoordInv2	1771	decisionValidInv)'
1703	<3>5. CASE BRother(self)	1772	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1704	BY <3>5, <1>4, <2>0, <2>6, SetsTheorem DEF TypeOk,	1773	init_src, decisionAbortInv, CoordInv2
1705	BRother, decisionValidInv, CoordInv2	1774	<4>4. (pc[CoordinatorID] = "decisionAbort" =>
1706	<3>6. QED	1775	decisionAbortInv)'
1707	BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient	1776	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1708		1777	init_src, decisionAbortInv, CoordInv2
1709	<2>7. CASE Terminating	1778	<4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM"
1710	BY <1>4, <2>0, <2>7, SetsTheorem DEF TypeOk,	1779	=> okRMInv)'
1711	Terminating, CoordInv2	1780	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1712	<2>8. QED	1781	init_src, decisionAbortInv, CoordInv2
1713	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next	1782	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF"
1714		1783	=> okRFInv)'
1715	<1>6. CASE pc[CoordinatorID] = "decisionAbort"	1784	BY <3>1, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1716	<2>0. decisionAbortInv	1785	init_src, decisionAbortInv, CoordInv2
1717	BY <1>6 DEF CoordInv2	1786	<4>7. QED
1718	<2>1. CASE Publisher	1787	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1719	BY <1>6, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,	1788	
1720	init_p, decisionAbortInv, CoordInv2	1789	<3>2. CASE lock(self)
1721	<2>2. CASE Coordinator	1790	<4>0. assets' = [assets EXCEPT ![AofS(self)] = "locked"]
1722	<3>1. CASE init_c	1791	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1723	BY <1>6, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c	1792	lock, decisionAbortInv
1724	<3>2. CASE decision	1793	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
1725	BY <1>6, <3>2, <2>2, SetsTheorem DEF TypeOk, decision	1794	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1726		1795	lock, decisionAbortInv
1727	<3>3. CASE decisionAbort	1796	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
1728	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'	1797	BY <3>2, <2>0, <1>6, <2>3, <4>0, SetsTheorem
1729	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1798	DEF TypeOk, lock, decisionAbortInv
1730	decisionAbort	1799	<4>3. (pc[CoordinatorID] = "decisionValid" =>
1731	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'	1800	decisionValidInv)'
1732	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1801	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1733	decisionAbort	1802	lock, decisionAbortInv, decisionAbortInv
1734	<4>3. (pc[CoordinatorID] = "decisionValid" =>	1803	<4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM"
1735	decisionValidInv)'	1804	=> okRMInv)'
1736	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1805	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1737	decisionAbort, decisionAbortInv	1806	lock, decisionAbortInv
1738	<4>4. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM"	1807	<4>5. (pc[CoordinatorID] = "decisionAbort" =>
1739	=> okRMInv)'	1808	decisionAbortInv)'
1740	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1809	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1741	decisionAbort, decisionAbortInv, okRMInv	1810	lock, decisionAbortInv, decisionAbortInv
