Formalisation et vérification des systèmes blockchain

Zeinab Nehaï

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

Université Paris Cité

list cea tech

THÈSE DE DOCTORAT EN INFORMATIQUE UNIVERSITÉ PARIS CITÉ

École Doctorale Sciences Mathématiques de Paris Centre (ED 386)
Institut de Recherche en Informatique Fondamentale

Formalisation et Verification des Systèmes Blockchain

Formalisation and Verification of Blockchain Systems

Présentée et soutenue publiquement par

Zeinab Nehaï

Dirigée par

Hugues Fauconnier

Pour l'obtention du grade de docteur

Le 12 juillet 2022

Composition du jury :

Pierre Fraigniaud	Directeur de recherche - Université Paris Cité	*Président du jury*
Emmanuelle Anceaume	Directrice de recherche - CNRS, IRISA	*Rapportrice*
Yann Régis-Gianas	Chercheur HDR - Nomadic labs	*Rapporteur*
Cezara Drăgoi	Chargée de recherche - ENS, INRIA Paris	*Examinatrice*
Mihaela Sighireanu	Professeur - ENS Paris-Saclay	*Examinatrice*
François Bobot	Ingénieur-chercheur - CEA List	*Co-Encadrant de thèse*
Hugues Fauconnier	Professeur - Université Paris Cité	*Directeur de thèse*
Sara Tucci-Piergiovanni	Cheffe de Laboratoire - CEA List	*invitée*

Résumé

Une *blockchain* est un système distribué qui permet de stocker des données ne pouvant être ni modifiées ni supprimées. *Bitcoin* est la première application ayant implémenté la technologie blockchain avec succès. Le but de Bitcoin est de pouvoir échanger de la monnaie virtuelle (des *bitcoins*) sans passer par un intermédiaire de confiance, par exemple une banque. Les applications blockchain sont très complexes, car elles sont constituées de plusieurs programmes, souvent tous liés les uns aux autres. Une caractéristique qui a suscité un fort intérêt pour les blockchains est la possibilité d'écrire des *smart contracts*. Ces derniers sont des programmes séquentiels dans lesquels des règles de transaction peuvent être définies. Ce type de programme a révolutionné l'utilisation de la blockchain, notamment dans le développement des applications décentralisées. Cependant, l'utilisation croissante de la blockchain a fait émerger les limites de cette technologie. En effet, l'augmentation du nombre de blockchains, a fait naître auprès des utilisateurs l'envie d'échanger avec d'autres utilisateurs n'étant pas sur la même blockchain qu'eux. Pour répondre à ce cas d'utilisation, les applications appelées *cross-chain swap* ont été développées. Leur but est d'assurer l'échange d'actifs entre différents utilisateurs se trouvant sur différentes blockchains. Les actifs peuvent être des crypto-monnaies ou des actifs physiques qu'on a virtualisé. Ces applications sont souvent basées sur l'utilisation des *smart contracts* pour établir les règles de transfert d'actifs. Bien qu'étant une technologie qui se popularise grandement, la blockchain souffre d'un manque de formalisme de ses systèmes. Par exemple, le langage le plus répandu qui permet d'écrire des *smart contracts* est *Solidity*. C'est un langage avec une sémantique non-formelle, rendant les *smart contracts* rédigés dans ce langage, vulnérable aux attaques. De plus, un autre aspect dont souffre la blockchain est la présence de participants malveillants, communément appelés participants *Byzantins*. Ces participants ont un comportement aléatoire, et sont susceptibles de ne pas suivre les règles du système. Ainsi, les systèmes, comme les *cross-chain swap*, ont nécessairement besoin d'employer des moyens pour assurer leur bon fonctionnement malgré la présence de *Byzantins*.

Un moyen d'y parvenir est l'utilisation des *méthodes formelles*. Ces techniques permettant de raisonner rigoureusement, à l'aide de logique mathématique, sur un programme informatique, afin de démontrer leur validité par rapport à une certaine spécification. *La vérification de modèles* et *la démonstration automatique de théorèmes* sont des exemples de techniques de vérification. La vérification de modèles est une méthode d'abstraction de système pour en faire un modèle à état. L'approche consiste à vérifier que le modèle satisfait bien la spécification donnée. Quant à la démonstration automatique de théorèmes, la méthode se base sur la formulation de théorème mathématique, du raisonnement, et de la logique, pour prouver un ensemble de propositions.

Dans cette thèse, nous proposons d'appliquer ces méthodes de vérification aux systèmes que nous avons cités, à savoir les *smart contracts* et les applications *cross-chain swap*. Dans un premier temps, nous proposons le langage *WhyML*, dédié à la vérification formelle de programme, comme langage d'écriture de *smart contract*. Le langage *WhyML*, se base sur de la logique mathématique pour prouver l'exactitude du programme. L'approche est de définir un *smart contract* sous forme de formules mathématiques et ensuite d'appliquer des outils qui permettent de prouver que les formules sont vraies. En appliquant cette approche, on s'assure de la correction des contracts avant leur stockage dans la blockchain. Les résultats obtenus à l'issue de cette première étude ont montré que *WhyML* convenait comme langage d'écriture de *smart contracts*, permettant ainsi d'avoir des programmes corrects par construction. Dans un second temps, nous appliquons des méthodes de formalisme à des applications *cross-chain swap*. La première étape de cette formalisation est de

définir une spécification formelle du problème des *cross-chain swaps*. L'approche consiste à définir les propriétés qui caractérisent la spécification du problème, par exemple, la spécification se doit d'être résiliente aux participants *Byzantins*. Dans les systèmes distribués, on caractérise deux types de propriétés : les propriétés de *sûreté*, qui garantissent que "*rien de mauvais n'arrivera*", et les propriétés de *vivacité* qui garantissent que "*quelque chose de bien finira par arriver*". La vivacité est une propriété qui est exprimée en fonction du temps, ce qui n'est pas le cas de la sûreté. Ainsi, le problème du *cross-chain swap* défini une propriété de sûreté et deux propriétés de vivacité.

La seconde étape est la construction d'un algorithme qui doit satisfaire la spécification du *cross-chain swap*. Une fois ces deux étapes accomplies, nous appliquons des méthodes formelles à l'algorithme pour prouver sa correction. Pour cette approche, nous avons modélisé notre algorithme en utilisant TLA$^+$ qui est un langage spécifique pour modéliser des systèmes distribués et concurrents. Nous avons ensuite vérifié si le modèle obtenu satisfait bien les propriétés du problème étudié. Nous avons employé deux méthodes différentes en fonction des types de propriétés. Pour cette vérification, nous avons appliqué la méthode par démonstration de théorème pour la preuve de la sûreté, et la méthode de vérification de modèles pour vérifier les propriétés de vivacité. Les résultats obtenus ont montré que l'algorithme du *cross-chain swap* que nous avons construit satisfaisait bien les propriétés de la spécification du *cross-chain swap*.

Mots clés : méthodes formelles, vérification de modèles, démonstration automatique de théorèmes, systèmes distribués,blockchain, smart contracts, cross-chain swap, why3, tla+.

Abstract

A *blockchain* is a distributed system for storing data that cannot be changed or deleted. *Bitcoin* is the first application to implement blockchain technology successfully. The purpose of Bitcoin is to exchange virtual money (*bitcoins*) without going through a trusted intermediary, such as a bank. Blockchain applications are very complex, as they consist of several programs that are often linked to each other. One feature that has attracted much interest in blockchains is the ability to write *smart contracts*. These are sequential programs in which transaction rules can be defined. This type of program has revolutionised the use of blockchain, particularly in developing decentralised applications. However, the increasing use of blockchain has brought to light the limits of this technology. Indeed, the increase in the number of blockchains has given rise to the desire of users to exchange with other users who are not on the same blockchain as them. To meet this use case, applications called *cross-chain swap* have been developed. They aim to ensure the exchange of assets between different users on different blockchains. The assets can be crypto-currencies or physical assets that have been virtualised. These applications are often based on *smart contracts* to establish the rules for the transfer of assets. Although blockchain is a rapidly growing technology, it suffers from a lack of formalism in its systems. For example, *Solidity* is the most widely used language for writing *smart contracts*. *Solidity* is a language with non-formal semantics, making *smart contracts* written in this language vulnerable to attacks. In addition, another aspect that blockchain suffers from is the presence of malicious participants, commonly known as *Byzantine* participants. These participants have random behaviour and are likely not to follow the system's rules. Thus, systems, such as the *cross-chain swap*, need to employ means to ensure their proper functioning despite the presence of *Byzantine* participants.

One way of doing this is through the use of *formal methods*. These are techniques for rigorous mathematical logic reasoning about a computer program to demonstrate its validity against a specific specification. *Model-checking* and *automatic theorem proving* are examples of verification techniques. Model-checking is a method of abstracting a system into a state model and checking if that model satisfies the given specification. As for automatic theorem proving, the method is based on mathematical theorem formulation, reasoning, and logic, to prove a set of propositions.

In this thesis, we propose to apply these verification methods to the systems we have mentioned, namely *smart contracts* and *cross-chain swap* applications. First, we propose the *WhyML* language, dedicated to formal program verification, as a *smart contract* writing language. The language is based on mathematical logic to prove the correctness of the program. The approach is to define a *smart contract* in the form of mathematical formulas and then apply tools that prove that the formulas are true. Applying this approach ensures that the contracts are correct before they are stored in the blockchain. The results obtained from this first study showed that *WhyML* was suitable as a language for writing *smart contracts*, thus allowing for programs correct by construction. In a second step, we apply formalism methods to *cross-chain swap* applications. The first step of this formalisation is to define a formal specification of the *cross-chain swap* problem. The approach consists in defining the properties that characterise the specification of the problem; for example, the specification must be resilient to *Byzantine* participants. In distributed systems, two types of properties are characterised: *safety* properties, which guarantee that "*nothing bad will happen*", and *liveness* properties, which guarantee that "*something good will eventually happen*". Liveness is a property expressed as a function of time, which is not the case for safety. Thus, the *cross-chain swap* problem defines one property of safety and two properties of liveness.

The second step is constructing an algorithm that must satisfy the specification of the *cross-chain swap*. Once these two steps are completed, we apply formal methods to the algorithm to prove its correctness. For this approach, we modelled our algorithm using TLA$^+$, a specific language for modelling distributed and concurrent systems. We then checked whether the resulting model satisfies the properties of the problem under study. We used two different methods depending on the types of properties. We applied the theorem proving method for the proof of safety and the model checking method to verify liveness properties. The results obtained showed that the *cross-chain swap* algorithm we constructed satisfies the properties of the *cross-chain swap* specification in the presence of *Byzantine* participants.

Keywords: formal methods, model-checking, theorem proving, distributed systems, blockchain, smart contracts, cross-chain swap, why3, tla+.

Contents

Part I

Introduction

Chapter 0

Introduction En Français

“ Ce n’est pas assez d’avoir l’esprit bon, mais le principal est de l’appliquer bien. ”

– René Descartes

Contents

0.1 Contexte et Motivation

Imaginons deux personnes, Alice, une investisseuse vivant à Paris, et Bob, un propriétaire d'immobilier vivant à Séoul. Alice souhaite investir une grosse somme d'argent dans un bien immobilier dont Bob est propriétaire. Elle ne prévoit pas de se rendre en Corée du Sud et souhaite investir à distance. Il en va de même pour Bob, qui ne souhaite pas se rendre en France. Les deux personnes ne sont pas amies et ne se font pas confiance. Elles doivent donc trouver un moyen efficace et sûr d'effectuer la transaction.

Une solution consiste à faire appel à un tiers ou à un intermédiaire. Appelons cet intermédiaire Charlie. La transaction se déroule comme suit : Alice donne à Charlie l'argent à investir. Bob fait de même et donne les droits immobiliers d'Alice à Charlie. Charlie a maintenant l'argent et les droits en sa possession et peut ensuite transférer l'argent à Bob et les droits à Alice. Cependant, Alice et Bob doivent faire confiance à Charlie. Charlie a un pouvoir total sur les actifs d'Alice et de Bob et peut décider de ne pas conclure la transaction et de repartir avec l'argent et les droits immobiliers. De plus, Alice ne peut pas envoyer l'argent directement à Bob via sa banque, car elle ne peut pas garantir que Bob reconnaîtra l'investissement une fois qu'il aura reçu son argent. Les solutions proposées jusqu'à présent ne sont pas suffisamment efficaces pour les deux personnes, car l'une ou l'autre peut être perdante lors du transfert. Le principal problème des solutions citées est que les transactions sont exécutées de manière centralisée et nécessitent une partie de confiance. Par conséquent, nous pouvons imaginer qu'une solution décentralisée pourrait résoudre le problème.

Des systèmes tels que *Bitcoin* [144] ou *Ethereum* [50] fournissent précisément une solution décentralisée, permettant des transactions en ligne utilisant un système décentralisé pour envoyer de l'argent ou d'autres données numériques directement d'une partie à une autre sans dépendre d'un tiers. Ces systèmes sont basés sur la technologie *blockchain* [144]. La blockchain a fait l'objet d'une attention croissante ces dernières années. Un système blockchain est un grand registre distribué qui stocke des données et ne peut être modifié. Cette technologie populaire a été appliquée à la finance [166], aux dossiers médicaux [75], et même à la politique avec le vote numérique [84]. Les transactions de diverses données peuvent être améliorées en utilisant des *smart contracts* [187].

Les smart contracts sont des programmes informatiques qui permettent de définir des règles de transactions. Lorsque le smart contract est écrit et approuvé par les deux parties, le contrat peut être stocké dans la blockchain, et personne ne peut le modifier. Par conséquent, si nous supposons que les droits immobiliers peuvent être dématérialisés et envoyés par voie numérique, Alice et Bob peuvent utiliser efficacement une blockchain et un smart contract pour réaliser la transaction. Les deux parties doivent établir un smart contract qui permet l'échange d'actifs de manière contrôlée et automatique. Les deux parties se mettent d'accord sur les conditions et les règles du transfert. Par exemple, une règle qui pourrait être énoncée dans le contrat serait que l'échange d'actifs doit se faire de manière atomique. Alice reçoit les droits immobiliers en même temps que Bob reçoit l'argent. Cette règle garantit que Bob ne peut pas récupérer l'argent d'Alice sans lui donner les droits immobiliers. Une fois écrit et exécuté sur la blockchain, le smart contract agira comme Charlie. Alice apporte son investissement au contrat, et Bob fait de même avec les droits immobiliers. Les deux parties obtiendront leurs actifs si les conditions du contrat sont remplies.

Cependant, pouvons-nous être sûrs que la transaction aura lieu en toute sécurité ?

Par sécurité, on entend que la transaction se déroule comme il se doit, sans bugs ni erreurs pendant son exécution. Néanmoins, le processus de transfert est basé sur des programmes informatiques tels que les smart contracts et la blockchain. Nous sommes confrontés à un problème commun : Tout logiciel ou ordinateur peut comporter des bugs ou des erreurs. Un bug dans la blockchain peut avoir de graves conséquences, par exemple la perte d'argent d'Alice ou les droits immobiliers de Bob. En outre, nous sommes également confrontés à un éventuel mauvais comportement des deux parties. Pour garantir une transaction sûre, nous devons appliquer des méthodes ou des techniques pour vérifier les programmes dont dépend la transaction.

Les méthodes formelles sont un moyen rigoureux et fiable de s'assurer qu'un programme fonctionne sans bugs. Elles désignent des techniques et un ensemble de notations permettant de modéliser et d'analyser des systèmes complexes en tant qu'entités mathématiques. La construction d'un modèle mathématique d'un système et l'utilisation de preuves mathématiques permettent de vérifier ses propriétés afin de garantir un comportement correct. Il existe un large éventail de techniques de vérification pour établir la justesse d'un système. Cette thèse se concentre sur les techniques de *vérification de modèles* [58] et de *démonstration automatique de théorèmes* [163]. En appliquant l'une des méthodes de vérification au système dont dépend la transaction d'Alice et Bob, nous pouvons garantir que la transaction peut être effectuée efficacement par la blockchain et en toute sécurité par des méthodes formelles. Ce scénario d'échange d'actifs entre Alice et Bob basé sur la blockchain a motivé les travaux réalisés dans cette thèse. Les principaux travaux de recherche peuvent être formulés comme suit :

- Comment s'assurer que le smart contract utilisé par Alice et Bob soit correct et respecte les conditions de transfert ?
- En supposant que le smart contract soit correct, comment garantir le transfert des actifs dans l'hypothèse où l'une des deux parties se comporterait de manière malveillante ? C'est-à-dire qu'elle ne respecterait pas les règles de transfert.

Dans ce qui suit, nous donnons un aperçu de la technologie utilisée dans cette thèse, comme les bases de la blockchain et les méthodes formelles, et nous expliquons notre contribution.

0.1.1 Les Bases de la Blockchain

Un système blockchain est un grand registre de comptes *distribué* et *décentralisé*. Le terme *décentralisé* fait référence aux niveaux de contrôle et de prise de décision. Dans les systèmes décentralisés, il n'y a pas d'entité centrale de contrôle. Au lieu de cela, le contrôle est partagé entre plusieurs entités indépendantes. Le terme *distribué* fait référence aux niveaux de localisation. Dans un système distribué, toutes les parties du système sont situées dans des lieux physiques distincts.

La blockchain est devenue connue comme la technologie sous-jacente qui permet l'existence des crypto-monnaies. Le bitcoin [144], la crypto-monnaie la plus connue, est le premier exemple de mise en œuvre réussie de la technologie blockchain. Il s'agit d'une monnaie numérique décentralisée que les utilisateurs peuvent transférer anonymement sans l'interférence d'une autorité tierce en envoyant la monnaie en pair à pair à travers le réseau Bitcoin. Les crypto-monnaies de l'utilisateur sont stockées dans des *portefeuilles* numériques. Outre le bitcoin, d'autres crypto-monnaies sont également alimentées par la technologie blockchain comme l'*ether* [50] (la crypto-monnaie d'Ethereum). Au moment d'écrire ces lignes, il existe pas moins de 10 000 autres crypto-monnaies en circulation [1].

La structure d'une chaîne de blocs. La blockchain conserve un historique en croissance continue d'informations ordonnées inaltérables organisées en une chaîne de blocs, comme le montre la Figure 1. Un bloc est identifié par un hachage généré à l'aide d'un algorithme de hachage cryptographique et possède une hauteur qui lui permet d'être positionné dans la chaîne ; le bloc N est plus ancien que le bloc $N+1$, qui est lui-même plus ancien que le bloc $N+2$. Chaque bloc fait référence au bloc précédent dans la chaîne, appelé bloc parent, par le biais du champ "Hash du bloc ..." de la Figure 1. Un bloc contient le hachage de son parent ; ainsi, la séquence de hachages reliant chaque bloc à son parent crée une chaîne remontant au premier bloc jamais créé, connu sous le nom de *genesis block*. Le *genesis block* est le premier bloc de la blockchain. C'est l'ancêtre commun de tous les blocs, ce qui signifie que si l'on part de n'importe quel bloc et que l'on suit la chaîne en remontant dans le temps, on finit par arriver au genesis block. Sans ce composant, il n'y aurait pas de chronologie et de connexion entre chaque bloc. Un autre composant important de la Figure 1 est la

[1] Données de février 2022 provenant du site web : https://www.statista.com/statistics/863917/number-crypto-coins-tokens/

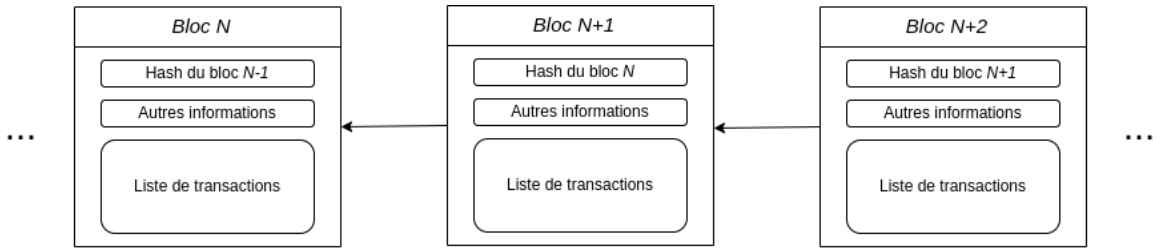


Figure 1 – Structure de données d'une blockchain

liste des transactions. Cette liste est une structure de données conteneur regroupant les transactions confirmées dans le bloc. Chaque transaction émise et confirmée par la blockchain est stockée dans un bloc afin d'avoir une traçabilité des transactions.