1742	<4>5. (pc[CoordinatorID] = "decisionAbort" =>	1811	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF"
1743	decisionAbortInv)'	1812	=> okRFInv)'
1744	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1813	BY <3>2, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1745	decisionAbort, decisionAbortInv	1814	lock, decisionAbortInv
1746	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF"	1815	<4>7. QED
1747	=> okRFInv)'	1816	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1748	BY <2>0, <1>6, <3>3, <2>2, SetsTheorem DEF TypeOk,	1817	
1749	decisionAbort, okRFInv, decisionAbortInv	1818	<3>3. CASE published(self)
1750		1819	BY <3>3, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,
1751	<4>7. QED	1820	published, decisionAbortInv, CoordInv2
1752	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2	1821	<3>4. CASE waitForD(self)

1822	<4>0. pc'[CoordinatorID] = pc[CoordinatorID] /\	1891	BY <8>1, <3>5, <2>0, <1>6, <2>4, SetsTheorem
1823	pc'[PublisherID] = pc[PublisherID]	1892	DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
1824	BY <3>4, <2>0, <1>6, <2>3, SetsTheorem DEF TypeOk,	1893	<9>2. QED
1825	waitForD	1894	BY <9>1, <8>1, <3>5, <2>0, <1>6, <2>4, SetsTheorem
1826	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'	1895	DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
1827	BY <4>0, <1>6	1896	<8>2. CASE ~(ProofPublish = TRUE /\
1828	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'	1897	assets[AofS(self)] = "OwS")
1829	BY <4>0, <1>6	1898	BY <8>2, <3>5, <2>0, <1>6, <2>4, SetsTheorem
1830	<4>3. (pc[CoordinatorID] = "decisionValid" =>	1899	DEF TypeOk, BlockAsset, decisionAbortInv, CoordInv2
1831	decisionValidInv)'	1900	<8>3. QED
1832	BY <4>0, <1>6	1901	BY <8>1, <8>2
1833	<4>4. ((pc[CoordinatorID] = "Done" /\ coordState="okRM")	1902	<3>6. CASE BSaskRF(self)
1834	=> okRMInv)'	1903	BY <3>6, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
1835	BY <4>0, <1>6	1904	BSaskRF, decisionAbortInv, CoordInv2
1836	<4>5. (pc[CoordinatorID] = "decisionAbort" =>	1905	<3>7. CASE BrecoveringAsset(self)
1837	decisionAbortInv)'	1906	BY <3>7, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,
1838	BY <4>0, <3>4, <2>0, <1>6, <2>3, SetsTheorem	1907	BrecoveringAsset, decisionAbortInv, CoordInv2
1839	DEF TypeOk, waitForD, decisionAbortInv	1908	<3>8. QED
1840	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState="okRF")	1909	BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
1841	=> okRFInv)'	1910	DEF BSource
1842	BY <4>0, <1>6	1911	
1843	<4>7. QED	1912	<2>5. CASE \E self \in CRecipients: Recipient(self)
1844	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2	1913	<3> SUFFICES ASSUME NEW self \in CRecipients,
1845	<3>7. QED	1914	Recipient(self)
1846	BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source	1915	PROVE CoordInv2'
1847		1916	BY <2>5
1848	<2>4. CASE \E self \in BSources: BSource(self)	1917	<3>1. CASE init_rcp(self)
1849	<3> SUFFICES ASSUME NEW self \in BSources,	1918	BY <3>1, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1850	BSource(self)	1919	init_rcp, decisionAbortInv, CoordInv2
1851	PROVE CoordInv2'	1920	<3>2. CASE waitForD_rcp(self)