Mécanisme de consensus. Les blocs sont ajoutés à la blockchain par un mécanisme de consensus qui garantit la préservation de la structure de la chaîne. Les utilisateurs chargés de la validation des blocs doivent se mettre d'accord sur le prochain bloc ajouté afin d'éviter les *forks*. On parle de fork lorsque deux ou plusieurs utilisateurs de la blockchain ont une vision différente de la chaîne. Il existe différents mécanismes de consensus, et chacun a ses spécificités. On peut citer, *Proof-of-Stake* (*PoS*) [178], *Proof-of-Authority* (*PoA*) [66], et *Practical Byzantine Fault-Tolerant* (*PBFT*) [54]. Dans le cas de Bitcoin, le mécanisme de consensus est le *Proof-of-Work* (*PoW*) qui nécessite la résolution d'un calcul cryptographique pour avoir le droit d'ajouter un bloc. Dans Bitcoin, les utilisateurs qui effectuent le calcul sont appelés *mineurs*. Cependant, le consensus *PoW* n'offre pas une cohérence solide, car des forks peuvent se produire, ce qui entraîne des problèmes critiques. Pour surmonter ce problème, de nouvelles techniques d'ajout de blocs ont vu le jour. Ces techniques garantissent qu'un fork ne peut pas se produire, en supposant des hypothèses claires. Ces mécanismes de consensus, par exemple *PBFT*, définissent un ensemble de *validateurs* pour valider les blocs et un sous-ensemble de validateurs signe chaque bloc.

Smart contracts. Une caractéristique qui a suscité un vif intérêt pour les blockchains est l'écriture de smart contrats [187]. Un contrat est un ensemble de promesses qui reconnaît et régit les devoirs et les règles de transaction préétablies découlant d'accords entre des participants non-confiants, qui sont appliqués par le consensus de la blockchain [57].

Nick Szabo a proposé pour la première fois des smart contrats en 1994 [172]. Szabo a défini les smart contrats comme des protocoles de transaction informatisés qui exécutent les termes d'un contrat. Des années après l'article de Szabo, les smart contrats ont été popularisés par Ethereum publié en 2015 [187]. Un smart contrat est devenu un protocole numérique écrit dans un langage de programmation de haut niveau. Par exemple, *Solidity* [78] est le langage de programmation de contrat d'Ethereum. Chaque contrat *Solidity* est identifié par une *adresse* et contient une quantité d'*ethers*, la crypto-monnaie d'Ethereum. Un contrat est un programme séquentiel impératif et exécutable qui s'exécute dans les blockchains. Cela consiste en un ensemble d'instructions pour effectuer des actions spécifiques. Il peut manipuler des fonctions et des variables et invoquer d'autres contrats en envoyant des transactions à l'adresse du contrat cible.

L'architecture blockchain. L'architecture blockchain peut être vue en couches, comme le montre la Figure 2. La couche matérielle peut être considérée comme la couche sur laquelle le système blockchain est construit. Le contenu de la blockchain (les blocs et les transactions) est stocké dans des serveurs physiques situés quelque part sur terre. En d'autres termes, la couche matérielle stocke la couche de données qui se compose des éléments de la Figure 1. La couche réseau représente la communication entre les utilisateurs de la blockchain. Lorsqu'un bloc est créé dans la blockchain, il est propagé à tous les utilisateurs de la blockchain. Cette propagation s'effectue en pair à pair à travers la couche réseau. Comme mentionné précédemment, les utilisateurs de la blockchain

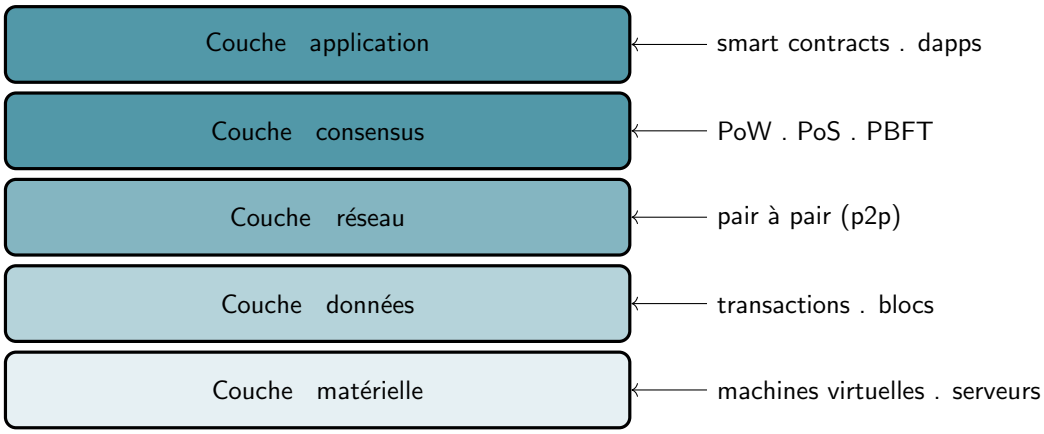


Figure 2 – Les couches d'une blockchain

doivent effectuer un mécanisme de consensus pour ajouter des blocs à la chaîne. La couche de consensus est chargée de valider les blocs, de les ordonner et de garantir que tout le monde est d'accord. Cette couche est l'une des fonctions les plus critiques des blockchains. La dernière est la couche d'application qui comprend les programmes que les utilisateurs finaux utilisent pour interagir avec la blockchain, par exemple les smart contracts et les applications décentralisées (dapp). Une application décentralisée (dapp) est construite sur un réseau décentralisé qui combine un smart contract et une interface utilisateur frontale.

0.1.2 Vulnérabilités des Smart Contracts

Les smart contracts sont un sujet d'actualité depuis leur apparition en 2015. Les avantages qu'ils procurent ont contribué à populariser l'utilisation de la blockchain. On trouve des smart contracts dans de nombreux domaines, de la finance [166] à l'agriculture [160]. Bien qu'il existe différents types de contrats sur le marché, tels que *Michelson* [173] et *Chaincode* [21], *Solidity* reste dominant quant au nombre de contrats déployés sur la blockchain. *Solidity* a connu une explosion d'utilisation, mais est maintenant victime de son succès. L'utilisation accrue des contrats s'est faite au détriment de la sécurité des contrats. Au fil du temps, il est devenu évident que les contrats présentent plusieurs failles et vulnérabilités. Ils sont souvent confrontés à des attaques croissantes exploitant les vulnérabilités d'exécution des smart contracts, ce qui conduit à d'importants scénarios malveillants. L'une des attaques les plus connues est l'attaque "*the DAO*" [24]. Un *DAO*, pour "decentralised autonomous organisation", est un smart contract déployé sur la blockchain Ethereum qui fonctionne comme un fonds de capital-risque décentralisé. Un hacker a exploité de manière récursive une faille dans le code de "*the DAO*" qui lui a permis de collecter des *ethers* dans une *DAO* secondaire à plusieurs reprises. L'attaque a entraîné une perte de 3,6 millions d'*ethers*.

Un autre exemple d'attaque contre les smart contracts est le "*Parity Wallet Hack*" [8]. Parity [7] est une société qui construit des infrastructures blockchain dans l'écosystème Ethereum, et les portefeuilles sont des smart contracts qui stockent de l'argent. L'origine de la faille provient d'une bibliothèque qui est elle-même un smart contract. Ce contrat bibliothèque possède des fonctions permettant de créer des portefeuilles multi-signatures. Les portefeuilles multi-signatures sont comme les portefeuilles ordinaires en ce sens qu'ils sont également des smart contracts, mais ils nécessitent plusieurs approbations pour retirer un montant quelconque du portefeuille. Tous les portefeuilles multi-signatures créés dépendent de la bibliothèque. Un hacker a profité d'une faille dans le contrat pour contrôler la bibliothèque, rendant tous les portefeuilles dépendants inutiles. Tous les fonds stockés dans les portefeuilles Parity affectés ne pouvaient plus être retirés. Les portefeuilles affectés avaient une somme estimée à 500 000 *ethers*.

Ces exemples montrent qu'une vulnérabilité dans un smart contract peut avoir de graves conséquences. De plus, les erreurs dans les smart contracts, une fois publiées, ne peuvent pas être corrigées en raison de la nature immuable de la blockchain.

Dans cette thèse, nous étudions, en particulier, les vulnérabilités des smart contracts écrits en *Solidity*. L'étude montre que *Solidity* présente différentes causes de bugs qui augmentent sa vulnérabilité aux attaques.

0.1.3 Interopérabilité entre les Blockchains

Revenons à l'exemple d'Alice et Bob, qui utilisent la blockchain pour effectuer des transferts d'actifs. Supposons que l'argent d'Alice soit sur une blockchain différente de celle de Bob, où les droits de propriété sont numérisés. La question est de savoir comment effectuer l'échange en sachant que les deux actifs sont sur des blockchains différentes ? Cette question devient courante à mesure que la technologie blockchain devient populaire dans de nombreux domaines. Ainsi, son utilisation a considérablement augmenté ces dernières années depuis la création de nombreuses crypto-monnaies et blockchains. Le besoin de communication entre les différentes blockchains est apparu chez les utilisateurs. Par conséquent, le développement d'infrastructures permettant la communication entre elles est devenu nécessaire.

En 1996, Wegner a déclaré que "*l'interopérabilité est la capacité de deux ou plusieurs composants logiciels à coopérer malgré les différences de langage, d'interface et de plate-forme d'exécution*" [183]. Améliorer l'interopérabilité entre les blockchains semble être la solution pour établir des moyens d'échange entre elles. L'avantage d'assurer l'interopérabilité entre les blockchains est d'explorer de nouvelles fonctionnalités, de mettre à l'échelle les fonctionnalités existantes et de créer de nouveaux cas d'utilisation, par exemple le cas d'Alice et Bob, qui devraient pouvoir transférer leurs actifs d'une blockchain à l'autre.

Il existe plusieurs techniques d'interopérabilité pour permettre la communication entre les blockchains [35]. Certaines permettent l'échange entre blockchains de la même famille, c'est-à-dire que les blockchains doivent être du même type et sont construites selon les mêmes règles. D'autres permettent la communication entre des blockchains qui ne sont pas de la même famille, c'est-à-dire que les blockchains ont des règles et des mécanismes de fonctionnement différents. En fonction du type de blockchain, le système assurant l'interopérabilité sera différent. En effet, une blockchain peut être décrite comme *permissioned*, *permissionless*, *public*, et *private*. La caractéristique de permissioned/permissionless fait référence à l'anonymat des utilisateurs, tandis que public/private fait référence à la participation au mécanisme de consensus. Cette thèse se concentre sur un système qui aborde ces questions d'applications distribuées pour le commerce d'actifs exploitant des smart contracts. Récemment, une application basée sur les smart contracts a gagné en popularité, à savoir les applications d'échanges inter-chaîne (*cross-chain swap*). Ces applications permettent aux utilisateurs de différentes blockchains de transférer des actifs de manière décentralisée et sans l'intervention d'un intermédiaire. Certaines applications *cross-chain swap* nécessitent une synchronisation entre les utilisateurs du système pour procéder au transfert ; d'autres ne le font pas, c'est-à-dire que le système peut être exécuté dans un environnement asynchrone, ce qui implique que les utilisateurs n'ont pas à synchroniser leurs actions.

0.1.4 Correction des Applications de *Cross-Chain Swap*

Un système d'échanges inter-chaîne implique plusieurs participants qui exécutent des actions du système pour atteindre un objectif commun connu. Cependant, ces systèmes sont compliqués à gérer, car ils sont distribués et souvent sujets à des comportements involontaires, c'est-à-dire des utilisateurs malveillants. Les auteurs de [193] prouvent qu'aucun système cross-chain asynchrone n'est tolérant aux utilisateurs malveillants à moins de supposer un tiers de confiance. Ce tiers de confiance peut être centralisé ou décentralisé, par exemple une autre blockchain. Le problème qui peut se poser est qu'un système qui prétend être tolérant aux utilisateurs malveillants dans un environnement asynchrone ne l'est pas. Par conséquent, les participants corrects du système peuvent être perdants à la fin de l'exécution du système.

Compte tenu de ces problèmes, il est apprécié d'appliquer une correction comportementale à de tels systèmes. La correction comportementale est la capacité de garantir que le système est exécuté comme prévu, sans conséquences involontaires, par exemple le verrouillage ou le vol d'actifs. Un moyen est de s'assurer que le système (ou l'algorithme du système [2]) soit correct en ce qui concerne sa spécification. Par exemple, on applique des méthodes de vérification formelle pour vérifier l'exactitude des algorithmes en fonction d'une spécification. Cependant, quand on parle de vérification formelle, on parle de vérification automatique ou semi-automatique, qui implique l'utilisation d'outils de vérification. Par ailleurs, il est essentiel de définir une spécification réaliste pour un problème de *cross-chain swap*. Par exemple, plusieurs spécifications existantes de *cross-chain swap* incluent la propriété d'*atomicité*, même dans un environnement asynchrone [191]. L'atomicité fait référence au transfert de tous les actifs ou d'aucuns. Cependant, cette propriété ne semble pas satisfaite dans un système avec des participants malveillants ; ainsi, l'atomicité est souvent remise en question.

Dans cette thèse, nous présentons une spécification du problème *cross-chain swap* ainsi qu'un algorithme satisfaisant la spécification. Nous expliquons comment nous assurons un algorithme sûr en supposant la présence de participants malveillants dans un environnement asynchrone sans assurer l'atomicité.

0.1.5 Les Méthodes Formelles

Les systèmes logiciels augmentent inévitablement en taille et en fonctionnalité ; le nombre d'erreurs subtiles augmente avec la complexité. Une erreur ou un bug est un problème courant que tout programme informatique peut rencontrer. Elle peut provenir d'une mauvaise écriture du programme, d'une faute de frappe ou d'une mauvaise gestion de la mémoire. De plus, certaines de ces erreurs peuvent devenir un problème important et entraîner des pertes catastrophiques d'argent, de temps, voire de vie humaine. On peut citer le tristement célèbre crash d'ARIANE 5 [72]. Il est donc nécessaire de construire des systèmes, et en particulier des systèmes critiques, en tenant compte de cette complexité. Les méthodes formelles font référence aux techniques et notations logiques permettant de modéliser et d'analyser des systèmes complexes en tant qu'entités mathématiques. La construction d'un modèle mathématique d'un système et l'utilisation de preuves mathématiques permettent de vérifier ses propriétés pour garantir un comportement correct. Elles s'appliquent aussi bien aux programmes séquentiels qu'aux programmes distribués.

Une propriété est une caractéristique d'un programme qui est vraie pour chaque exécution possible de ce programme. Les propriétés d'intérêt pour les programmes distribués se divisent en deux catégories : *sûreté* et *vivacité*. Une propriété de sûreté affirme que "*rien de mauvais n'arrivera pendant l'exécution*", c'est-à-dire que le programme n'atteint pas un mauvais état. Les propriétés de sûreté représentent des exigences que le système doit maintenir en permanence. Elles expriment souvent des propriétés d'invariance. La propriété de vivacité affirme que "*quelque chose de bon finira par arriver*", c'est-à-dire que le programme doit finir par atteindre un bon état. Les propriétés de vivacité représentent des exigences qui n'ont pas besoin d'être maintenues en permanence, mais qui doivent garantir une réalisation éventuelle (ou répétée). Habituellement, dans les programmes séquentiels, ce qui est prouvé, ce sont les propriétés de sûreté.

La vérification formelle des programmes est un domaine de recherche actif depuis les débuts de l'informatique, et diverses techniques sont apparues depuis. Nous distinguons trois grandes familles d'approches de la vérification : la *vérification de modèles*, l'*interprétation abstraite* et la *démonstration automatique de théorèmes*.

Démonstration automatique de théorèmes

La démonstration de théorèmes, ou démonstration automatisée de théorèmes, repose sur la formulation d'un théorème mathématique, le raisonnement et la logique, pour prouver un ensemble de propositions. Elle peut être utilisée pour traiter des systèmes infinis. Ces systèmes sont

[2] Un algorithme caractérise une suite d'étapes qui permet de fournir un résultat à partir d'éléments d'entrée [134].

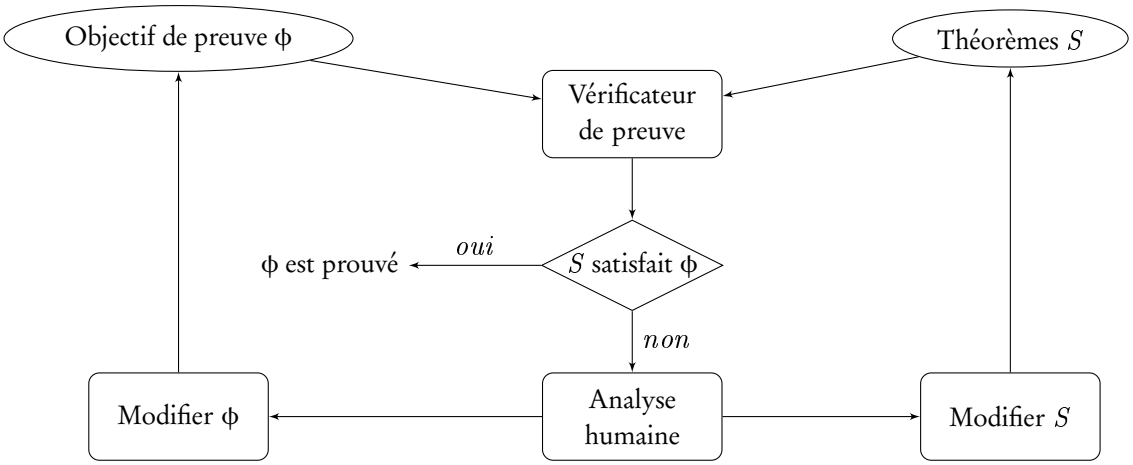


Figure 3 – Processus interactif de démonstration de théorèmes

définis et spécifiés par les utilisateurs dans une logique mathématique appropriée. Les prouveurs de théorèmes vérifient les propriétés fondamentales et critiques du système et utilisent des techniques d'aide à la preuve. Les fondements de la preuve automatique de théorèmes sont *la logique propositionnelle*, *la logique du premier ordre* et *la logique de l'ordre supérieur* [107]. L'utilisation de ces langages permet d'énoncer rigoureusement un large éventail de problèmes de manière non-ambiguë. La logique propositionnelle est utilisée pour représenter les propositions atomiques à l'aide d'opérateurs booléens mathématiques. La logique du premier ordre est l'extension de la logique propositionnelle qui autorise les quantificateurs. La logique des prédicats est la catégorie générale à laquelle appartient la logique du premier ordre. La logique d'ordre supérieur étend la logique du premier ordre en supportant de nombreux types de quantification. La logique de l'ordre supérieur permet aux prédicats d'accepter des prémisses (également des prédicats) et permet la quantification sur les prédicats et les fonctions, ce qui n'est pas le cas pour la logique du premier ordre. Cependant, dans cette thèse, nous utilisons la logique propositionnelle et la logique du premier ordre.

La Figure 3 illustre le cadre interactif du prouveur de théorème pour construire une preuve vérifiée mécaniquement. Une étape automatique et une étape d'interaction humaine sont menées consécutivement pour chaque objectif de preuve. Une méthodologie de preuve interactive commence par la construction manuelle de la preuve, qui consiste à décrire le but de la preuve ϕ et à fournir les théorèmes correspondants S qui sont soit prouvés soit supposés. Étant donné le but de la preuve ϕ et un ensemble de théorèmes S comme exigences écrites dans un langage de spécification formel, un prouveur de théorèmes automatique peut être lancé comme indiqué dans la Figure 3 pour déduire automatiquement la preuve en utilisant les règles ou les calculs intégrés mis en œuvre dans ces prouveurs. Si le but de la preuve ϕ est mécaniquement dérivable de l'ensemble des théorèmes S, le prouveur répondra avec une preuve le vérifiant. Sinon, en fonction de l'analyse humaine, soit les théorèmes, soit le but doivent être modifiés.

Des outils comme Dafny [130], Frama-C [64], VeriFast [109], et *Why3* [83] prouvent des programmes en utilisant cette technique appelée aussi *la vérification déductive*. Par exemple, *Why3* est une plateforme de vérification déductive de programmes. Elle fournit un langage riche pour la spécification (code logique) et la programmation (code impératif) appelé *WhyML*.

Vérification de modèles

La vérification de modèles est une technique de vérification formelle automatique basée sur une description du comportement compris dans une machine à états finis [58]. Cette technique effectue une inspection systématique efficace de toutes les séquences d'états possibles décrites par le modèle. La technique prouve si le modèle satisfait certaines propriétés comportementales. La sémantique de la machine à états est donnée par un système de transitions qui peut être plus ou moins complexe.

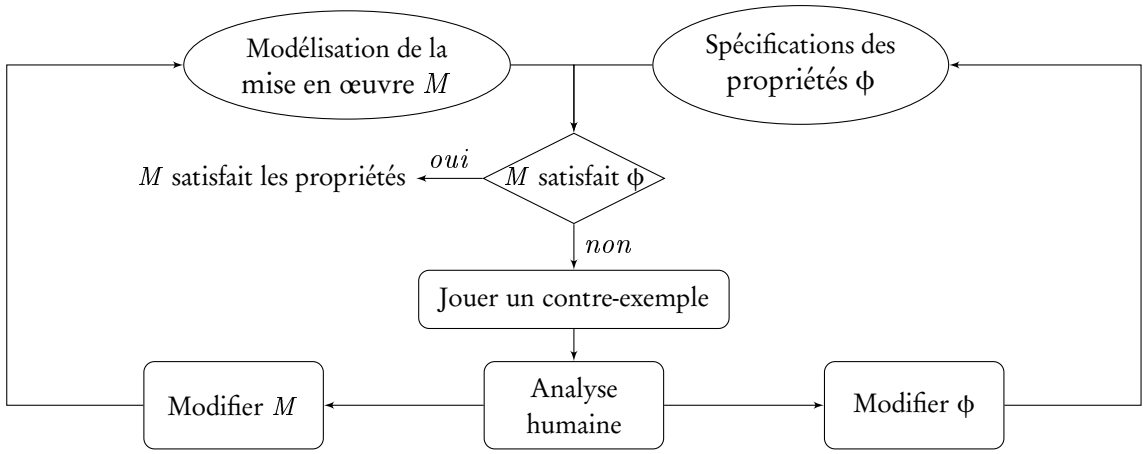


Figure 4 – Processus de la vérification de modèles

Un système de transitions va des machines à états finis (automates finis) aux programmes réels (machines de Turing). Ainsi, le principal défi de la vérification de modèles est l'explosion combinatoire du modèle. Néanmoins, cette technique est pertinente pour vérifier les spécifications partielles au début du processus de conception [58]. La Figure 4 représente le processus de vérification de modèles, qui vérifie si le modèle M d'un système satisfait ou non sa spécification ϕ écrite sous forme de formule logique. Si le modèle viole une propriété considérée, la vérification de modèles fournit un contre-exemple d'une séquence qui conduit à la propriété de violation. Le contre-exemple peut être avantageux pour adapter le modèle (ou la spécification). En effet, de nombreuses applications industrielles réussies témoignent de la performance des outils de vérification de modèles [27].

Un large choix d'outils effectue la vérification de modèles, dont la toolbox TLA$^+$ [120]. TLA$^+$ utilise comme vérificateur de modèle TLC [190] qui fournit une plateforme pour vérifier le modèle des spécifications écrites dans le langage TLA$^+$. Parmi les autres outils existants, SPIN [106], ProB [131], UPPAAL [96] et NuSMV [55] peuvent être cités pour représenter trois types de vérificateurs de modèles basés sur différentes techniques de modélisation [85].

Interpretation abstraite

La résolution de la fiabilité des programmes informatiques est un problème bien connu dans le domaine des logiciels. L'analyse statique du comportement des programmes pendant leur exécution fournit des solutions à ce problème. Cependant, ce moyen d'analyse peut être indécidable et nécessite une certaine forme d'approximation. L'objectif de l'interprétation abstraite est de formaliser cette idée d'approximation [63].

L'interprétation abstraite est basée sur un raisonnement sémantique abstrait, moins précis mais plus facile à manipuler. Par conséquent, certaines informations seront volontairement perdues, résultant d'une exécution partielle d'un programme informatique. Cette méthode permet d'obtenir des informations sur la sémantique du programme sans effectuer tous les calculs.

Cette méthode est appliquée à la sûreté et à la sécurité des systèmes informatiques matériels et logiciels complexes. Sa principale application est l'analyse statique formelle et l'extraction automatique d'informations sur les exécutions possibles des programmes informatiques. Ces analyses ont deux usages principaux : à l'intérieur des compilateurs, pour analyser les programmes afin de décider si des optimisations ou des transformations spécifiques sont applicables, pour le débogage, ou même la certification des programmes contre des classes de bugs.

0.2 Contributions et Organisation

Ce manuscrit est organisé comme suit. Le Chapitre 2 fournit un contexte théorique sur la technologie utilisée dans cette thèse, et le Chapitre 3 présente l'état de l'art sur les aspects de la recherche

abordés dans cette thèse et donne un aperçu de ce qui existe pour fournir un positionnement scientifique claire. Le Chapitre 4 fournit les contextes techniques tels que les notations et les outils utilisés tout au long du manuscrit. Les Chapitres 5, 6, 7, et 8 présentent les résultats techniques de cette thèse, et le Chapitre 9 donne une conclusion générale sur les différents résultats et les directions pour les travaux futurs dans la continuité des résultats obtenus dans ce manuscrit.

Dans ce qui suit, nous donnons un aperçu des contributions de la thèse.

Des smart contracts corrects et prouvés. Les systèmes de blockchain manipulent les informations relatives aux crypto-monnaies et aux transactions par le biais de smart contracts. Par conséquent, si un bug survient dans la blockchain, de graves conséquences peuvent se produire, comme par exemple la perte d'argent. Un contrat peut définir n'importe quel ensemble de règles représentées dans son langage de programmation (Par exemple, *Solidity* pour Ethereum), permettant ainsi la mise en œuvre d'applications décentralisées. Les smart contracts sont des programmes qui présentent des vulnérabilités pouvant être exploitées et attaquées. Il est crucial de garantir des smart contracts sûrs et corrects et d'éviter les bugs informatiques avant leur utilisation. Il serait intéressant d'utiliser des langages formels pour écrire, vérifier et compiler de tels programmes et définir leurs propriétés.

Cette thèse propose un langage dédié à la vérification déductive, appelé *WhyML*, pour être un nouveau langage d'écriture de smart contracts formels et vérifiés. L'objectif est d'éviter les attaques exploitant les vulnérabilités d'exécution de ces contrats. Nous appliquons les concepts de la vérification déductive et développons une méthodologie de preuve des smart contracts. La méthode présentée a été appliquée à un cas d'utilisation qui décrit une place de marché de l'énergie permettant le commerce local de l'énergie entre les habitants d'un quartier. La modélisation qui en résulte permet d'établir un contrat d'échange non-trivial pour mettre en relation des consommateurs et des producteurs désireux d'échanger de l'énergie. En outre, ce dernier point démontre qu'avec une approche déductive, il est possible de modéliser et de prouver des programmes à une échelle réaliste, permettant ainsi la vérification de propriétés fonctionnelles plus réalistes.

Cette contribution, détaillée dans le Chapitre 5, a été publiée et présentée dans les actes d'une conférence évaluée par des pairs [146].

Description d'un algorithme de *cross-chain swap*, $\mathscr{P}_{swap}$, et sa vérification formelle. Une fois stocké dans la blockchain, tout utilisateur de la blockchain peut utiliser et appeler un smart contract, y compris d'autres smart contracts. Par conséquent, les applications blockchain peuvent exploiter les smart contracts stockés et les utiliser pour répondre aux besoins de l'application. Les applications *cross-chain swap* sont de tels systèmes qui utilisent des smart contracts pour réaliser des transactions entre utilisateurs. Les avantages de ce type de système ont donné lieu à de nombreux articles de recherche ces dernières années. Cependant, la plupart d'entre eux ne sont pas suffisamment formels et ne reflètent pas la réalité. Dans cette thèse, nous avons modélisé formellement un algorithme qui permet des échanges inter-chaîne entre différents registres distribués appelé $\mathscr{P}_{swap}$. En effet, puisque qu'un registre distribué est une classe haut niveau des blockchain, l'algorithme $\mathscr{P}_{swap}$ abstrait toute notion de blockchain afin de ne pas limiter son cadre d'application aux blockchains. De plus, cet algorithme satisfait une spécification réaliste qui prend en compte des hypothèses d'implémentation jamais considérées dans la littérature. Ainsi, la spécification est tolérante aux utilisateurs malveillants, sans hypothèses de proportions, que l'on retrouve le plus souvent dans les systèmes distribués. La spécification définit une propriété de sûreté et une propriété de vivacité qui reste satisfaite même dans un environnement asynchrone. De plus, la description de l'algorithme est définie de manière formelle, ce qui facilite sa compréhension et la possibilité de vérifier son comportement. Cette partie de la contribution est détaillée dans le Chapitre 6. Le Chapitre 7 présente un autre aspect de la contribution, à savoir la vérification formelle de l'algorithme. Par conséquent, nous appliquons l'outil TLA$^+$ à l'algorithme pour prouver qu'il satisfait l'ensemble des propriétés de la spécification *cross-chain swap*. Nous appliquons la méthode de vérification déductive pour prouver la propriété de sûreté et la vérification de modèles pour prouver les propriétés de vivacité sur un modèle qui

inclut des participants malveillants.

Cette contribution et les résultats ont été publiés et présentés dans des actes de conférence examinés par des pairs [147].

Analyse de la compatibilité du *cross-chain swap* $\mathscr{P}_{swap}$ dans un environnement blockchain. La spécification et l'algorithme présentés dans la contribution précédente font des hypothèses de mise en œuvre et imposent des exigences d'instanciation de l'algorithme $\mathscr{P}_{swap}$. Toutefois, certains registres distribués, tel que certains types de blockchains, ne peuvent pas satisfaire l'ensemble de ces exigences. Par conséquent, dans cette thèse, une analyse est faite sur un ensemble de blockchains existantes qui peuvent ou non mettre en œuvre l'algorithme $\mathscr{P}_{swap}$. L'analyse est basée sur les caractéristiques du type de blockchain, qu'elle soit permissioned/permissioneless ou public/private, et sur leur capacité à instancié les exigences de l'algorithme $\mathscr{P}_{swap}$.

Cette contribution, détaillée dans le Chapitre 8, a été partiellement publiée et présentée dans les actes d'une conférence évaluée par des pairs [147].

Chapter 1

Introduction

" Research is what I'm doing when I don't know what I'm doing. "

– Wernher von Braun

Contents

1.1 Context and Motivation

Let us imagine two people: Alice, an investor living in Paris, and Bob, a real estate owner living in Seoul. Alice wants to invest a large sum of money in real estate that Bob owns. She does not plan to travel to South Korea and wishes to invest remotely. So does Bob, who does not wish to travel to France. The two peoples are not friends and do not trust each other. Though, they need to develop an efficient and safe way to undertake the transaction.

One solution is to use a third party or intermediary. Let us call this intermediary Charlie. The transaction proceeds as follows: Alice gives Charlie the money to invest. Bob does the same and gives Alice's immovable rights to Charlie. Now, Charlie has the money and the rights in his possession and can then transfer the money to Bob and the rights to Alice. However, both Alice and Bob have to trust Charlie. Charlie has total power over Alice's and Bob's assets and can decide not to complete the transaction and leave with the money and the immovable rights. Moreover, Alice cannot send the money directly to Bob via her bank because she cannot guarantee that Bob will acknowledge the investment once he receives Alice's money. The solutions so far are not efficient enough for both people, as either one may lose out on the transfer. The main problem with the cited solutions is that the transactions are executed centrally and need a trusting part. Consequently, we can imagine that coming up with a possible decentralised solution could solve the problem.

Systems such as *Bitcoin* [144] or *Ethereum* [50] provide a precisely decentralised solution, allowing online transactions using a decentralised system to send money or other digital data directly from one party to another without relying on a third party. Such systems are based on *blockchain* technology [144]. Blockchain has received increasing attention these recent years. A blockchain system is a distributed ledger that stores data and cannot be modified. This popular technology has been applied to finance [166], medical records [75], and even politics with digital voting [84]. Transactions of various data can be enhanced using *smart contracts* [187].

Smart contracts are computer programs that allow setting rules of transactions. When the smart contract is written and approved by both parties, the contract can be stored in the blockchain, and nobody can modify it. Therefore, if we assume immovable rights can be dematerialised and sent digitally, Alice and Bob can efficiently use a blockchain and smart contract to complete the transaction. Both have to establish a smart contract that allows the exchange of assets in a controlled and automatic way. The two parties agree on the conditions and rules of the transfer. For instance, a rule that could be set out in the contract would be that the exchange of assets should be done in an atomic way. Alice receives immovable rights at the same time that Bob receives the money. This rule guarantees that Bob cannot get Alice's money back without giving immovable rights to her. Once written and executed on the blockchain, the smart contract will act as Charlie. Alice provides her investment to the contract, and Bob does the same with the immovable rights. Both parties will get their assets if the contract's conditions are met.

However, can we be sure that the transaction will take place safely?

Safely is meant that the transaction takes place as it should be without bugs or errors during the transaction's execution. Nevertheless, the transfer process is based on computer programs such as smart contracts and the blockchain. We face a common problem: any software or computer problem may have bugs or errors. A bug in the blockchain can have serious consequences, e.g. the loss of Alice's money or Bob's immovable rights. Moreover, we also face possible misbehaviour from both parties. To ensure a safe transaction, we must apply methods or techniques to verify programs on which the transaction depends.

Formal methods are a rigorous and reliable way to ensure that a program works without bugs. These refer to techniques and a collection of notations for modelling and analysing complex systems as mathematical entities. Building a mathematical system model and using mathematical proof makes it possible to verify its properties to ensure correct behaviour. There is a wide range of verification techniques to establish the correctness of a system. This thesis focuses on the *model-*

checking [58] and the *theorem proving* [163] techniques. By applying one of the verification methods to the system on which Alice's and Bob's transaction depends, we can guarantee that the transaction can be done efficiently through the blockchain and safely through formal methods. This scenario of trading assets between Alice and Bob based on blockchain drove the work done in this thesis. The main research work can be formulated as follows:

- How to ensure that the smart contract used by Alice and Bob is correct and respects the transfer conditions?
- Assuming the smart contract is correct, how to ensure the transfer of the assets assuming that one of the two parties may behave maliciously? i.e. does not respect the transfer rules.

In the following, we give an overview of the technology used in this thesis, such as blockchain basics and formal methods, and explain our contribution.

1.1.1 Blockchain Basics

A blockchain system is a *distributed* and *decentralised* ledger. The term *decentralised* refers to levels of control and decision-making. In decentralised systems, there is no central controlling entity. Instead, control is shared among several independent entities. The term *distributed* refers to levels of location. In a distributed system, all parts of the system are located in separate physical locations.

Blockchain became known as the underpinning technology that enables the existence of cryptocurrency. Bitcoin [144], the best-known cryptocurrency, is the first example of the successful implementation of blockchain technology. It is a decentralised digital currency that users can anonymously transfer without the interference of a third-party authority by sending the currency in a peer-to-peer way through the Bitcoin network. The user's cryptocurrencies are stored in digital *wallets*. Besides bitcoin, other cryptocurrencies are powered by blockchain technology like *ether* [50] (Ethereum's cryptocurrencies). When writing these lines, there are no less than 10,000 other cryptocurrencies in circulation [1].

The structure of a chain of blocks. The blockchain maintains a continuously growing history of unalterable ordered information organised in a chain of blocks, as depicted in Figure 1.1. A block is identified by a hash generated using a cryptographic hash algorithm and has a height that allows it to be positioned in the chain; Block N is older than Block $N+1$, which is itself older than Block $N+2$. Each block refers to the previous block in the chain, known as the parent block, through the "Hash of block ..." field in Figure 1.1. A block contains the hash of its parent; thus, the sequence of hashes linking each block to its parent creates a chain going back to the first block ever created, known as the *genesis block*. The *genesis block* is the first block in the blockchain. It is the common ancestor of all the blocks, meaning that if we start at any block and follow the chain backwards in time, we will eventually arrive at the genesis block. Without this component, there would be no chronology and connection between each block. Another significant component from Figure 1.1 is the list of transactions. This list is a container data structure aggregating confirmed transactions within the block. Each transaction issued and confirmed by the blockchain is stored in a block to have transactions traceability.

Consensus mechanism. Blocks are added to the blockchain through a consensus mechanism that ensures the preservation of the chain structure. The users in charge of the block validation must agree on the next added block to avoid *forks*. A fork is when two or more blockchain users have a different view of the chain. There are different consensus mechanisms, and each has its specificities. One can cite, *Proof-of-Stake* (*PoS*) [178], *Proof-of-Authority* (*PoA*) [66], and *Practical Byzantine Fault-Tolerant* (*PBFT*) [54]. In Bitcoin, the consensus mechanism is the *Proof-of-Work* (*PoW*) which

[1] Data of February 2022 from the website: https://www.statista.com/statistics/863917/number-crypto-coins-tokens/

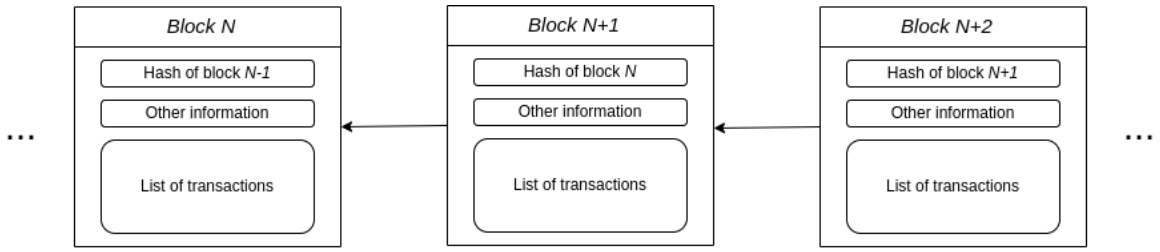


Figure 1.1 – Data structure of a blockchain

requires solving a cryptographic computation to have the right to add a block. In Bitcoin, the users that perform the computation are called *miners*. However, *PoW* consensus does not provide a strong consistency since forks can happen, leading to critical issues. To overcome this issue, new techniques of adding blocks have emerged. These techniques ensure that a fork cannot happen, assuming clear assumptions. Those consensus mechanisms, e.g. *PBFT*, define a set of *validators* to validate blocks and a subset of validators signs each block.