1852	BY <2>4	1921	<4>1. (pc[CoordinatorID] = "init_c" => init_cInv)'
1853	<3>1. CASE init_bsrc(self)	1922	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1854	BY <3>1, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,	1923	waitForD_rcp, decisionAbortInv, CoordInv2
1855	init_bsrc, decisionAbortInv, CoordInv2	1924	<4>2. (pc[CoordinatorID] = "decision" => decisionInv)'
1856	<3>2. CASE BdirectToR(self)	1925	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1857	<8>1. CASE assets[AofS(self)] = "OwS"	1926	waitForD_rcp, decisionAbortInv, CoordInv2
1858	<9>1. assets' = [assets EXCEPT ![AofS(self)] = "OwR"]	1927	<4>3. (pc[CoordinatorID] = "decisionValid" =>
1859	BY <8>1, <3>2, <2>0, <1>6, <2>4, SetsTheorem	1928	decisionValidInv)'
1860	DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2	1929	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1861	<9>2. QED	1930	waitForD_rcp, decisionAbortInv, CoordInv2
1862	BY <9>1, <8>1, <3>2, <2>0, <1>6, <2>4, SetsTheorem	1931	<4>4. (pc[CoordinatorID] = "decisionAbort" =>
1863	DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2	1932	decisionAbortInv)'
1864	<8>2. CASE assets[AofS(self)] # "OwS"	1933	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1865	BY <8>2, <3>2, <2>0, <1>6, <2>4, SetsTheorem	1934	waitForD_rcp, decisionAbortInv, CoordInv2
1866	DEF TypeOk, BdirectToR, decisionAbortInv, CoordInv2	1935	<4>5. ((pc[CoordinatorID] = "Done" /\ coordState = "okRM"
1867	<8>3. QED	1936	=> okRMInv)'
1868	BY <8>1, <8>2	1937	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1869	<3>3. CASE Bother(self)	1938	waitForD_rcp, decisionAbortInv, CoordInv2
1870	<8>1. CASE assets[AofS(self)] = "OwS"	1939	<4>6. ((pc[CoordinatorID] = "Done" /\ coordState = "okRF"
1871	<9>1. assets' = [assets EXCEPT ![AofS(self)] = "other"]	1940	=> okRFInv)'
1872	BY <8>1, <3>3, <2>0, <1>6, <2>4, SetsTheorem	1941	BY <3>2, <1>6, <2>0, <2>5, SetsTheorem DEF TypeOk,
1873	DEF TypeOk, Bother, decisionAbortInv, CoordInv2	1942	waitForD_rcp, decisionAbortInv, CoordInv2
1874	<9>2. QED	1943	<4>7. QED
1875	BY <9>1, <8>1, <3>3, <2>0, <1>6, <2>4, SetsTheorem	1944	BY <4>1, <4>2, <4>3, <4>4, <4>5, <4>6 DEF CoordInv2
1876	DEF TypeOk, Bother, decisionAbortInv, CoordInv2	1945	
1877	<8>2. CASE assets[AofS(self)] # "OwS"	1946	<3>5. QED
1878	BY <8>2, <3>3, <2>0, <1>6, <2>4, SetsTheorem	1947	BY <2>5, <3>1, <3>2 DEF Recipient
1879	DEF TypeOk, Bother, decisionAbortInv, CoordInv2	1948	
1880	<8>3. QED	1949	<2>6. CASE \E self \in BRecipients: BRecipient(self)
1881	BY <8>1, <8>2	1950	<3> SUFFICES ASSUME NEW self \in BRecipients,
1882	<3>4. CASE BaskRM(self)	1951	BRecipient(self)
1883	BY <3>4, <2>0, <1>6, <2>4, SetsTheorem DEF TypeOk,	1952	PROVE CoordInv2'
1884	BaskRM, decisionAbortInv, CoordInv2	1953	BY <2>6
1885		1954	<3>1. CASE init_brcp(self)
1886	<3>5. CASE BlockAsset(self)	1955	BY <3>1, <1>6, <2>0, <2>6, SetsTheorem DEF TypeOk,
1887	<8>1. CASE (ProofPublish = TRUE /\	1956	init_brcp, decisionAbortInv, CoordInv2
1888	assets[AofS(self)] = "OwS")	1957	<3>2. CASE BaskRF(self)
1889	<9>1. assets' =	1958	BY <3>2, <1>6, <2>0, <2>6, SetsTheorem DEF TypeOk,
1890	[assets EXCEPT ![AofS(self)] = "locked"]	1959	BaskRF, decisionAbortInv, CoordInv2

1960	<3>3. CASE BRretrievingAsset(self)	2029	BdirectToR, okRMInv, CoordInv2
1961	BY <3>3, <1>6, <2>0, <2>6, SetsTheorem DEF TypeOk,	2030	<3>3. CASE Bother(self)
1962	BRretrievingAsset, decisionAbortInv, CoordInv2	2031	BY <3>3, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
1963	<3>4. CASE BRdirectToS(self)	2032	Bother, okRMInv, CoordInv2
1964	BY <3>4, <1>6, <2>0, <2>6, SetsTheorem DEF TypeOk,	2033	<3>4. CASE BaskRM(self)
1965	BRdirectToS, decisionAbortInv, CoordInv2	2034	BY <3>4, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
1966	<3>5. CASE BROther(self)	2035	BaskRM, okRMInv, CoordInv2
1967	BY <3>5, <1>6, <2>0, <2>6, SetsTheorem DEF TypeOk,	2036	
1968	BROther, decisionAbortInv, CoordInv2	2037	<3>5. CASE BlockAsset(self)
1969	<3>6. QED	2038	BY <3>5, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
1970	BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient	2039	BlockAsset, okRMInv, CoordInv2
1971		2040	<3>6. CASE BSaskRF(self)
1972	<2>7. CASE Terminating	2041	BY <3>6, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
1973	BY <1>6, <2>0, <2>7, SetsTheorem DEF TypeOk,	2042	BSaskRF, okRMInv, CoordInv2
1974	Terminating, CoordInv2	2043	<3>7. CASE BrecoveringAsset(self)
1975	<2>8. QED	2044	BY <3>7, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,
1976	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next	2045	BrecoveringAsset, okRMInv, CoordInv2
1977		2046	<3>8. QED
1978	<1>3. CASE pc[CoordinatorID] = "Done" /\ coordState = "okRM"	2047	BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7
1979	<2>0. okRMInv	2048	DEF BSource
1980	BY <1>3 DEF CoordInv2	2049	
1981	<2>1. CASE Publisher	2050	<2>5. CASE \E self \in CRecipients: Recipient(self)
1982	BY <1>3, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,	2051	<3> SUFFICES ASSUME NEW self \in CRecipients,
1983	init_p, okRMInv, CoordInv2	2052	Recipient(self)
1984	<2>2. CASE Coordinator	2053	PROVE CoordInv2'
1985	<3>1. CASE init_c	2054	BY <2>5
1986	BY <1>3, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c	2055	<3>1. CASE init_rcp(self)
1987	<3>2. CASE decision	2056	BY <3>1, <1>3, <2>0, <2>5, SetsTheorem DEF TypeOk,
1988	BY <1>3, <3>2, <2>2, SetsTheorem DEF TypeOk, decision,	2057	init_rcp, okRMInv, CoordInv2
1989	CoordInv2	2058	<3>2. CASE waitForD_rcp(self)
1990	<3>3. CASE decisionValid	2059	BY <3>2, <1>3, <2>0, <2>5, SetsTheorem DEF TypeOk,
1991	BY <1>3, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid	2060	waitForD_rcp, okRMInv, CoordInv2
1992	<3>4. CASE decisionAbort	2061	<3>5. QED
1993	BY <1>3, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort	2062	BY <2>5, <3>1, <3>2 DEF Recipient
1994	<3>7. QED	2063	
1995	BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator	2064	<2>6. CASE \E self \in BRecipients: BRecipient(self)
1996		2065	<3> SUFFICES ASSUME NEW self \in BRecipients,
1997	<2>3. CASE \E self \in CSources: Source(self)	2066	BRecipient(self)
1998	<3> SUFFICES ASSUME NEW self \in CSources,	2067	PROVE CoordInv2'
1999	Source(self)	2068	BY <2>6
2000	PROVE CoordInv2'	2069	<3>1. CASE init_brcp(self)
2001	BY <2>3	2070	BY <3>1, <1>3, <2>0, <2>6, SetsTheorem DEF TypeOk,
2002	<3>1. CASE init_src(self)	2071	init_brcp, okRMInv, CoordInv2
2003	BY <3>1, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,	2072	<3>2. CASE BRaskRF(self)