Smart contracts. A feature that has given rise to a strong interest in blockchains is writing *smart contracts* [187]. A contract is a set of promises that recognises and governs duties and pre-specified transaction rules arising from agreements between non-trusting participants, which are enforced by the blockchain's consensus [57].

Nick Szabo first proposed smart contracts in 1994 [172]. Szabo defined smart contracts as computerised transaction protocols that execute the terms of a contract. Years after Szabo's paper, smart contracts were popularised by the Ethereum framework released in 2015 [187]. A smart contract became a digital protocol written in a high-level programming language. For example, *Solidity* [78] is the Ethereum contract-oriented programming language. Each *Solidity* contract is identified by an *address* and holds an amount of *ethers*, Ethereum's cryptocurrency. A contract is an imperative sequential and executable program that runs in blockchains. That program consists of a set of instructions for performing specific actions. It can manipulate functions and variables and invoke other contracts by sending transactions to the target contract address.

The blockchain architecture. The blockchain architecture can be viewed in layers, as depicted in Figure 1.2. The hardware layer can be seen as the layer on which the blockchain system is built. The blockchain's content (the blocks and the transactions) is stored in physical servers located somewhere on earth. In other words, the hardware layer stores the data layer that consists of the elements in Figure 1.1. The network layer represents the communication between the blockchain users. When a block is created in the blockchain, it is propagated to all the blockchain users. This propagation is carried out in a peer-to-peer way across the network layer. As mentioned earlier, blockchain users must perform a consensus mechanism to add blocks to the chain. The consensus layer is in charge of validating the blocks, ordering them and guaranteeing that everyone agrees. This layer is one of the most critical features in blockchains. The last one is the application layer that comprises the programs that end-users use to interact with the blockchain, e.g. smart contracts and decentralised applications (dapp). A decentralised application (dapp) is built on a decentralised network that combines a smart contract and a frontend user interface.

1.1.2 Vulnerabilities of Smart Contracts

Smart contracts have been a hot topic since they emerged in 2015. The benefits they provide have helped popularise the use of blockchain. Smart contracts can be found in many fields, from finance [166] to agriculture [160]. Although there are different contracts on the market, such as *Michelson* [173] and *Chaincode* [21], *Solidity* remains dominant regarding the number of contracts deployed on the blockchain. *Solidity* has undergone an explosion of use but is now a victim of its

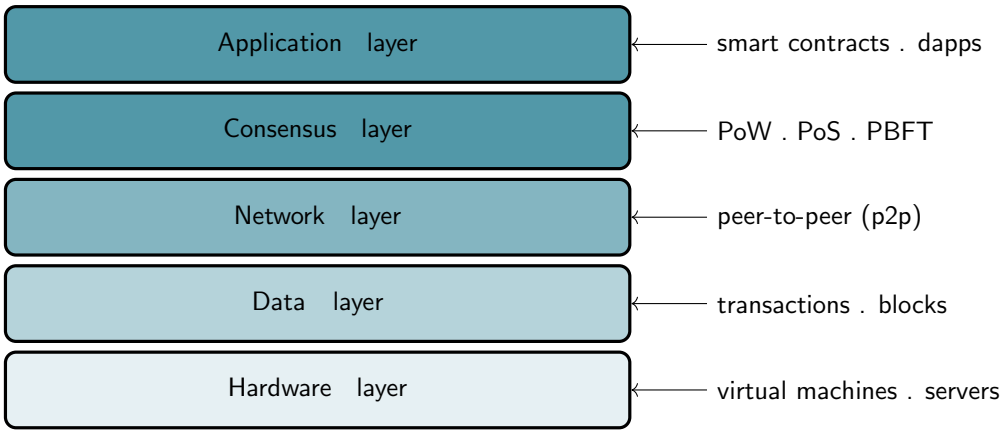


Figure 1.2 – Blockchain layers

success. The increased use of contracts has been at the expense of contract security. Over time, it has become apparent that contracts have several flaws and vulnerabilities. They are often confronted with increasing attacks exploiting smart contract execution vulnerabilities leading to significant malicious scenarios. One of the best-known attacks is the "*the DAO* attack" [24]. A *DAO*, for "decentralised autonomous organisation", is a smart contract deployed on the Ethereum blockchain that operates as a decentralised venture capital fund. The hacker recursively exploited a flaw in the code of "*the DAO*" that allowed the hacker to collect *ethers* in a secondary *DAO* repeatedly. The attack resulted in a loss of 3.6 million *ethers*.

Another example of an attack on smart contracts is the "*Parity Wallet Hack*" [8]. Parity [7] is a company that builds blockchain infrastructure in the Ethereum ecosystem, and wallets are smart contracts that store money. The origin of the flaw comes from a library which is a smart contract itself. This library contract has functions to create multi-signature wallets. Multi-signature wallets are like regular wallets in that they are also smart contracts, but they require multiple approvals to withdraw any amount from the wallet. All multi-signature wallets created are dependent on the library. The hacker took advantage of a loophole in the contract to control the library, making all dependent wallets useless. All the funds stored in affected Parity wallets were no longer withdrawable. The affected wallets had an estimated sum of 500,000 *ethers*.

These examples show that a vulnerability in a smart contract can have serious consequences. Moreover, once published, errors in smart contracts cannot be corrected due to the immutable nature of the blockchain.

In this thesis, we study, in particular, the vulnerabilities of smart contracts written in *Solidity*. The study shows that *Solidity* has different bug causes that increase its vulnerability to attacks.

1.1.3 Interoperability Between Blockchains

Let us go back to Alice and Bob's example, which uses blockchain to perform asset transfers. Suppose that Alice's money is on a different blockchain than Bob's, where property rights are digitised. The question is, how to make the exchange knowing that the two assets are on different blockchains? This issue is becoming common as blockchain technology becomes popular in many areas. As a result, its use has increased considerably in recent years since the creation of many cryptocurrencies and blockchains. The need for communication between the different blockchains has arisen among users. Consequently, the development of infrastructures allowing communication between them has become necessary.

In 1996, Wegner stated that "*interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform*" [183]. Enhancing the interoperability between blockchains seems to be the solution to establishing ways of exchanging between them. The advantage of providing interoperability between blockchains is exploring new functionalities, scaling the existing ones, and creating new use cases, e.g. the use case of Alice and

Bob, where they should be able to transfer their assets from one blockchain to another.

There are several interoperability techniques for enabling communication across blockchains [35]. Some enable exchanging between blockchains of the same family, i.e. the blockchains must be of the same type and are built according to the same rules. Others enable the communication between blockchains that are not of the same family means that blockchains have different rules and operating mechanisms. Depending on the type of blockchain, the system ensuring interoperability will differ. Indeed, a blockchain can be described as *permissioned*, *permissionless*, *public*, and *private*. The characteristic of permissioned/permissionless refers to the users' anonymity, while public/private refers to the participation in the consensus mechanism. This thesis focuses on a system that addresses these issues of distributed applications for trading assets exploiting smart contracts. Recently, one application based on smart contracts has gained popularity, namely the *cross-chain swap* applications. These applications allow users of different blockchains to transfer assets in a decentralised manner and without the involvement of an intermediary. Some *cross-chain swap* applications requires synchrony between users of the system to proceed with the transfer; others do not, i.e. the system can be executed in an asynchronous environment implying that users do not have to synchronise their actions.

1.1.4 Correctness of *Cross-Chain Swap* Applications

A *cross-chain swap* system involves several participants executing actions from the system to achieve a common known goal. However, these systems are complicated to manage because they are distributed and often subject to unintended behaviour, i.e. malicious users. The authors in [193] prove that no asynchronous cross-chain system is tolerant to malicious users unless assuming a trusted third party. A trusted third party can be centralised or decentralised, e.g. another blockchain. The issue that can arise is that a system that claims to be tolerant to malicious users in an asynchronous environment is not. As a result, correct participants in the system may lose out at the end of the system's execution.

Given these issues, applying behavioural correctness to such systems is appreciated. Behavioural correctness is the ability to guarantee that the system is issued as intended, without unintended consequences, e.g. asset lock or asset theft. A way is to ensure that the system (or the system's algorithm [2]) is correct concerning its specification. For instance, one applies formal verification methods to verify the correctness of algorithms according to a specification. However, when we say formal verification, we mean automatic or semi-automatic verification, which involves using verification tools. Besides, it is essential to define a realistic specification for a *cross-chain swap* problem. For instance, several existing *cross-chain swap* specifications include *atomicity* property, even in an asynchronous environment [191]. The atomicity refers to the transfer of all assets or none. However, this property seems not satisfied in a system with malicious participants; thus, atomicity is often questioned.

In this thesis, we present a *cross-chain swap* problem specification along with an algorithm satisfying the specification. We explain how we ensure a safe algorithm assuming the presence of malicious participants in an asynchronous environment without ensuring atomicity.

1.1.5 Formal Methods

Software systems inevitably increase in scale and functionality; the number of subtle errors increases along with complexity. An error or a bug is a common problem that any computer program may encounter. It can occur from poorly writing the program, a typing error or bad memory management. Moreover, some of these errors can become a significant issue and result in catastrophic losses of money, time, or even human life. One can cite the infamous crash of ARIANE 5 [72].

[2] An algorithm characterises a sequence of steps that provides a result from input elements [134]

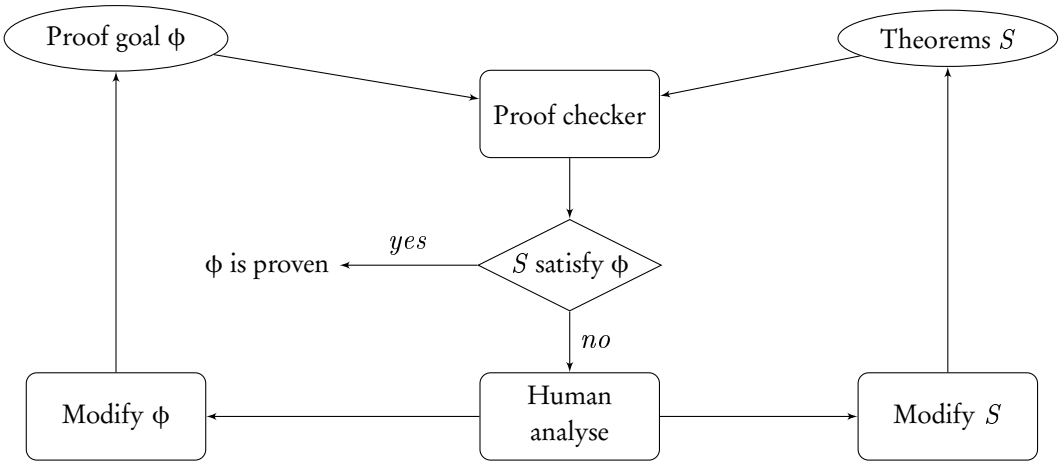


Figure 1.3 – Interactive theorem proving process

Therefore, it is necessary to build systems, particularly critical systems, considering this complexity. Formal methods refer to logical techniques and notations for modelling and analysing complex systems as mathematical entities. Building a mathematical system model and using mathematical proof makes it possible to verify its properties to ensure correct behaviour. They are applied to both sequential and distributed programs.

A property is a characteristic of a program that is true for every possible execution of that program. Properties of interest for distributed programs fall into two categories: *safety* and *liveness*. A safety property asserts that "*nothing bad happens during execution*", e.g. the program does not reach a bad state. Safety properties represent requirements that the system should continuously maintain. They often express invariance properties. Liveness property asserts that "*something good eventually happens*", e.g. the program must eventually reach a good state. Liveness properties represent requirements that do not need to hold continuously but must ensure eventual (or repeated) realisation. Usually, in sequential programs, what is proven is safety properties.

The formal verification of programs has been an active research area since the early days of computer science, and various techniques have appeared since then. We distinguish three main family approaches to verification: *model-checking*, *abstract interpretation* and *theorem proving*.

Theorem Proving

Theorem proving, or automated theorem proving, relies on formulating a mathematical theorem, reasoning, and logic, to prove a set of propositions. It can be used to handle infinite systems. These systems are defined and specified by users in an appropriate mathematical logic. Theorem provers verify the fundamental and critical properties of the system and use techniques for helping a proof. The basic foundation of automated theorem proving is *Propositional Logic* (PL), *First-Order Logic* (FOL) and *Higher-Order Logic* (HOL) [107]. The use of such languages allows to state rigorously a wide range of problems in an unambiguous way. Propositional logic is used to represent atomic propositions with the help of mathematical boolean operators. First-order logic is the extension of propositional logic that allows quantifiers. Predicate logic is the general category to which FOL belongs. Higher-order logic extends FOL by supporting many types of quantification. HOL permits predicates to accept premises (also predicates) and allows quantification over predicates and functions, which is not the case for FOL. However, in this thesis, we use PL and FOL.

Figure 1.3 illustrates the interactive theorem prover framework to construct a mechanically verified proof. An automatic and human-interacting step is conducted consecutively for each proof goal. An interactive proof methodology starts with manual construction of the proof, which concerns describing the proof goal ϕ and providing their corresponding theorems S that are either proved or assumed. Given the proof goal ϕ and a set of theorems S as requirements written in a formal specification language, an automatic theorem prover can be launched as shown in Figure 1.3

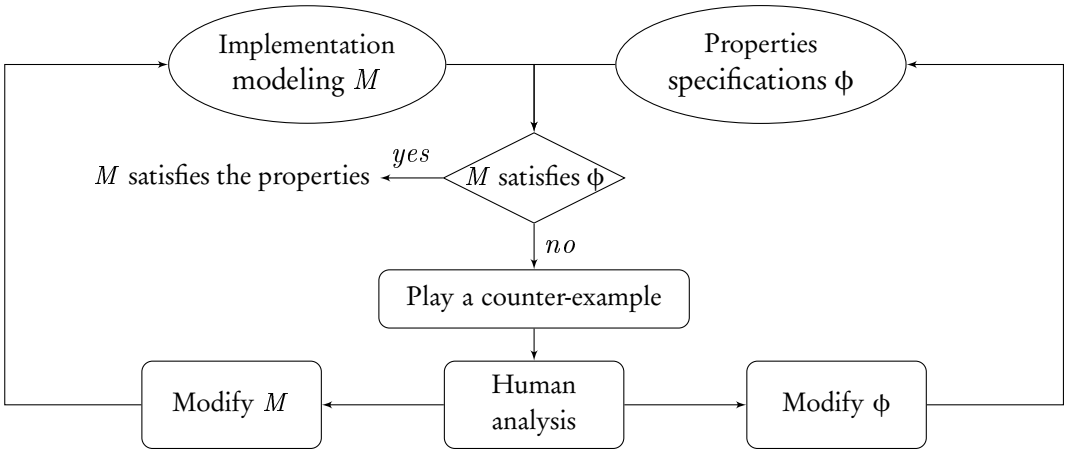


Figure 1.4 – Model-checking process

to automatically deduce the proof using the embedded rules or calculi implemented in these provers. If the proof goal ϕ is mechanically derivable from the set of theorems S, the prover will reply with a proof verifying it. Otherwise, depending on the human analysis, either the theorems or the goal must be modified.

Tools like Dafny [130], Frama-C [64], VeriFast [109], and *Why3* [83] prove programs using this technique, also called *deductive verification*. For example, *Why3* is a platform for deductive program verification. It provides a rich language for specification (logic code) and programming (imperative code) called *WhyML*.

Model-Checking

Model-checking is an automatic formal verification technique based on a description of the behaviour under study in a finite-state machine [58]. This technique performs an efficient systematic inspection of all possible state sequences described by the model. The technique proves if the model satisfies some behavioural properties. The state machine's semantics is given by a system of transitions that can be more or less complex. A system of transitions ranges from finite state machines (finite automata) to real programs (Turing machines). Thus, the main challenge in model-checking is the combinatory explosion of the model. Nevertheless, this technique is relevant to checking partial specifications early in the design process [58]. Figure 1.4 represents the process of the model-checking, which verify whether or not the model M of a system satisfies its specification ϕ written as a logical formula. If the model violates a property under consideration, the model checker provides a counter-example of a sequence that leads to the violation property. The counter-example can be advantageous in adapting the design (or the specification). Indeed, many successful industrial applications witness the performance of model-checking tools [27].

A wide choice of tools performs model-checking, including the TLA$^+$ toolbox [120]. TLA$^+$ uses as model-checker TLC [190] that provides a platform to model-check the specifications written in TLA$^+$ language. Among other existing tools, SPIN [106], ProB [131], UPPAAL [96] and NuSMV [55] can be cited to represent three kinds of model-checkers based on different modelling techniques [85].

Abstract Interpretation

Reliability resolution of computer programs is a well-known problem in software verification. Static analysis of the behaviour of programs during execution provides solutions to this problem. However, this means of analysis can be undecidable and requires some form of approximation. The objective of abstract interpretation is to formalise this idea of approximation [63].

Abstract interpretation is based on abstract semantic reasoning, which is less precise but easier to handle. As a result, some information will be voluntarily lost, resulting from a partial execution

of a computer program. This method gains information about the program semantics without performing all the calculations.

This method is applied to the safety and security of complex hardware and software computer systems. Its main application is the formal static analysis and the automatic extraction of information about the possible executions of computer programs. Such analyses have two main usages: inside compilers, to analyse programs to decide whether specific optimisations or transformations are applicable, for debugging, or even the certification of programs against classes of bugs.

1.2 Contributions and Organisation

This manuscript is organised as follows. Chapter 2 provides theoretical background on the technology used in this thesis, and Chapter 3 presents state of the art on research aspects addressed in this thesis. It gives an overview of what exists to provide a clear scientific position. Chapter 4 provides technical backgrounds such as notations and tools used throughout the manuscript. Chapters 5, 6, 7 and 8 present the technical results of this thesis, and Chapter 9 gives a general conclusion about the different results and directions for future work in the continuity of the results obtained in this manuscript.

In the following, we give a glance at the thesis contributions.

Correct and proven smart contracts. Blockchain systems manipulate cryptocurrency and transaction information through smart contracts. Therefore, if a bug occurs in the blockchain, severe consequences can happen, e.g. losing money. A contract can define any set of rules represented in its programming language (e.g. *Solidity* for Ethereum Blockchain), thus enabling the implementation of decentralised applications. Smart contracts are programs that present vulnerabilities that can be exploited and attacked. It is crucial to ensure safe and correct smart contracts and avoid computer bugs before use. It would be interesting to use formal languages to write, check, and compile such programs and define their properties.