2004	init_src, okRMInv, CoordInv2	2073	BY <3>2, <1>3, <2>0, <2>6, SetsTheorem DEF TypeOk,
2005	<3>2. CASE lock(self)	2074	BRaskRF, okRMInv, CoordInv2
2006	BY <3>2, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,	2075	<3>3. CASE BRretrievingAsset(self)
2007	lock, okRMInv, CoordInv2	2076	BY <3>3, <1>3, <2>0, <2>6, SetsTheorem DEF TypeOk,
2008		2077	BRretrievingAsset, okRMInv, CoordInv2
2009	<3>3. CASE published(self)	2078	<3>4. CASE BRdirectToS(self)
2010	BY <3>3, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,	2079	BY <3>4, <1>3, <2>0, <2>6, SetsTheorem DEF TypeOk,
2011	published, okRMInv, CoordInv2	2080	BRdirectToS, okRMInv, CoordInv2
2012	<3>4. CASE waitForD(self)	2081	<3>5. CASE BROther(self)
2013	BY <3>4, <2>0, <1>3, <2>3, SetsTheorem DEF TypeOk,	2082	BY <3>5, <1>3, <2>0, <2>6, SetsTheorem DEF TypeOk,
2014	waitForD, okRMInv, CoordInv2	2083	BROther, okRMInv, CoordInv2
2015		2084	<3>6. QED
2016	<3>7. QED	2085	BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient
2017	BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source	2086	
2018		2087	<2>7. CASE Terminating
2019	<2>4. CASE \E self \in BSources: BSource(self)	2088	BY <1>3, <2>0, <2>7, SetsTheorem DEF TypeOk,
2020	<3> SUFFICES ASSUME NEW self \in BSources,	2089	Terminating, vars, CoordInv2, init_cInv, decisionInv,
2021	BSource(self)	2090	okRMInv, decisionValidInv
2022	PROVE CoordInv2'	2091	<2>8. QED
2023	BY <2>4	2092	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next
2024	<3>1. CASE init_bsrc(self)	2093	
2025	BY <3>1, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,	2094	<1>5. CASE (pc[CoordinatorID] = "Done" /\ coordState = "okRF")
2026	init_bsrc, okRMInv, CoordInv2	2095	<2>0. okRFInv
2027	<3>2. CASE BdirectToR(self)	2096	BY <1>5 DEF CoordInv2
2028	BY <3>2, <2>0, <1>3, <2>4, SetsTheorem DEF TypeOk,	2097	<2>1. CASE Publisher

2098	BY <1>5, <2>0, <2>1, SetsTheorem DEF TypeOk, Publisher,	2167	Recipient(self)
2099	init_p, okRFInv, CoordInv2	2168	PROVE CoordInv2'
2100	<2>2. CASE Coordinator	2169	BY <2>5
2101	<3>1. CASE init_c	2170	<3>1. CASE init_rcp(self)
2102	BY <1>5, <3>1, <2>2, SetsTheorem DEF TypeOk, init_c	2171	BY <3>1, <1>5, <2>0, <2>5, SetsTheorem DEF TypeOk,
2103	<3>2. CASE decision	2172	init_rcp, okRFInv, CoordInv2
2104	BY <1>5, <3>2, <2>2, SetsTheorem DEF TypeOk, decision,	2173	<3>2. CASE waitForD_rcp(self)
2105	CoordInv2	2174	BY <3>2, <1>5, <2>0, <2>5, SetsTheorem DEF TypeOk,
2106	<3>3. CASE decisionValid	2175	waitForD_rcp, okRFInv, CoordInv2
2107	BY <1>5, <3>3, <2>2, SetsTheorem DEF TypeOk, decisionValid	2176	<3>5. QED
2108	<3>4. CASE decisionAbort	2177	BY <2>5, <3>1, <3>2 DEF Recipient
2109	BY <1>5, <3>4, <2>2, SetsTheorem DEF TypeOk, decisionAbort	2178	
2110	<3>7. QED	2179	<2>6. CASE \E self \in BRecipients: BRecipient(self)
2111	BY <2>2, <3>1, <3>2, <3>3, <3>4 DEF Coordinator	2180	<3> SUFFICES ASSUME NEW self \in BRecipients,
2112		2181	BRecipient(self)
2113	<2>3. CASE \E self \in CSources: Source(self)	2182	PROVE CoordInv2'
2114	<3> SUFFICES ASSUME NEW self \in CSources,	2183	BY <2>6
2115	Source(self)	2184	<3>1. CASE init_brpc(self)
2116	PROVE CoordInv2'	2185	BY <3>1, <1>5, <2>0, <2>6, SetsTheorem DEF TypeOk,
2117	BY <2>3	2186	init_brpc, okRFInv, CoordInv2
2118	<3>1. CASE init_src(self)	2187	<3>2. CASE BMaskRF(self)
2119	BY <3>1, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,	2188	BY <3>2, <1>5, <2>0, <2>6, SetsTheorem DEF TypeOk,
2120	init_src, okRFInv, CoordInv2	2189	BMaskRF, okRFInv, CoordInv2
2121	<3>2. CASE lock(self)	2190	<3>3. CASE BRretrievingAsset(self)
2122	BY <3>2, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,	2191	BY <3>3, <1>5, <2>0, <2>6, SetsTheorem DEF TypeOk,
2123	lock, okRFInv, CoordInv2	2192	BRretrievingAsset, okRFInv, CoordInv2
2124	<3>3. CASE published(self)	2193	<3>4. CASE BRdirectToS(self)
2125	BY <3>3, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,	2194	BY <3>4, <1>5, <2>0, <2>6, SetsTheorem DEF TypeOk,
2126	published, okRFInv, CoordInv2	2195	BRdirectToS, okRFInv, CoordInv2
2127	<3>4. CASE waitForD(self)	2196	<3>5. CASE BROther(self)
2128	BY <3>4, <2>0, <1>5, <2>3, SetsTheorem DEF TypeOk,	2197	BY <3>5, <1>5, <2>0, <2>6, SetsTheorem DEF TypeOk,
2129	waitForD, okRFInv, CoordInv2	2198	BROther, okRFInv, CoordInv2
2130		2199	<3>6. QED
2131	<3>7. QED	2200	BY <2>6, <3>1, <3>2, <3>3, <3>4, <3>5 DEF BRecipient
2132	BY <2>3, <3>1, <3>2, <3>3, <3>4 DEF Source	2201	
2133		2202	<2>7. CASE Terminating
2134	<2>4. CASE \E self \in BSources: BSource(self)	2203	BY <1>5, <2>0, <2>7, SetsTheorem DEF TypeOk,
2135	<3> SUFFICES ASSUME NEW self \in BSources,	2204	Terminating, vars, CoordInv2, init_cInv, decisionInv,
2136	BSource(self)	2205	okRFInv, decisionValidInv
2137	PROVE CoordInv2'	2206	<2>8. QED
2138	BY <2>4	2207	BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF Next
2139	<3>1. CASE init_bsrc(self)	2208	
2140	BY <3>1, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2209	<1>7. QED
2141	init_bsrc, okRFInv, CoordInv2	2210	BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6 DEF TypeOk, CStates
2142	<3>2. CASE BdirectToR(self)	2211	
2143	BY <3>2, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2212	
2144	BdirectToR, okRFInv, CoordInv2	2213	
2145	<3>3. CASE Bother(self)	2214	THEOREM InvInvariant ==
2146	BY <3>3, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2215	ASSUME Inv, Next
2147	Bother, okRFInv, CoordInv2	2216	PROVE Inv'
2148	<3>4. CASE BaskRM(self)	2217	BY TypeOkInvariant, CoordInvariant DEF Inv, TypeOk, CoordInv2, Next
2149	BY <3>4, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2218	
2150	BaskRM, okRFInv, CoordInv2	2219	
2151		2220	THEOREM InvImpliesConsistency ==
2152	<3>5. CASE BblockAsset(self)	2221	ASSUME TypeOk /\ CoordInv2
2153	BY <3>5, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2222	PROVE Consistency
2154	BblockAsset, okRFInv, CoordInv2	2223	<1> USE DEF Finish, AvailableS, AvailableR
2155	<3>6. CASE BSaskRF(self)	2224	<1>1. CASE pc[CoordinatorID] = "init_c"
2156	BY <3>6, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2225	BY <1>1, SetsTheorem DEF CoordInv2, init_cInv, Consistency
2157	BSaskRF, okRFInv, CoordInv2	2226	<1>2. CASE pc[CoordinatorID] = "decision"
2158	<3>7. CASE BrecoveringAsset(self)	2227	<2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "0wS"}
2159	BY <3>7, <2>0, <1>5, <2>4, SetsTheorem DEF TypeOk,	2228	BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