This thesis proposes a language dedicated to deductive verification, called *WhyML*, to be a new language for writing formal and verified smart contracts. The aim is to avoid attacks exploiting such contract execution vulnerabilities. We apply concepts of deductive verification and develop a methodology of proof of smart contracts. The presented method was applied to a use case that describes an energy marketplace allowing local energy trading among neighbourhood inhabitants. The resulting modelling allows a non-trivial trading contract to match consumers with producers willing to trade energy. In addition, this last point demonstrates that with a deductive approach, it is possible to model and prove programs at a realistic scale, thus allowing the verification of more realistic functional properties.

This contribution, detailed in Chapter 5, has been published and presented in the proceedings of a peer-reviewed conference [146].

Description of a *cross-chain swap* algorithm, $\mathscr{P}_{swap}$, and its formal verification. Once stored in the blockchain, any blockchain user can use and call a smart contract, including other smart contracts. Therefore, blockchain applications can exploit stored smart contracts and use them to meet the application's needs. *Cross-chain swap* applications are such systems that use smart contracts to achieve transactions across users. The advantages of this type of system have led to many research articles in recent years. However, most of them are not sufficiently formal and do not reflect reality. In this thesis, we have formally modelled an algorithm that allows cross-chain exchanges between different distributed ledgers called $\mathscr{P}_{swap}$. Indeed, since a distributed ledger is a high-level class of blockchain, the $\mathscr{P}_{swap}$ algorithm abstracts any notion of blockchain so as not to limit its scope to blockchains. Moreover, this algorithm satisfies a realistic specification that considers implementation assumptions never before considered in the literature. Thus, the specification is tolerant to malicious users, without assumptions of proportions, which are most often found in distributed systems. The specification defines a safety and a liveness property that remains satisfied even in an

asynchronous environment. Moreover, the algorithm's description is defined formally, facilitating its understanding and the possibility of verifying its behaviour. This part of the contribution is detailed in Chapter 6. Chapter 7 gives another aspect of the contribution, which is the formal verification of the algorithm. Consequently, we apply the TLA$^+$ tool to the algorithm to prove that it satisfies the set of properties of the *cross-chain swap* specification. We apply the deductive verification method to prove the safety property and model-checking to prove the liveness properties on a model that includes malicious participants.

This contribution and the results have been published and presented in peer-reviewed conference proceedings [147].

Analysis of the *cross-chain swap* $\mathscr{P}_{swap}$ compatibility in a blockchain environment. The specification and algorithm introduced in the previous contribution make implementation assumptions and impose instantiation requirements of the $\mathscr{P}_{swap}$ algorithm. However, some distributed ledgers, such as certain types of blockchains, can not satisfy all these requirements. Therefore, in this thesis, an analysis is made on a set of existing blockchains that may or may not implement the $\mathscr{P}_{swap}$ algorithm. The analysis is based on the characteristic of the blockchain type, whether permissionless/permissioned or public/private, and on their ability to instantiate the requirements of the $\mathscr{P}_{swap}$ algorithm.

This contribution, detailed in Chapter 8, has been partially published and presented in the proceedings of a peer-reviewed conference [147].

Part II

Background

Chapter 2

Basics of Distributed Systems and Blockchain

“ The scientist is not a person who gives the right answers, he’s one who asks the right questions. ”

– Claude Levi-Strauss

Contents

This chapter introduces the notions about blockchain and distributed systems that are necessary and sufficient for understanding this thesis. The first Section 2.1 describes distributed systems and their specific characteristics, such as the failure and communication message model. Section 2.2 gives an overview of blockchain systems and their specificities, such as the different involved participants, the consensus protocols used, and the different types of blockchains.

2.1 Basics of Distributed Systems

A *distributed system* [134] consists of several entities communicating to achieve a common goal using a common *protocol* and appearing as a single entity to the user. A protocol is a set of rules that govern how each entity in a system must operate to achieve the desired outcome [134]. Using distributed systems has several advantages, such as dealing with *fault tolerance* [110]. Fault tolerance refers to the ability of a system to continue operating without interruption when one or more of its components fail (see Section 2.1.3 for more details). Unlike centralised systems that are prone to a single point of failure, distributed systems do not have this problem. Indeed, if a single point of failure occurs, the distributed system will continue to function as the working servers of the system take over, and the fault becomes unnoticeable. Moreover, distributed systems are characterised by concurrency between system entities that operate simultaneously. It refers to the ability of different entities to be executed simultaneously without affecting the desired protocol outcome.

Distributed systems are often more complex than centralised systems that run on a single computer. This complexity of distributed systems arises because different parts of the system are independently managed, and there is no single authority in charge of it. In addition, distributed systems face a lack of observability of the system's state and concurrency. This complexity makes them challenging to study and analyse. While distributed systems can satisfy fault tolerance and continuity of service despite some entities failing, they are also victims of increased potential faults. Indeed, these systems are designed to perform several tasks by adding components, making them more complex and more exposed to failures. A single fault can bring the system down if the system is not correctly designed. Hence, fault tolerance is not automatic, but a system must be designed to be so to ensure the continuity of service.

Finally, scalability in distributed systems also represents a challenge in maintaining a consistent performance as the number of participants increases. Such a system must maintain service stability to ensure its security and reduce its vulnerability to attacks.

2.1.1 Examples of Distributed Systems

Distributed systems are found almost everywhere. For instance, they can be found in web services [165], online games [73], client-server applications [152] and peer-to-peer applications [171]. In the following, we define underlying concepts of distributed systems that are crucial for the reader to follow the rest of the thesis. We define two examples of such systems: the *Two-Phase Commit* protocol [37] and *Distributed Ledgers* [136]. We have chosen these examples because they will be studied in greater depth in the following chapters.

The *Two-Phase Commit* Protocol

The *Two-Phase Commit* protocol is a distributed synchronisation algorithm that solves the *atomic commitment problem* [97]. The protocol ensures that a transaction either *commits* at all the participants or *aborts* at all of them. In other words, the "commit" result allows the transaction to occur, while the "abort" result enables the transaction to be aborted so that it does not occur. Essentially, the protocol is used when a set of participants wish to update a distributed database by sending transactions [92]. There is a need for synchronisation among participants to achieve atomicity, ensuring a unanimous outcome for each distributed transaction regardless of failures. In distributed systems, synchronisation is achieved via *clocks* [122] so that the participants can obtain a common notion of time. Synchronised clocks are used to realise some behaviours that need to be executed in

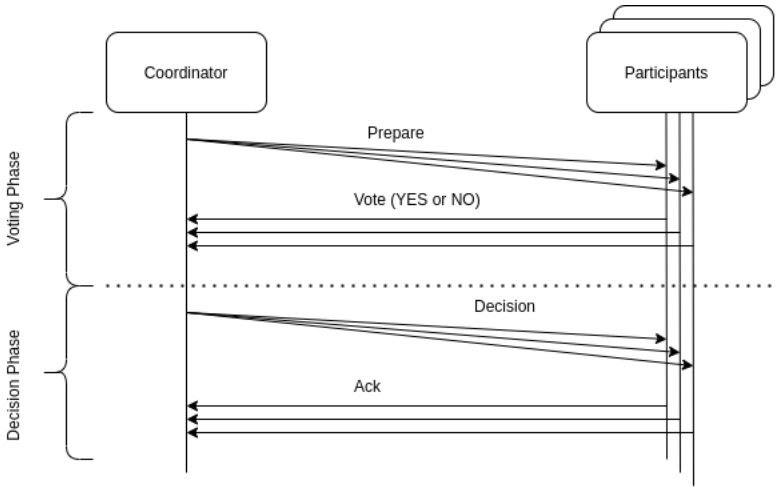


Figure 2.1 – Phases of the *Two-Phase Commit* algorithm

a known time range (see Section 2.1.4 for more details). As a result, the *Two-Phase Commit* protocol ensures that a transaction to a distributed system (or database) is executed atomically while being fault-tolerant.

The *Two-Phase Commit* algorithm comprises two different types of entities: the *coordinator* and the *participants* (or *followers*). The coordinator has the role of managing a transaction to *commit* or to *abort*, and the participants are those who will generate transactions. As the name of the algorithm implies, The *Two-Phase Commit* is divided into two phases (see Figure 2.1):

1. *Voting Phase.* The first phase is when the coordinator sends a query to commit a transaction to all the participants. The request is given through the *prepare* message. After the sending message, the coordinator waits for a reply. On the participants' side, they receive the *prepare* message and have to give the coordinator a response, either a *yes* vote if they agree to commit the transaction or a *no* vote if they do not agree.

2. *Decision Phase.* Once all the participants have given a response, the coordinator can decide. If all participants answered the coordinator with *yes*, the coordinator sends a *commit decision* message to all participants. If at least one participant votes *no* for the commit, the coordinator sends an *abort decision* message to all the participants. Once the participant receives the decision message (either commit or abort), it sends an acknowledgement (*Ack*) to the coordinator. The coordinator completes the transaction when all acknowledgements have been received.

For example, the acknowledgement step (*Ack*) makes it possible to detect if one of the participants has failed from a crash. Therefore, the coordinator only considers the transaction to be committed or aborted if it receives as many acknowledgements as the number of participants. However, this step leads to the drawback of latency. Since the coordinator waits for all the acknowledgements, a single slow participant will slow down the transaction process. Moreover, the *Two-Phase Commit* algorithm may encounter a "blocking problem". Suppose every participant votes "yes" for the transaction commit, and the coordinator fails (due to a crash, for example) before sending the decision message. In that case, the participants will be blocked as they await the coordinator's decision.

Distributed Ledger Technology

A distributed ledger is a replicated, shared, and synchronised digital database spread across a set of participants [49]. It enables the secure functioning of a decentralised digital database. It can be

seen as a ledger of transactions maintained in a decentralised form without the need for a central authority. All participants having access to the ledger have the same view; hence any changes or additions made to the ledger are known to all. A distributed ledger is more resilient to attacks than a centralised ledger because for an attack to be successful, most of the distributed ledger servers must be attacked.

Underlying distributed ledgers is the same technology used by *blockchain* [144], which is a distributed database recording information about transactions. When we mention distributed ledgers, we often think of blockchain, but it is just the most famous type of distributed ledger. In the next section, we define blockchain technology, but one can cite other distributed ledgers not backed by a blockchain, e.g. *Tangle* [159] and *Hedera* [29] [1].

Tangle [159] is the distributed ledger of the *IOTA* cryptocurrency created for the *IoT* industry [90]. IOTA does not use a chain of blocks like Bitcoin (the first deployed blockchain) but a DAG (Distributed Acyclic Graph), also called *Tangle*. The DAG consists of nodes and edges. Each node represents a single transaction called a "*site*" connected to other sites via edges. A site contains all transactions details, such as the sender, the receiver and the number of coins. A transaction, or a site, must be connected to at least two other transactions. This connection by edges validates those two transactions. This referencing of transactions is considered an approval and indirectly that a subsection of the Tangle is valid and compliant with the Tangle's protocol rules. A transaction that does not have incoming edges is unconfirmed and cannot yet be trusted.

Adding a new transaction to the Tangle requires connecting it to two unconfirmed transactions selected randomly by an algorithm. By adding the new transaction, the two unconfirmed transactions are now verified. This approach is very scalable since every added transaction confirms two others. The more transactions are added, the more unconfirmed transactions become verified. Tangle uses a weight attributed to the site/transaction proportional to the level of trust in the transactions. This weight represents the amount of work a user has done to generate this transaction. The higher the weight, the more time the user spends validating that transaction. Moreover, each transaction has a cumulative height representing the sum of its weight and that of all those transactions. The transactions with high cumulative weight are older, so we can trust those transactions more than others. As a result, Tangle is an open-source framework that is scalable and permissionless.

Hedera [29] is a public distributed ledger for building and deploying decentralised applications and microservices. Hedera was created as an alternative to blockchain. It has a native cryptocurrency *HBAR* and is structured as a *DAG* based on a *Byzantine fault-tolerant* protocol [125] (see Section 2.2.4). Hedera's network is atop the Hashgraph consensus algorithm [28]. Hashgraph uses "gossip" to share information and establish consensus. A gossip protocol [39] works on the same principle as information sharing on social networks, i.e. the information is spread among the users.

A node of the DAG shares its information about some transactions with multiple other random nodes via gossip. Each gossip message contains information about one or more transactions and is sent to network nodes. The nodes of the DAG combine all newly received information about the transactions and obtain aggregated information. The latter is then sent to other random nodes. The protocol continues similarly until all nodes have the complete information about all the transactions created at the beginning. The history of how the information is related to each other is called a *gossip about gossip*.

2.1.2 Participants

A distributed system consists of processes or participants identified by a unique identifier. Participants run the protocol within the distributed system and execute the action they are supposed

[1] Note that these two types of distributed ledger do not fall within the scope of our study; we introduce them to provide an overview of how these alternatives to blockchains work.

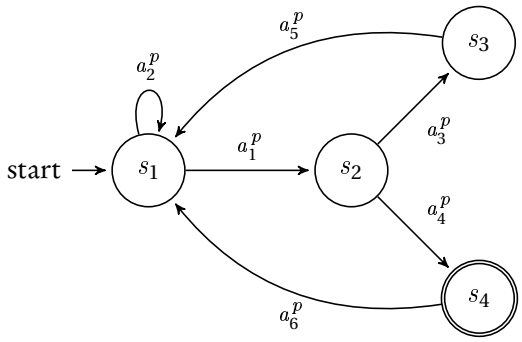


Figure 2.2 – State machine of participant p

$$
\begin{aligned}
Q_p &= \{s_1, s_2, s_3, s_4\} \\
\Sigma_p &= \sum_{i=1}^{6} a_i^p \\
\delta_p &= \left| \begin{array}{l} s_1 \times a_1^p \longmapsto s_2 \\ s_1 \times a_2^p \longmapsto s_1 \\ s_2 \times a_3^p \longmapsto s_3 \\ s_2 \times a_4^p \longmapsto s_4 \\ s_3 \times a_5^p \longmapsto s_1 \\ s_4 \times a_6^p \longmapsto s_1 \end{array} \right. \\
q_{0_p} &= \{s_1\} \\
F_p &= \{s_4\}
\end{aligned}
$$

Table 2.1 – Elements of the participant p's state machine

to carry out. They communicate through messages sent across the system's network, where each message has a unique identifier. The participants have a local clock [122] that allows them to order the occurrence of events and know when these events occur. Sending or receiving a message is an event for our participants. Local clocks use logical time rather than physical time (i.e. real-time) to order events, i.e. they assign them a number corresponding to their occurrence's time. The participants are said to be *synchronous* if they all take the same amount of time to execute an action; thus, their local clocks tend to be synchronised. The participants are called *asynchronous* if their time to execute an action is unpredictable. There are no bounds on the participants' execution speed and no bounds on clock drifts. Clock drift is when a clock does not run at the same rate as a reference clock.

A participant's behaviour can be formally described as an *input/output automaton* [134] (a type of state machine). By running their state machine, the participants execute each action at a time. The execution of action can change the state of the state machine. A state machine is defined by the following elements $(Q, \Sigma, \delta, q_0, F)$:

- Q: a non-empty finite set of states;
- Σ: a non-empty finite collection of internal, input and output actions;
- δ: the transition relation from one state to another as caused by an action in Σ. The set of all transition is $\delta : Q \times \Sigma \to Q'$.
- q_0: the non-empty set of initial states.
- F: the set of final states (possibly empty), where $F \subseteq Q$.

The participants run their state machines sequentially. When a transition is activated, the participant performs an action that allows it to change state. Figure 2.2 is a state machine example that represents the behaviour of the participants p. Its elements $(Q_p, \Sigma_p, \delta_p, q_{0_p}, F_p)$ are defined in Table 2.1. The participant p has four states defined by Q_p, with s_1 its initial state and s_4 its final state. p can change states by performing any action from Σ_p thanks to the relation transitions defined in δ_p.

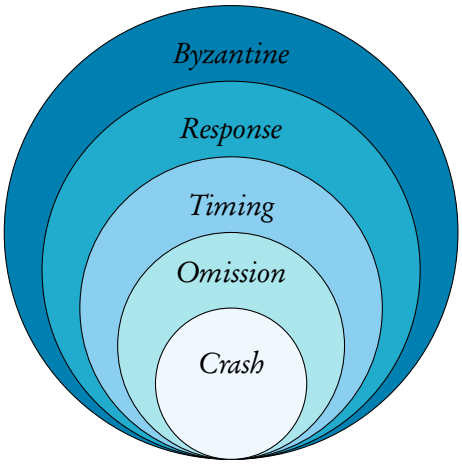


Figure 2.3 – Classification of failure types

2.1.3 Failure Model

The appearance of failures in any system is inevitable. A distributed system comprises many components working together to complete a task. As the system gets more complex and has more components, failures will increase [110]. *Faults* and *failures* are often confused, but they do not have the same meaning. A fault is the incorrect internal state of the system, whereas failure is the inability of the system to complete a task. Faults will lead to failures if they are not correctly handled on time. A *faulty* participant is defined as a participant that does not follow its protocol. A distributed system can have several types of failures, but the most common failures are *Crash* failures, *Omission* failures, *Timing* failures, *Response* failures and *Byzantine* failures (see Figure 2.3).

Crash failure occurs when a component (or a participant) crashes. The faulty participant will follow its protocol and actions correctly and then suddenly stop following it. Omission failure occurs when a participant does not receive incoming requests from the client or fails to send messages in response to the client's request. Timing failure occurs when a participant fails to respond within a particular time frame. Response failure occurs when a participant sends an incorrect message in response to the client's message.

Finally, we have the arbitrary type of failure – the Byzantine failure. Arbitrary failure occurs when a participant sends an arbitrary message. It is the most general form of failure and encompasses all types of failures, making it difficult to manage because we cannot predict how the participant will behave. Therefore, it is challenging to study, and design protocols subject to these failures, also called Byzantine attacks. The solution is to introduce fault tolerance in risky systems, thus allowing the system to continue functioning when a failure of any kind occurs.

The cited failures do not have the same severity level, and there is an ordering among them. For example, if we know that a set of participants might show omission failures, we can assume that the participants will also show crash failures. Furthermore, a participant who might show a timing failure might show an omission failure as well and so on. More formally: Byzantine failures $\supset$ Response failures $\supset$ Timing failures $\supset$ Omission failures $\supset$ Crash failures.

Definition 2.1. (*Correct Participant*). A participant who never fails in the system is said to be *correct*.

A correct participant will always follow its protocol and never deviates from it.

Definition 2.2. (*Byzantine Participant*). A *Byzantine* is a participant for whom nothing can be assumed about its behaviour.

A Byzantine participant can behave in any imaginable way, e.g. it can delay or modify its messages but never that of others participants. It can cause, among others, all failures from Figure 2.3. Note that a Byzantine participant can also follow its protocol.

2.1.4 Messages Communication Model

Participants in distributed systems use messages to communicate, exchange data, and synchronise their actions. The model of messages transmissions plays an important role in distributed systems. However, communication in distributed systems presents complex challenges like the unreliability of communication on a large scale. Messages can be lost, duplicated or delayed because of communication failures. Techniques and protocols have been designed to solve these problems and guarantee reliable message delivery. In addition to the assumption of communication reliability, the communication model can be classified according to assumptions of message transmission delay. A message can be transmitted synchronously or asynchronously, and messages may be received in the same order they were sent or out of order.

When a participant sends a message, the delivery of that message is not instantaneous because of the message transmission delay. Let us define $\Delta \geq 0$ as the maximum message delay between two participants. Based on the value Δ, we define three communication models: the *synchronous communication*, the *asynchronous communication*, and the *semi-synchronous communication*:

- *Synchronous communication.* In synchronous communication systems, the value Δ is finite and known by the participants. Correct participants receive sent messages at the latest at $t+\Delta$, with t the time when the message was sent. A Byzantine participant can delay its message by at most Δ.
- *Asynchronous communication.* In asynchronous communication systems, the value Δ is unbounded. That means that there are no assumptions about the message transmission delay. Therefore, a Byzantine participant can delay the delivery of its message by any finite amount of time. It is assumed that all correct participants will eventually receive the message of other correct participants.
- *Semi-synchronous communication.* Semi-synchronous communication is a trade-off between the two previous models. In semi-synchronous communication systems, Δ is bounded but not known by the participants.

2.1.5 Common Properties of Distributed Systems

A property is an attribute of a system that is true for each of its possible executions. A set of properties specifies any distributed system problem, and for evaluating the correctness of a system, the properties belonging to the set must be satisfied. Properties of interest for distributed systems fall into: *safety* and *liveness*. Any specification can be expressed regarding liveness and safety properties [124]. Informally, safety is a property that guarantees us that "*nothing bad will happen*", and liveness guarantees us that "*something good will eventually happen*". In other words, safety is concerned with a program not reaching a bad state and liveness is concerned with a program eventually reaching a good state.

Safety. Safety properties represent requirements that the system should continuously maintain. They often refer to *invariant* properties that are required to be always true. A safety property does not ensure termination, but all terminating computations produce correct results. Therefore, it can be qualified as partial correctness. Some safety property examples are deadlock freedom (lack of a blocking state), first-come-first-serve and mutual exclusion. The example of *mutual exclusion* property states that only one process is allowed to execute the critical section at any given time. In first-come-first-serve, the property states that requests are served in the order they were made. Unlike liveness properties, if a safety property is violated, a finite execution always shows the violation.

Liveness. Liveness properties require the system to progress and guarantee termination. The *progress* property asserts that it is always the case that at least one action is eventually executed.

If we consider a system with a failure point, the property expresses that a subset of the participants eventually progresses. Another example of liveness property is *starvation-free*. This property is stronger than the progress, and it guarantees that all participants must eventually progress in the system.

The intersection of partial correctness (safety) and termination (liveness) gives total correctness [15].

2.2 Blockchain Overview

This section aims at providing the reader with enough information about blockchain without too much detail. We discuss the main participants in a blockchain and how virtual ownership of assets is managed. Furthermore, we define the different consensus mechanisms and types of blockchains.

2.2.1 Participants in Blockchain

We can identify two distinct types of participants in a blockchain; the *block validators* and *participants* connected to the blockchain. A block validator participates in the consensus by verifying transactions and validating blocks added to the blockchain. The participants get involved in the operation of the blockchain by generating transactions. Once connected to a blockchain, a participant is assigned to a *wallet*. A blockchain wallet allows participants to store, manage and transfer their cryptocurrencies. Both types of participants (regular participants and validators) maintain the reliability of the blockchain by having its complete history from the genesis block (i.e. the very first block) to the current block.

A participant is identified by two keys, a *private* and a *public key*. A public key is known information, like an address, through which anyone receives transactions. A private key is a key that unlocks the right for the participant to spend the associated cryptocurrencies or tokens, proving its ownership. Since the cryptocurrencies are stored in the wallet, the private key is the only way to unlock the wallet and access the cryptocurrencies. Therefore, the private key should remain private and never be shared. It is assumed that the keys cannot be compromised.

2.2.2 Asset's Ownership in Blockchain

In this section, we are interested in looking at cryptocurrency asset ownership from a legal point of view in a blockchain environment. In the real world, the transfer of the ownership of physical assets is done by applying national laws. Property law enumerates thoroughly how ownership may be transferred from one party to another. Agreements must be established between the current asset owner and the future owner [128]. However, transferring and managing cryptocurrency assets in a blockchain are not done through intermediaries as they can be in real life. Transferring assets is independent of any legal requirements, and there is no guarantee that these requirements will be established for blockchain transfers. Transferring cryptocurrency assets between participants is possible by achieving a consensus or validating pre-defined smart contracts rules. As a recall, smart contracts are computer programs deployed in the blockchain that allows transaction rules to be set. What proves the ownership of a cryptocurrency asset is the concept of correct public and private keys that are associated with the transferred asset. The private key is used to prove the legitimacy of a cryptocurrency and acquire it. Therefore, when a participant owns cryptocurrencies, it owns a private key.

The blockchain protocol guarantees specific requirements about asset ownership that may limit the wrong actions of malicious participants. A cryptocurrency asset has a unique owner that can use it through a unique private key; thus, a participant who owns assets can use them at will. In addition, a participant cannot claim to own an asset if it is not the case, thanks to the validation

operation. If a participant claims to own an asset that it does not have and attempts to send it, the transaction validation process at the consensus level will refuse the transaction. As the blockchain records all transactions since its inception (the genesis block), it is easy to verify whether or not a participant owns an asset.

As we can see, this technological solution does not need the intervention of notaries, lawyers or any legal institution like banks or insurance. Moreover, it does not need any legal agreement document. In this sense, the depiction of the "*code is law*" seems to be entirely appropriate. The "*code is law*" implies that the technology scrupulously applies the code. Hence, it does not consider mistakes, fraud, or improper threats, because these are not part of the protocol. Therefore, a mistaken transfer would be effective from a technological point of view. Hence, it is possible to have fraudulent actions made by Byzantine participants that can be seen as valid actions. To illustrate the case, suppose a hacker (Byzantine participant) has stolen a participant's private key and appropriates its corresponding crypto-assets. The hacker wishes to transfer the assets to its public key (i.e. its wallet). Legally, this transfer should be invalid, given that the legal owner of the asset has never agreed to it. However, since the hacker owns the private key, the hacker becomes the rightful owner of the stolen assets from the blockchain perspective. Therefore, a hacker who wishes to transfer dishonestly obtained assets to its address will not be considered Byzantine by the validators. This private key gives the Byzantine participant the real power to dispose of the asset even though there was no legal basis for the transfer. As a result, the transaction will be validated, and even though the hacker illegally possesses crypto-assets, it can dispose of them.

Such a situation will be difficult to undo because when the validators accept a transaction, it will be added to the blockchain permanently. The only possible way to invalidate a transaction is if most participants of the blockchain vote for a *hard fork* – a permanent divergence from the current blockchain version (see Section 2.2.5). However, a hard fork is not feasible except for the most extreme and rare cases, such as discovering a significant hack that corrupts a vast number of transfers. For all other purposes, undoing a transfer is impractical.

2.2.3 Asset Tokenisation

The initial purpose of creating the blockchain was to transfer cryptocurrencies. Nowadays, blockchain use is not limited to transferring virtual currencies and other crypto-assets but can also transfer objects of the physical world, such as gold, land, or houses, using *Asset Tokenisation*. Therefore, the tokenisation of real-world assets has expanded the blockchain's application areas. *Tokenisation* refers to the digitalisation of a real-world item into a token. Thus, asset tokenisation is when an issuer creates digital tokens on a distributed ledger or blockchain to represent a physical asset. Asset tokenisation depends on the legally enforceable linkage between token and asset. Sound legal structuring enables the holder of a tokenised asset to have a legal claim on the physical asset itself. If an asset is truly tokenised, the token owner has a clear legally-supported claim on ownership of the asset. Owning the whole set of tokens (as tokens are often divisible) that correlate to an asset means that the owner wholly owns the asset without any restrictions. Thereby, blockchain guarantees that once a participant buys tokens representing an asset, no single authority can erase or change its ownership which remains entirely immutable [192].

2.2.4 Consensus Protocols

The notion of consensus in a blockchain is essential to guarantee that participants observe the same blockchain state view. Moreover, through the consensus protocol, participants agree on validating transactions. The consensus of blockchain is that all participants must maintain the same distributed ledger and guarantee the system's stable operation. A suitable consensus protocol guarantees the fault tolerance and security of the blockchain systems, including Byzantine participants. The consensus protocol must fit the blockchain; therefore, depending on the type of blockchain, the consensus protocol will be different regarding the *finality* of added blocks. The *finality* affirms that

no well-formed block will be revoked once added to the blockchain. Hence, consensus protocols currently used in most blockchains can be divided into two categories: the probabilistic-finality consensus protocols; like *Proof-of-Work* (*PoW*), *Proof-of-Stake* (*PoS*) and *Delegated Proof-of-Stake* (*DPoS*), and the absolute-finality consensus protocols; like *Proof-of-Authority* (*PoA*) and *Practical Byzantine Fault Tolerant* (*PBFT*). In the following, we introduce some popular blockchain consensus protocols.

Proof-of-Work (PoW). The *PoW* algorithm is the first consensus protocol used in blockchains, including Bitcoin [144]. This protocol requires specific participants, called *miners*, to solve a difficult cryptographic problem to add a new block to the blockchain. The miner applies a hashing algorithm to find the result, and the first miner solving this problem will be the next to create a block and add it to the blockchain. The difficulty of mining is readjusted by the network every 2,016 blocks [44]. Taking part in the consensus by solving the problem uses considerable computing power. This protocol tends to demand more computing power from the miners, who have to make several attempts to find the correct result. However, it can require less computing power to keep an average time of 10 minutes between each block creation [86]. This calculation method is intended to deter malicious participants from attacking the system, such as *denial-of-service* [53] or *Sybil attacks* [71]. A Sybil attack is when a malicious participant creates multiple pseudonyms to influence or control the system.

A *PoW* validation system is cost-effective and gives a high level of security. This system is complex to produce and costs the one who performs the calculation a consequent computer processing power resulting in time and energy consumption. This problem is designed so that the work involved in solving it must be difficult to achieve (in terms of computing power and energy) for the applicant but easily verifiable by a third party. Therefore, everyone feels incentivised to work towards the proper functioning of the network. However, the massive energy consumption remains the most significant disadvantage of the *PoW*. Thereby, in order to reduce this drawback, more efficient and much less energy-consuming protocols have been proposed, such as *PoS*, *DPoS* and *PoA*.

Proof-of-Stake (PoS). The process of *PoS* [178] is that each network participant must prove that it has a particular share of the circulating supply currency if it wishes to take part in block validation. The network protocol will then choose to delegate the validation of a new block to one of the network participants according to an algorithm taking into account a few criteria such as the age of the coins owned or the quantity of owned coins. Under a *PoS*-type consensus mechanism, the probability for a participant to be selected to validate a new block corresponds to its percentage of ownership (its "stake") in the circulating supply. Unlike *PoW*, *PoS* does not need considerable computing power to solve the consensus; thus, it is less energy-consuming.

Delegated Proof-of-Stake (DPoS). The difference between the classic *PoS* mechanism and the *DPoS* mechanism [189] is that in a *DPoS* system, participants of the network vote and elect delegates to validate the next block. Delegates are also called block producers. When performing the validation of the transactions, delegates sign each of the new blocks with their private key. Thereby, they guarantee the data inviolability in the ledger and recover the costs of the transactions entered in the block. Compared to *PoW*, *DPoS* is less energy consuming and has a better transaction throughput.

Proof-of-Authority (PoA). This consensus method allows a limited number of participants to participate in the validation of transactions and blocks [66]. One or more validators are responsible for generating each new block of transactions included in the blockchain. The new block does not necessarily need validation to be accepted. Therefore, *PoA* consumes much less energy than *PoW* since there is no complicated computation for validating transactions.

Practical Byzantine Fault Tolerant (PBFT). When an algorithm solves the *Byzantine Generals' Problem* [125], the algorithm is said to be *Byzantine fault-tolerant* (*BFT*). The Byzantine generals' problem is a metaphor that questions the reliability of transmissions and the integrity of interlocutors. Therefore, the question is how and to what extent it is possible to consider information whose source or transmission channel is suspect. The solution involves the establishment of an appropriate algorithm that must be tolerant of components that may be malicious. Thereby, *BFT* is a mechanism to reach consensus even when some system participants are malicious (or Byzantine).

Practical Byzantine fault tolerance is an example of such an algorithm. It is a consensus mechanism proposed by [54] that constitutes the first practical solution to the problem of the Byzantine generals' problem in reaching consensus despite Byzantine faults. This mechanism can withstand up to 'f' Byzantine faults if and only if the network consists of at least '$3f+1$' participants. The *PBFT* algorithm in blockchain inherits many concepts from its version used in classical distributed systems. A set of validators are allowed to take part in the consensus protocol. An elected leader creates an ordered list of transactions broadcasted to other validators, who execute them. After the transactions' execution, validators compute the hash code for the new block, which is then broadcast to their peers. If two-thirds of the received hash codes are the same, the block is committed and added to the blockchain.

2.2.5 Forks

Probabilistic consensus mechanisms, like *Proof-of-Work* [144] and *Proof-of-Stake* [178], can generate *forks* when adding blocks to the chain. When a blockchain is said to be forked, there is a divergence in the structure of the chain. As a result, two participants in the network may have different views of the blockchain and, therefore, not the same version. There are three types of forks: *hard forks*, *soft forks*, and *temporary forks*. A hard fork is a permanent divergence from the previous blockchain version. It is a fork intended by the blockchain community. It occurs when there is a change in the blockchain system, e.g. adding new features. The result of the fork is the creation of two completely separate versions of the blockchain, for example, classical Bitcoin with Bitcoin cash [82, 117]. Soft forks are generally used to implement software updates that do not require a separation of the blockchain. The changes remain compatible with earlier versions of the system. A temporary fork is when two or more blocks have the same height, as depicted in Figure 2.4. The blockchain starts with b_0, the genesis block with a height of 0. Block b_i has two children b_{i+1} and b'_{i+1} of the same height $i+1$. Temporary forks occurrence is a crucial issue for adopting blockchain technologies in critical applications.

This type of fork occurs in public blockchains using probabilistic-finality consensus protocols that allow all participants to participate in the consensus to validate transactions. This situation is usually resolved quickly by respecting, for example, the rule of the longest chain [82]. The rule that participants adopt the longest chain of blocks allows every blockchain participant to agree on what the blockchain looks like and agree on the same transaction history. The longest chain is the one that took the most effort to construct.

Consensus mechanisms such as *PBFT* or *PoA* can avoid the appearance of forks in their blockchain because of absolute-finality consensus protocols [66].

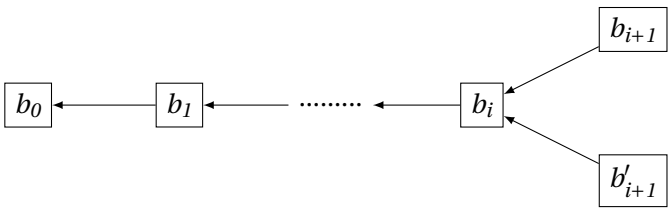


Figure 2.4 – Temporary fork of a blockchain

2.2.6 Types of Blockchains

Blockchain technology has continued to evolve since its emergence in 2008 [144]. It has attracted much interest from many people, and the application areas are increasing.

However, blockchain's definitions are not standardised due to its rapid evolution, and only a few documents formally describe blockchain aspects [20]. This section will try to overview the different types of blockchain as precisely as possible. The sources come mainly from white papers [94, 164], blogs and websites like 101Blockchains [1], medium [6] and FOLEY [5]. In the following, we will detail the interest of each type of blockchain by citing examples of applications.

The Limitations of First-Generation Blockchain

Bitcoin [144] was the first generation of blockchain technology, and at its inception, it was seen as a real technological innovation with great promises. The idea of decentralised and encrypted currencies that Bitcoin brings increased its popularity. Bitcoin was created to carry out cryptocurrency transactions by anonymous participants without a central controlling body. The Bitcoin blockchain technology has several qualities, such as data immutability, that have attracted the private sector, including companies. Ethereum [187] is also a first-generation blockchain. It is an electronic platform that allows people to transfer cryptocurrency, *ethers* (Ethereum's currency), and build decentralised applications.

Enterprises took an interest in this technology and wanted to take advantage of it. However, the first generation of blockchains had multiple drawbacks, including inefficiency and unscalability. Moreover, Bitcoin and other first-generation blockchains are public blockchains; anyone can join the network and do transactions. While this is part of the reason for Bitcoin's success, this feature is not suitable for everyone and all applications. For example, enterprises or organisations will be concerned about the public aspect of blockchains like Bitcoin or classic Ethereum due to business confidentiality [157]. For instance, they may have critical data that must be kept private from competitors. Banks, for example, deal with loads of transactions every day, so there should be no scalability problems. Therefore, a Bitcoin-based blockchain would not be suitable for their use cases. In addition, another drawback of Bitcoin is its energy consumption since it uses *PoW* to validate transactions. In its early days, computing tasks did not consume as much energy as today. The difficulty of the calculation increased with time, as did the amount of energy needed for the calculation. This inefficiency makes it unsuitable for any system that needs to stay efficient no matter what.

Creating other blockchains was inevitable to solve these issues because the use cases have evolved, and the needs have changed. In the early days of blockchain, with Bitcoin and Ethereum, it could only be characterised as *public permissionless* blockchains. Today, the evolution of this technology means that there are, by contrast, additional characteristics called *private* and *permissioned*, which are described below. Each of them has its specificities to solve a particular problem or set of problems; however, all types of blockchain have one goal in common: carrying out transactions and exchanging information through a secure network. The types of blockchains are differentiated according to the participants' anonymity (*permissionless* and *permissioned*) and their participation in the consensus protocol (*public* and *private*). The permissionless characteristic gives anonymity to participants, where a public key identifies them. No permission is required to join the network, and participants' rights are not restricted. Conversely, the permissioned characteristic requires the identification of participants and asks for permission to join the network. Permissionless and permissioned characteristics indicate how the network will perceive the participants. The possibility of participating in the consensus is assessed according to the *public* and *private* characteristics. The public feature implies that everyone can participate in the consensus process. In contrast, the private feature is more restricted. Only a group of participants will have the authority over the network. Thereby, by combining these two groups of characteristics, we obtain the following four types of

blockchain: *Public permissionless*, *private permissioned*, *public permissioned* and *private permissionless*.

We can define the public permissionless and private permissioned as classical blockchains. The remaining two types can be described as *hybrid* blockchains that benefit from the most valuable aspects of the classical blockchains. The hybrid blockchains support many customisation options and can be modified according to needs.

Public Permissionless Blockchains

The first generation of public blockchains, like Bitcoin and Ethereum, consists of permissionless distributed ledgers. Blockchain technology has become known through this category. Its main feature is that anyone can access the network making it an open environment. Moreover, they access it completely anonymously without following any rules or regulations. Each participant has a copy of the ledger, and the only requirement to get it is a good internet connection and a computer. A public permissionless blockchain is non-restrictive since all participants have equal rights to read and write in the ledger. Writing in the blockchain means sending and validating transactions, and reading means having free access to the blockchain transactions. The participants can check the validity of each recorded transaction. These benefits offer public blockchains the possibility of being entirely decentralised, transparent and trustless, i.e. there is no need for intermediaries. A public permissionless blockchain gives a high level of data immutability. A block cannot be modified or deleted, and no one can tamper with the system or rob the money. Suppose someone tries to tamper with the blocks like *double-spending*: all the other participants will reject the transaction. *Double-spending* is the risk that a cryptocurrency can be used more than once.