2160	BrecoveringAsset, okRFInv, CoordInv2	2229	<2>2. \A s \in CSources: assets[AofS(s)] \in {"locked", "0wS"}
2161	<3>8. QED	2230	BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
2162	BY <2>4, <3>1, <3>2, <3>3, <3>4, <3>5, <3>6, <3>7	2231	<2>3. \A s \in CSources: pc[s] = "Done" =>
2163	DEF BSource	2232	assets[AofS(s)] = "0wS"
2164		2233	BY <1>2, SetsTheorem DEF CoordInv2, decisionInv, Consistency
2165	<2>5. CASE \E self \in CRecipients: Recipient(self)	2234	<2>4. QED
2166	<3> SUFFICES ASSUME NEW self \in CRecipients,	2235	BY <2>1, <2>2, <2>3 DEF Consistency

```

2236 <1>3. CASE pc[CoordinatorID] = "decisionValid" 2305
2237 <2>1. \A a \in Assets: assets[a] = "locked" 2306
2238 BY <1>3, SetsTheorem DEF CoordInv2, decisionValidInv,
2239 Consistency
2240 <2>2. \A a \in AssetsFromCS: assets[a] = "locked"
2241 BY <1>3, SetsTheorem DEF CoordInv2, decisionValidInv,
2242 Consistency
2243 <2>3. \A s \in CSources: pc[s] # "Done"
2244 BY <1>3, SetsTheorem DEF CoordInv2, decisionValidInv,
2245 Consistency
2246 <2>4. QED
2247 BY <2>1, <2>2, <2>3 DEF Consistency
2248 <1>4. CASE pc[CoordinatorID] = "decisionAbort"
2249 <2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "OwS"}
2250 BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
2251 Consistency
2252 <2>2. \A s \in CSources: assets[AofS(s)] \in
2253 {"locked", "OwS"}
2254 BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
2255 Consistency
2256 <2>3. \A s \in CSources: pc[s] = "Done" =>
2257 assets[AofS(s)] = "OwS"
2258 BY <1>4, SetsTheorem DEF CoordInv2, decisionAbortInv,
2259 Consistency
2260 <2>4 QED
2261 BY <2>1, <2>2, <2>3 DEF Consistency
2262 <1>5. CASE (pc[CoordinatorID] = "Done" /\ coordState = "okRM")
2263 <2>1. ProofOkRM = TRUE => \A a \in AssetsForCR: assets[a]
2264 \in {"locked", "OwR"}
2265 BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
2266 <2>2. \A r \in CRecipients : pc[r] = "Done" =>
2267 assets[AofR(r)] \in {"OwR", "locked"}
2268 BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
2269 <2>3. ProofOkRM = TRUE
2270 BY <1>5, SetsTheorem DEF CoordInv2, okRMInv, Consistency
2271 <2>4 QED
2272 BY <2>1, <2>2, <2>3 DEF Consistency
2273 <1>6. CASE (pc[CoordinatorID] = "Done" /\ coordState = "okRF")
2274 <2>1. \A a \in AssetsFromCS: assets[a] \in {"locked", "OwS"}
2275 BY <1>6, SetsTheorem DEF CoordInv2, okRFInv, Consistency
2276 <2>2. \A s \in CSources: pc[s] = "Done" =>
2277 assets[AofS(s)] = "OwS"
2278 BY <1>6, SetsTheorem DEF CoordInv2, okRFInv, Consistency
2279 <2>3 QED
2280 BY <2>1, <2>2 DEF Consistency
2281 <1>7. QED
2282 BY <1>1, <1>2, <1>3, <1>4, <1>5, <1>6 DEF TypeOk
2283
2284 THEOREM Safety2 == Spec => [] Consistency
2285 <1>1. Init => Inv
2286 BY InitImpliesInv, SMT DEF Inv
2287 <1>2. Inv /\ [Next]_vars => Inv'
2288 <2> SUFFICES ASSUME Inv,
2289 [Next]_vars
2290 PROVE Inv'
2291 OBVIOUS
2292 <2>1. CASE Next
2293 BY <2>1, SMTT(60), InvInvariant DEF vars
2294 <2>2. CASE UNCHANGED vars
2295 BY <2>2, SMTT(60) DEF vars, Inv, TypeOk, CoordInv2, okRFInv,
2296 okRMInv, decisionAbortInv, decisionValidInv, init_cInv,
2297 decisionInv
2298 <2>3. QED
2299 BY <2>1, <2>2
2300
2301 <1>3. Inv => Consistency
2302 BY SMT, InvImpliesConsistency DEFS Consistency, Inv
2303 <1>4. QED
2304 BY ONLY <1>1, <1>2, <1>3, PTL DEF Spec

```