Because of their anonymous characteristic, public permissionless blockchain can attract malicious participants. However, it is possible to prevent fraudulent behaviour through a high decentralisation and a high number of correct active participation. The more correct participants there are in the network, the more difficult it will be for malicious participants to attack. The result is increased network security. Nevertheless, the network must employ additional verification mechanisms to increase security further. The transactions' validation is done through consensus methods such as *PoW*. The first participant to complete the calculation will be rewarded by the blockchain. In addition, to complete a transaction with *PoW*, a certain fee to pay is often included in the transactions. The fee can increase significantly due to the pressure of participants requesting transactions. These fees and rewards require the public permissionless blockchain to have a cryptocurrency.

One of the main drawbacks of a public permissionless blockchain is a slow transaction validation. Since anyone can send a transaction, too many sent transactions may slow down the network. Reaching consensus on the status of many transactions takes time due to the calculation during the *PoW*. This issue impacts blockchain efficiency, taking a few minutes to hours before a transaction is validated. For instance, Bitcoin can only manage seven transactions per second compared to a centralised payment processor such as Visa, which can execute on average 56,000 transactions per second [174]. There are also limits on the number of transactions entering a block, making the transaction validation slower. Some engaging solutions improve the validation efficiency; for example, Bitcoin works to lighten the network by taking transactions off-chain to make the Bitcoin network faster and more scalable [158].

Private Permissioned Blockchains

This type of blockchain has emerged to meet business needs. It is designed to help companies develop their private blockchain and provide them with means of safely and securely exploiting blockchain technology. It gives complete privacy where information, e.g. specific transactions, can be secured and private. Also called *consortium* or *federated* blockchain, this kind of blockchain is

where a group of participants, the organisation, controls the system. In a private permissioned blockchain, some aspects of the network can be public, while others remain private. This feature allows companies to take advantage of blockchain technology without making everything public. A private permissioned blockchain is a closed network that offers a restrictive environment where the organisation chooses pre-selected known participants having the authorisation to enter the network.

Since a group of participants control the blockchain, the authority is partially decentralised. They are the only ones having complete access and rights to the network, i.e. writing, reading and validating transactions. The set of participants who will contribute to the validation of transactions, i.e. the consensus, is determined beforehand. Unlike public permissionless blockchain, where the consensus is built in the system, in private permissioned blockchain, a consensus can easily be customised. For example, switching from a *PoW* consensus to a *PoA* is easy. However, *PoW* consensus is not engaging in this type of blockchain. There is no need to encourage participants to maintain the network as the organisation takes care of this.

For a participant to join the network, it must follow specific rules and regulations established by the organisation. All these restrictions considerably reduce the number of participants in the network. The low number of participants and validators makes it more scalable and faster to reach consensus than public permissionless blockchains. As a small group of participants validates transactions, the number of participants joining the network does not affect the speed and efficiency of the validation. Hence, speed and efficiency always remain the same, resulting in high efficiency and low energy consumption. In addition, it does not suffer from high transaction fees as a public permissionless blockchain. As only a handful of people can request transactions, there is not any form of delay, and the fee remains the same. There is no requirement for a private permissioned blockchain to hold a cryptocurrency. However, few participants can already affect the network's security on the downside. It is easier for a small group of malicious participants to compromise the consensus outcome and take control of the network.

There are many controversial opinions about calling private permissioned blockchains. Indeed, by definition, a blockchain is decentralised. In contrast, private permissioned blockchains are partially decentralised because the network must be constructed and maintained by a company or consortium of industry participants, which can be viewed as centralisation. This form of centralisation is one of the most significant disfavors of private permissioned blockchains and goes against the core philosophy of blockchain technology. However, although private permissioned blockchains do not have all the characteristics of a blockchain, they do satisfy essential blockchain features. They are append-only ledgers with immutable records ensuring that a transaction cannot be deleted. Every network participant has the complete replication of the ledger, and each transaction is verified and validated through a consensus mechanism. Some examples of blockchains that meet the characteristics of a private permissioned blockchain are *ConsenSys Quorum* [161], *Corda* [47] and *Hyperledger Fabric* [21].

Quorum, Corda and Hyperledger Fabric are managed by a group of participants that allow only trusted and identified participants to participate in the blockchain. These examples of blockchain are flexible in the implementation of consensus. They can be tailored to the trust assumption of a particular deployment or solution. This modular architecture allows the platform to rely on well-established crash fault-tolerant or Byzantine fault-tolerant arrangement tools. Quorum, built on top of Ethereum with privacy extensions, supports protocols such as Raft [153], IBFT [2] and *PoA*, which provide immediate block finality, and a short time to reach consensus. Moreover, Corda applies consensus through *notaries* to prevent double-spending. The notaries attest that a given transaction has not already been executed. In Hyperledger Fabric, the ordering of transactions is

[2]IBFT, for Istanbul Byzantine Fault Tolerant, is an alternative to *PoW* in Quorum. https://consensys.net/docs/goquorum/en/21.10.0/configure-and-manage/configure/consensus-protocols/ibft/

delegated to a modular component for consensus that is logically decoupled from the participants that execute transactions and maintain the ledger. In Hyperledger Fabric, channels can be created which participants on a Fabric network establish a sub-network where every member can access a particular set of transactions. Thus, only participants who participate in a channel have access to specific information, such as smart contracts (called chaincode in Fabric) and transactions.

Remark (Corda architecture)**.** Corda differs from a classic blockchain in several ways. Corda architecture is not built as a sequence of blocks, and the transaction information is not broadcast to all participants but in a point-to-point manner. Information is shared on a need-to-know basis. The architecture of the ledger is such that there is no single version of the network. Two participants do not have the same network view and may have a completely different network construction. Each participant p maintains its ledger, and it is only visible to participants who interact with p. The Corda Ledger is a subjective construct from each peer's point of view, and for some, Corda can not be considered a blockchain but rather a distributed ledger.

Public Permissioned Blockchains

A public permissioned network is open to everyone but with some restrictions. It combines the participant identification from private permissioned blockchain with the liberty to participate in the consensus from public permissionless blockchains. The network is fully decentralised; thus, there is no need for a trusted party or intermediary. Participants who want to join the network are not entirely anonymous and must be identifiable or partially identifiable. Anyone who meets the predefined criteria of the network can download the protocol and have total write equity, i.e. they can create transactions and participate in the validation process. Transactions are issued among partially identified participants without a central authority, and depending on the use case, the network can handle a cryptocurrency. For example, *Ripple* [22], one of the largest cryptocurrencies, supports permission-based roles for participants in a public environment.

For a public permissioned network, participants' permissioning allows the usage of consensus protocols other than *PoW* or *PoS*, taking advantage of the known identities of the participants executing the consensus algorithm. The consensus being open to everyone increases the transaction speed. The transaction validation process is lengthy and can usually take minutes. The consensus is chosen according to the use case; it could be *PoS*, *PoA*, or *BFT*. However, a public permissioned network requires a consensus protocol with strong transaction finality. *EOS* [188], *Sovrin* [112], and *Monet* [23] are examples of such blockchains.

EOS is designed for enterprise use cases, and it can be used in both private and public environments thanks to its customisation capacity. A human-readable name identifies participants. EOS provides a permissioned system and secure application transactions processing. The implemented consensus is *DPoS*, and it achieves high transaction throughputs because *DPoS* does not need to wait for all the participants to complete a transaction to achieve finality. This behaviour results in faster confirmations and lower latency.

Sovrin is an open-source network built on distributed ledger technology for self-sovereign identity. Anyone can join the network but must be identified and follow the Sovrin foundation's specific rules. The blockchain uses the decentralised identifier (DID) to create unique and permanent participant identifiers. Sovrin is the first global public utility exclusively for self-sovereign identity and verifiable declarations.

Monet brings decentralisation and easily scalable blockchains to mobile devices. It is based on the Babble consensus system [25]. Monet is an open-source infrastructure that allows groups of people involved in any task or activity to form temporary networks with their mobile devices and coordinate themselves without trusted intermediaries. Monet is a public network architecture, which means anyone can initiate a temporary mobile network. The blockchain is formed by a small, localised group of individuals taking part in the same activity, making the blockchain permissioned.

Anyone willing to participate in the consensus and stake some voting tokens can become a validator in Monet.

Private Permissionless Blockchains

Like the previous type of blockchain, this blockchain is a combination of private permissioned and public permissionless blockchain. The network is best suited for enterprise usage as it allows the enterprise to comply and meet the needed privacy. Anyone can join a private permissionless blockchain with full transactions, read equity and transparency. Participants are not preapproved and do not have to be identified; they can remain anonymous. However, permission to write is restricted, and not all participants can issue transactions. The participants participating in the consensus are few and are restricted to a selection of participants. Therefore, the transaction validation is short, resulting in a fast transaction speed and reduced transaction fees. The implemented consensus depends on the use case and the network rules. If participants are unknown, it could be *PBFT*, although malicious participants cannot perform writing transactions and only read information. It is difficult to define precisely this type of blockchain as enterprises may set different rules for different applications.

Multichain [93] is an example of a private permissionless blockchain. It is a platform for creating a private blockchain based on a fork of the Bitcoin core. Anyone can connect to the network, but only restricted participants can send or receive transactions. In other words, the blockchain is publicly readable and only applies restrictions on the ability to transact. The consensus process is also restricted to chosen participants, and the participants are kept anonymous.

	Public Permissionless	Private Permissioned	Public Permissioned	Private Permissionless
Participant anonymity	anonymous	identified	identified	anonymous
Consensus	anyone	selected participants	anyone	selected participants
Transaction speed	slow	fast	slow	fast
Efficiency	low	high	medium	high

Table 2.2 – Comparison between blockchain types

2.3 Conclusion

The emergence of distributed systems has significantly advanced computer science. However, their design and usage can be complicated, mainly due to the number of participants involved, the overlap of their actions, and the involvement of Byzantine participants. We showed the importance of the formal definition of a system concerning the representation of a participant's behaviour (via an automaton, for example), the communication model, and the definition of the failure model, which impacts the problem specification. In this chapter, we gave the example of *Two-Phase Commit* and distributed ledgers to illustrate distributed systems. This choice is not insignificant because these two examples have a central role in the work presented in this thesis. Indeed, in the second part of the chapter, we introduced blockchain technology, an example of a distributed ledger. We tried to provide clear and straightforward definitions to give the reader the necessary knowledge to understand the rest of the manuscript. We provided a comparative analysis of the different types of blockchain, which are too often confused. Table 2.2 compares blockchains differentiated according to the anonymity of the participants and the participants who perform the consensus protocol. The table also highlights the speed of transaction validation and the level of efficiency of the blockchain. This chapter does not pretend to be complete regarding distributed systems and blockchains, but it provides the necessary notions for understanding the thesis.

Chapter 3

Formalisation and Formal Proof of Blockchain Systems

" Everything is theoretically impossible, until it is done. "

– Robert A. Heinlein

Contents

The literature on blockchain systems has been growing in recent years. Its success stems from its characteristics as an immutable distributed ledger, authenticated transactions, data transparency, and the absence of a trusted third party. As a result, several scientific papers related to blockchain systems have been published, addressing very diverse aspects. The popularity of this technology has enriched the possibility of use, thus, leading to the emergence of new applications, e.g. using blockchain to manage artists' royalties [177]. This thesis focuses on blockchain systems and their formal verification. The blockchain community has realised that applying formal verification methods to such systems is critical to ensure their proper functioning. A blockchain system is complex and consists of several components. In this thesis, we focus on applying verification methods to smart contracts and on the study and correctness proof of a recent blockchain application, the cross-chain swap. This chapter designs a non-exhaustive state of the art concerning applying formal methods to blockchain systems. First, we discuss the use of formal proof on smart contracts and the employed techniques to ensure the correctness of such programs (Section 3.1). Then, in a second step, we go through various algorithms that provide asset exchange among distributed ledgers and different formal approaches to prove these algorithms (Section 3.2).

3.1 Proof of Smart Contracts

Interest in blockchains has been overgrowing, thanks in particular to smart contracts. These computer programs capable of setting up transaction rules have made blockchain technology successful in the academic [114], industrial [80] and governmental services [145]. If, at the early stage of smart contracts, the application of formal verification was not of much interest, following the infamous "*the DAO*" attack [24] the interest in formal methods at the level of smart contracts has increased. The attack has exploited a combination of vulnerabilities of "*the DAO*" smart contract that resulted in the loss of 3.6 million *ethers* (valued at the time at $50 million). A formal analysis of "*the DAO*" smart contract could have prevented this tragedy because the flaw that led to the attack would have been detected [38, 98, 111, 133].

In the literature, most smart contracts subject to formal verification are *Solidity* contracts [78], as they represent one of the vast majority of contracts existing on blockchain platforms. Several techniques are used, but we focus on model-checking and proof by deduction methods. In the following, we overview works that use the two mentioned methods on smart contracts.

3.1.1 Proof by Model-Checking

Model-checking is a method that checks if a finite-state model of a system satisfies a given specification. If the model does not satisfy one of the specification's properties, a counter-example is generated and provides the trace execution that leads to the violation. This technique is widely used in verifying smart contracts because of their ability to be modelled in state machines. A smart contract starts its life in an initial state and then transits to intermediate states before ending its life in its final state. Moreover, the contract can execute different functionalities in each state, thus changing its behaviour from one state to another. The ability to model a smart contract as a state machine facilitates the choice of applying model-checking for the verification like in [26, 61, 154]. Moreover, one advantage of applying model-checking is checking the termination of a program, represented by liveness properties. In the literature, we notice two approaches to smart contracts representation. The first is modelling smart contracts as a state machine (or, more generally, an automaton) [10, 26, 61, 148, 155], and the second is the translation of a contract written in its implementation language into the verification language [111, 154, 182].

The authors in [26] provide a smart contract formal template that standardises the design of a smart contract and asks the user to fill the parameters to its needs. As said before, this formalisation easily allows the application of formal verification methods such as model-checking. The authors use the SPIN [106] model checker for correctness and security verification. Models are written in PROMELA, which is the SPIN modelling language. PROMELA supports modelling asynchronous

distributed systems and expresses Linear Temporal Logic (LTL) properties which are properties that specify behaviours over time (see Section 4.1). They illustrate their methodology through a shopping example and verify the absence of deadlock and livelock. They also show that the contract can be in exactly one state at a time.

Similarly, the authors in [154] use SPIN and PROMELA to verify smart contracts, but unlike [26], they do not build a smart contract model from scratch but from a translation of existing contracts. The authors propose a framework to translate *Solidity* smart contracts from the specification to the operation level. They provide a set of logical formulas that represent *Solidity*'s functional semantics. The methodology translates the contracts into an equivalent PROMELA model, considering the functional semantics of *Solidity* to ensure the proper translation.

The implementation consists of a parser that parses the *Solidity* code and generates an abstract syntax tree from the parsed code. The generated tree represents the input of the framework that outputs a PROMELA model. The following step is to add assertions into the generated PROMELA model, apply the SPIN model checker, and verify LTL properties. The assertions depend on the verified contract and must be added by the developer in charge of the verification.

The authors in [148] [1] describe an approach to model the operation of an Ethereum application and formulate properties in temporal logic to verify that the model satisfies them. They provide an example of an energy marketplace application and define safety and liveness properties. The article established a methodology to construct a three-fold application model, with properties formalised in temporal logic CTL (see Section 4.1). The three-fold model consists of (*i*) the kernel layer that models the Ethereum blockchain behaviour, (*ii*) the application layer that models functions of the smart contracts, and finally, (*iii*) the environment layer that models the execution framework. The authors use the NuSMV model checker [55] to verify the application's smart contracts.

In the same approach, the authors in [10] provide a minimal blockchain system model to verify smart contracts' behaviour in their execution framework. They simulate the interaction between the smart contract, the user and the blockchain model (execution framework). Moreover, they define several behavioural scenarios from users according to possibly malicious actions from malicious users. A smart contract is modelled as a component, and each function is represented as an automaton. They make use of statistical model-checking to achieve modelling and verification. Statistical model-checking is a technique that combines simulation and statistical methods to analyse stochastic systems. The authors use the framework BIP (Behavior Interaction Priorities) [31] to benefit from its modelling formalism and statistical model-checking engine. They verify properties expressed in PB-LTL (Probabilistic Bounded Linear Time Logic) formalism [10]. To illustrate their methodology, they apply the approach to a name registration contract. The contract consists in associating a blockchain account address to a unique username. Therefore, they analyse the probability that a malicious user succeeds in stealing users' identities by registering their username with its address.

In another approach, the authors in [111] combine abstract interpretation and symbolic model-checking[2] to ensure the correctness of smart contracts using the ZEUS framework [46]. ZEUS is a tool that consists of a policy builder, a source code translator and a verifier. The tool supports Ethereum and Hyperledger Fabric smart contracts. It takes as input the smart contract written in its programming language (e.g. *Solidity*) and generates, with the assistance of the user, an XACML (eXtensible Access Control Markup Language) template [162]. XACML is a proposed XML syntax for describing authorisation and rights policies. ZEUS adds annotations on the smart contract input and then translates it into an LLVM (Low-Level Virtual Machine) bitcode. This low-level representation also helps ZEUS support the verification of smart contracts, written in Java, C# and Go, from different blockchain platforms. ZEUS only accounts for parameters that can be computed at the source code level and hence cannot verify properties such as *gas* consumption.

[1] This article refers to a working project from before the thesis.

[2] Symbolic model-checker [138] allow the verification of extremely large state-spaces.

Other tools like UPPAAL [34], UMC4M [180] and Cubicle [60] are used to verify smart contracts. The authors in [155] provide a way to verify smart contracts by applying UPPAAL on an auction smart contract and check if the model satisfies the required properties. The smart contract is modelled as timed automata, and properties are expressed in TCTL formula (Timed Computational Tree Logic) [16]. The authors in [182] provide a method for modelling, simulating and verifying smart contracts using a verification language, MSVL [181], which is a temporal logic programming language. They developed a tool, SOL2M, that translates *Solidity* contracts into an MSVL model. Then, they define PPTL (Propositional Projection Temporal Logic) formula to express the security properties of smart contracts. Finally, the model checker UMC4M [180] verifies whether the MSVL model satisfies the PPTL security properties. They apply their methodology to a bank transfer smart contract to verify properties as functional and logical correctness. Nevertheless, the SOL2M does not translate all of *Solidity*'s languages.

Finally, the authors in [61] use a parametrised model checker, Cubicle [60], based on SMT and define the properties in first-order logic (FOL) formulas (see Section 4.1). They provide a two-layers framework for smart contract verification [61]. The first layer consists of the model of the blockchain transactional mechanism, while the second layer is a model of the smart contract. They defined a methodology of proving smart contracts' safety using ghost variables; thus, to avoid code changes while proving it. Thereby, to be verified, functional properties such as safety are expressed in their negated form, characterising unsafe states. If Cubicle finds a way to reach an unsafe state, an error trace is printed (i.e. a counter-example). To illustrate their approach, they use the example of an auction smart contract.

Table 3.1 compares the different approaches presented above. The cited articles differ according to the used model checker and the expression of the properties to be proved. The properties are generally expressed in temporal logic except in ZEUS [111] and Cubicle [61], where properties are expressed in first-order logic (FOL). ZEUS supports quantifier-free FOL for defining safety properties expressed as assertions in the code. The approach of Cubicle in [61] is the only one to use a parametric tool, thus avoiding the generation of combinatorial explosions. However, since both rely on FOL to express their properties, the tools do not support liveness verification that needs temporal logic to express those properties.

The approach in [148] gives interesting results from the modelling method; however, the expression of property verification is limited due to combinatorial explosion and invariant generation (most frequently implicit). Thus, proving properties involving many states was impossible to achieve. Hence, ambitious verification could not be achieved because of the limitation of the model checker, e.g. a model for m consumers and n producers.

The authors of the cited articles illustrate their approach through examples, which often have to handle money (tokens). Two verification methods can be noticed, either translating an existing contract into the language of the verification tool or modelling the contract, abstracting the implementation language. For example, in [26], the authors did not focus on specific smart contract languages since the methodology is independent of any language. Their technique allows them to be very generic on all contracts, but they do not discuss the problems related to the language. A model may seem correct, but the implementation can contain bugs. In addition, although the authors in [10] mention the *Solidity* language, it does not seem to depend on the implementation language of smart contracts. The modelling is generic enough to be applied to other languages. However, the simulation and models are dependent on the verified contract.

Finally, the authors in [154] focus their study solely on *Solidity*. Nevertheless, the translation function in [154] does not handle every *Solidity* feature; events, inheritance, structs, local variables and strings are not covered in the article. Moreover, the framework is limited to the correctness of individual contracts. Indeed, it does not handle the verification of a network of smart contracts that interact with each other.

Reference	Tools	Logic properties	Type of properties	Method of verification
Bai et al. [26]	SPIN /PROMELA	LTL	• Safety – no deadlock • Liveness – no livelock	Modelling smart contracts into state machines.
Osterland et al. [154]	SPIN /PROMELA	LTL	• Safety – "the initial balance is correctly set"	Translates *Solidity* contract into PROMELA model.
Nehaï et al. [148]	NuSMV	CTL	• Safety – "Alice cannot sell more energy that the amount she has supplied to the grid" • Liveness – "Once opened, the market will eventually be closed"	Modelling *Solidity* smart contracts into NuSMV modules.
Abdellatif et al. [10]	BIP	PB-LTL	• Safety – "A hacker cannot register a user name"	Modelling smart contracts into timed automata.
Kalra et al. [111]	ZEUS	FOL	• Safety – policy confirmation	Translates *Solidity* contract into LLVM bitcode.
Park et al. [155]	UPPAAL	TCTL	• Safety – "bidding cannot be made by two people at the same time" • Liveness – "when the auction begins, the auction must end at the specified end time"	Modelling smart contracts into timed automata.
Wang et al. [182]	UMC4M	PPTL	• Safety – logical correctness • Liveness – functional correctness	Translates *Solidity* contract into MSVL language.
Conchon et al. [61]	Cubicle	FOL	• Safety – "no loss of money"	Modelling smart contracts into automaton.

Table 3.1 – Various approachs of smart contracts' verification using model-checking

3.1.2 Proof by Deduction

The deductive approach has the advantage of being parametric and does not require the exhaustive generation of all possible states of a system. It allows complex systems to be verified without encountering the main problem of model-checking – the combinatorial explosion. Aware of this advantage, several research studies apply the deductive verification method to smart contracts. The methodology can consist of annotating the source code to apply verification tools for proving the contract's correctness [32, 98] or designing an abstract contract model to be verified [18, 103]. However, much deductive verification work involves translating smart contracts into the formal verification language [12, 36, 38, 65, 98]. The deductive approach often implies a language-dependent methodology. In this section, most of the contracts studied are written in *Solidity* language, which is the most widespread.

The authors in [38] outline a framework to analyse and verify the runtime safety and the functional correctness of Ethereum contracts translated to F*. F* is a function dependently typed programming language aimed at program verification. The language uses an SMT solver to prove functional properties. The F* framework consists of two advanced tools: Solidity* to verify *Solidity* smart contracts and EVM* to verify the Ethereum Virtual Machine (EVM). EVM* is a decompiler of EVM bytecode into an F* code.

The process can be decomposed into three steps. *(i)* Solidity* translates *Solidity* contracts to F*, and this level of verification allows verifying functional specification and safety properties. F* is a modular language, and each translated contract represents a module in F*. *(ii) Solidity* contracts are compiled into the EVM, and the EVM generates a bytecode. *(iii)* The generated bytecode is given

as input to EVM*, which generates an equivalent F* code as output. The bytecode level enables analysing of low-level properties such as the amount of *gas* consumed to complete a transaction. The F* framework verifies the equivalence between the F* code generated by EVM* and Solidity*.

The approach defined in [98] describes a method for verifying *Solidity* contracts. The authors employ the tool SOLC – Verify [98], which uses annotation and translation methods. The verification is at the source code level, on top of the *Solidity* compiler – i.e. EVM. Although the tool reason on high-level contract properties, it models low-level language semantics precisely.

The methodology uses the source code annotation done by the developer to perform the verification. It is built such that *Solidity* contracts, including specification annotations, are translated into the Boogie intermediate verification language [129]. The extended compiler creates a Boogie program from the *Solidity* contract, and Boogie transforms the program into verification conditions (VCs) and discharges them using SMT solvers. Boogie uses Z3 [67] and CVC4 [30] but can also support YICES2 [74]. The results are mapped back and presented at the *Solidity* code level.

SOLC – Verify targets functional correctness of contracts such as invariants, loop invariants, and pre- and postconditions. The tool can infer implicit specifications in unannotated contracts. Examples of such implicit specifications are overflow checking, e.g. array lengths or loop counters, requiring statements and assertion checking. Conversely, some properties such as flaw detection need to be annotated in the contract.

The authors in [103] use the Lem language [142] to formally define the EVM and apply interactive theorem provers for smart contract verification. Examples of some popular theorem provers are Coq [62], Isabelle/HOL [151] and HOL4 [170]. The corresponding article applies Isabelle/HOL[3] for proving safety properties and invariants of a smart contract presenting a reentrancy flaw (see Section 5.1.1 for reentrancy definition). The authors specify the interface between smart contract execution and the rest of the world. Moreover, they define a function to calculate the exact *gas* consumption during the execution of an instruction. The authors argue that their formal definition of the EVM can serve as a basis for further analysis and development of Ethereum smart contracts.

It is indeed based on that paper that the authors in [18] have developed a methodology of smart contract verification. The authors argue that the method is independent of any high-level language compiler. They use the Isabelle/HOL theorem prover to analyse smart contract correctness at the bytecode level. The reason for targeting the bytecode is because it is the only language understood by the EVM. All compiled smart contracts are translated into bytecode. Thereby, the authors base their work on an EVM formalism [103] and extend the Isabelle/HOL framework to verify smart contracts. Moreover, they provide logical rules of proof to automatically generate verification conditions (VCG [4]). However, the framework does not support reasoning about inter-contract message calls.

In [32], the authors show that the architecture of smart contracts provides a suitable computational model for applying deductive verification methods. They use the KeY tool [11] to verify smart contracts of the Hyperledger Fabric blockchain [21]. Fabric makes it possible to write smart contracts in different languages like Go or Java. Since KeY allows proving programs written in Java, the authors focus on smart contracts written in that language. Their article presents three different classes of smart contract correctness properties: generic properties, specific properties and properties of the distributed ledger application. The generic properties are independent of the smart contract, and the specific correctness properties relate to the behaviour of the smart contract program (commonly expressed in functional properties). Correctness of distributed ledger application implies invariants and liveness properties defined in temporal logic. Although the authors mention

[3] Isabelle is an interactive logical framework for theorem proving.

[4] A VCG synthesises a set of formal verification conditions by analysing an annotated program, which is verified using a theorem prover.

the proof of liveness at the level of the distributed ledger application, it seems that in their article, no example defines such property.

The KeY tool benefits from an advantage in that it supports transaction verification to deal with the rollback of interrupted transactions. This feature is the mechanism used by EVM to deal with transaction failures. Consequently, the authors in [12] also use the KeY tool and present an approach to verifying smart contracts written in *Solidity*. In addition, benefiting from the advantage of the KeY tool, they provide support for rollbacks in case of exceptions. This work is a translation-based verification of *Solidity* contracts. The authors design a tool, JAVADITY, that takes as input the *Solidity* contract and generates a Java program to be completed with the specification in JML (Java Modeling Language [127]) that can express ghost fields. The translation is automatic, and the resulting Java program can then be checked with the KeY tool. They verify business logic and contract-specific specifications.

More recently, other smart contract writing languages have emerged, such as the *Michelson* language [173]; the smart contract language of the *Tezos* blockchain [14]. WhylSon [65] is a tool for verifying smart contracts written in that language. The formal language behind the WhylSon tool is *WhyML*, a programming language of the *Why3* framework (see Section 4.2 for more detail about *Why3*). The input of WhylSon is a *Michelson* contract that goes through a parser producing an abstract syntax tree. The tree is sent to the *Why3* API that generates a *WhyML* contract to apply verification to it. The translation within WhylSon is semi-automatic, thanks to the *Why3* framework. One significant advantage of this approach is the VCG (verification condition generation) provided by the *Why3* framework and the backend support for several automated theorem provers. Moreover, WhylSon can infer some categories of safety conditions as the length of the array and type variables. However, the authors did not formalise the internal details of the cryptographic operations; instead, they defined these instructions as abstract operations that follow the expected specification.

Similarly, Mi – Cho – Coq [36] is a framework for verifying smart contracts written in *Michelson*. The framework implements a *Michelson* interpreter in Coq [62] and applies the *weakest precondition calculus* (see Section 4.1.2). The interpreter translates *Michelson* contracts into Mi – Cho – Coq abstract syntax tree. Mi – Cho – Coq makes use of Coq for proving functional properties. Still, the tool is currently not expressive enough to state properties about the lifetime of a smart contract nor the interaction between smart contracts.

The deductive approach is of great interest because it can be parametric and express functional properties. Moreover, smart contracts programs are perfectly adapted to the application of deductive verification tools because of their architecture and sequential structure. Table 3.2 summarises the methods presented in this section by mentioning the tools used by each and the target of the smart contracts being verified. The targeted smart contract language is often *Solidity* [12, 38, 98]. However, applying theorem proving techniques to *Solidity* contracts is challenging because its semantics have no formal definition. Therefore, translating *Solidity* contracts into a formal language can be tricky. For example, the authors in [12] face the difficulty of translating nested *Solidity* expressions into Java. Therefore, although they advance an automatic proof approach, verifying some expressions requires user assistance. Moreover, the authors in [65] admit having encountered difficulties in automating proof of *Michelson* contracts. For example, the numerous encoding of the *Michelson* language in *Why3* made the proof of safety properties difficult to achieve by the SMT. Similarly, in [36] where the approach does not provide automation.

The authors in [18] and [103] rely on EVM bytecode contracts to overcome these language issues and are separated from all high-level languages. However, like in [12], the level of automation in the [18] approach is limited, and the user needs to interact with the proof system to discharge elaborated claims.

Table 3.2 also defines the verification method and how properties are expressed by giving examples. The use of a deductive approach increases the possibility of property expressions. Many of the cited work rely on Hoare's logic (see Section 4.1) to prove their program. For example, an important property to verify is the *gas* consumption, as was done in [18, 38, 103]. However, the authors in [12, 32, 36, 65, 98] do not formalise *gas* semantics nor provide proof of *gas* consumption.

Reference	Tools	Smart contracts	Type of properties	Method of verification
Bhargavan et al. [38]	F*	*Solidity* and EVM bytecode	• Source level (Solidity*): functional correctness (contract's invariants), runtime errors. • Bytecode level (EVM*): *gas* consumption.	Translates *Solidity* contract into F* contract.
Hajdu et al. [98]	SOLC – Verify	*Solidity*	• Pre- and postconditions, loop invariants, invariants, assertions and functional correctness. • Examples of properties: reentrancy detection, absence of overflow.	Annotates *Solidity* contract and translates it into Boogie program.
Yoichi Hirai [103]	Lem	EVM bytecode	• Safety invariants. • Example of property: reentrancy detection.	Modelling EVM and smart contracts in a formal definition.
Amani et al. [18]	Lem	EVM bytecode	• Safety and security. • Functional correctness, pre- and postconditions. • Examples of properties: *gas* consumption.	Modelling EVM and smart contracts in a formal definition.
Beckert et al. [32]	KeY	Hyperledger Fabric	• Generic properties: termination. • Specific properties: functional correctness. • Correctness of distributed ledger: invariants.	Annotates the original contracts.
Ahrendt et al. [12]	KeY (JAVADITY)	*Solidity*	• Pre- and postconditions, invariants and functional correctness. • Examples of properties: absence of under and overflow.	Translates *Solidity* contract into Java program.
da Horta et al. [65]	WhylSon and *Why3*	*Michelson*	• Functional properties, invariants, pre- and postcondition. • Examples of properties: length of variables, type correctness.	Translates *Michelson* contract into *WhyML* program.
Bernardo et al. [36]	Mi – Cho – Coq	*Michelson*	• Weakest precondition and functional correctness.	Translates Michelson contract into Mi – Cho – Coq program.

Table 3.2 – Various approachs of smart contracts' verification using deductive verification

3.1.3 Alternative Formal Verification Methods

The method of symbolic execution [45] is also an approach to verify existing smart contracts. The approach consists of providing the code to the symbolic execution tool, which will analyse all the possible paths to generate test or verify assertions. OYENTE is a symbolic execution tool to find security bugs. It has been developed to analyse Ethereum smart contracts to detect flaws as a pre-deployment mitigation. The analysis focuses on the EVM bytecode; thus, no high-level language is

targeted (e.g. *Solidity*). In the corresponding paper [133], the authors were able to run OYENTE on the bytecode of 19,366 existing Ethereum contracts, and as a result, the tool flagged 8,833 of them as vulnerable. The tool was able to detect the "*the DAO*" bug [24]. OYENTE is based on a formalisation of the Ethereum semantics, for example, the execution of EVM instructions and recommended solutions based on this formalism. The methodology is that the bytecode is given as input of the tool along with the Ethereum global state to initialise the contracts variables. The goal is to browse a control flow graph that corresponds to the symbolic execution of the bytecode smart contract being analysed. Then they use solvers, like Z3, to decide on the feasibility of the branching conditions. The tool can detect four well-known bugs in smart contracts. This work represents one of the first approaches to formal verification of smart contracts, although the tool is not a formal verification tool per se. OYENTE is more defined as a debugger than a prover. Therefore, although that work provides interesting conclusions, it uses symbolic execution on Ethereum bytecode, analysing execution paths, so it does not allow to prove functional properties.

The authors in [137] provide a framework to model smart contracts as state machines in rigorous semantics. The framework called FSolidM automatically generates Ethereum smart contracts from the designed state machine. Their approach is to create correct contracts before their deployment on the blockchain. The framework automatically generates *Solidity* contracts from a graphical representation of the smart contract (the FSM state machine). Each transition of the FSM is translated as a *Solidity* function. They provide solutions for known vulnerabilities such as reentrancy and unpredictable state by using a set of plugins that users can add to their contracts. Although the paper does not propose a formal smart contract language, it allows for building a contract from a formal representation (i.e. state machines). The tool is built to generate *Solidity* code, although the authors claim it can generate other types of contracts.

3.1.4 Our Contribution to Smart Contract's Proof of Correctness

This thesis studies how a language dedicated to deductive verification can be a suitable language for writing correct and proven contracts. The proposed language is *WhyML* from the framework *Why3*, introduced in Chapter 4. In Chapter 5, we introduce the methodology of writing smart contracts correct by design using writing and proof rules. Then, we formulate properties as the absence of runtime errors and functional properties, including the verification of *gas* consumption. These two contributions are described in Section 5.1. Finally, Section 5.3 defines the contribution where we describe the approach of compiling *WhyML* contracts into EVM code to prove the cost of *gas*.

These contributions to the verification of smart contracts have been presented and published in the proceedings of peer-reviewed conferences [146].

3.2 Cross-Chain Swap Algorithms

As blockchain has become more widespread, some limitations have emerged, such as communication between different blockchains. On this note, one of the most popular applications in recent years involving blockchain and smart contracts is *cross-chain swap* algorithms. At a high level, *cross-chain swap* algorithms aim to have a set of participants settling transfers on different blockchains. For example, Alice, Bob and Charlie are respectively in blockchains A, B and C, but they want to exchange some assets without the need for intermediaries. Suppose that Alice wants to transfer coin A to Bob, which wants to transfer coin B to Charlie, who wants to transfer coin C to Alice. The users being on different blockchains, a distributed protocol is needed for realising the swap among participants. What is often expected from *cross-chain swap* is that at the end of the algorithm, all transfers must take place or none at all, often translated into the *atomicity* property. However, in the current literature, specifications do not agree on what a swap protocol should guarantee regarding the safety and liveness properties. For instance, the author in [104] analyses *cross-chain swap* protocols and the feasibility of the atomicity property. The author argues that it is impossible to

have atomicity as it is defined without proper assumptions in an asynchronous system. They use a Kripke model [5] of intuitionistic propositional logic to prove it. In addition, underlying timing and failure assumptions vary from one protocol to another [95, 101, 176, 191].

This section provides an overview of existing swap algorithms divided into two families. Algorithms based on *Hashed Timelock Contracts* (HTLC), defined in Section 3.2.1, and those using *Verifiable Proofs* to accomplish the swap, are defined in Section 3.2.2. An HTLC [186] is a smart contract used in blockchain applications. It reduces participant risk by creating a time-based escrow that requires a cryptographic secret for unlocking. In the literature, swaps protocol are often modelled as a directed graph (DAG) [19, 95, 101, 169, 191, 195].

3.2.1 *Cross-Chain Swap* based on HTLC

The one that remains the reference is the *atomic cross-chain swap* of Herlihy [101]. This paper introduces a distributed transactions algorithm, the atomic cross-chain swap, based on HTLC. The algorithm is modelled as a directed graph to transfer assets (cryptocurrency, cars, houses) across different blockchains. The author defines two types of users: *rational* users that act in their self-interest and can deviate from the protocol if it is profitable for them, and *irrational* users (also called Byzantine) that may deviate from the protocol whether this is beneficial to them or not. If all participants behave rationally, the swap takes place atomically. However, if a participant acts irrationally and deviates from the protocol, only the irrational participant will worsen. The protocol presented in [101] is designed such as rational participants have no incentive to deviate from the protocol. The HTLCs are used to escrow (or lock) the transferred assets in the algorithm. For example, consider Alice, Bob and Charlie from previously. Alice first generates a random secret s and produces a hashlock h, where $h = H(s)$ and H is a cryptographic hash function. Then, Alice publishes a contract to lock her coin A. She adds to the contract the hashlock h and sets a timelock t to ensure a refund of her coin if something goes wrong. Bob and Charlie publish their contract using the same h but different t. Then, Alice reveals her secret s to Charlie's contract to unlock the asset, which reveals s to Bob's contract and receives coin B, which in turn reveals s to Alice's contract and receives the coin A.

The algorithm is designed so that its time execution depends on the size of the swap because each execution (contract publication) depends on the previous execution. This process results in increased latency. The authors in [108] propose a protocol to improve the space and local time complexity of [101]'s protocol by using only signatures to set hashed timelocks instead of the graph topology. Unlike Herlihy's paper, the authors in [108] provide a formal description of the participants' behaviour through detailed algorithms.

Herlihy's protocol strongly inspires the paper [169]. The paper presents a uniform protocol for generic cross-chain transactions modelled as a directed graph that uses HTLCs. In [101, 102], Herlihy showed no uniform protocol for cross-chain transactions exists unless the transactions are strongly connected.

However, in [169], the authors present a synchronous three-phase protocol (3PP) to execute cross-chain transactions, including graphs with sequence steps and graphs with off-chain steps that may not require strongly connected transactions. A sequenced step is an asset on an outgoing edge of the graph representing the sender's asset that it does not own yet (the case of *cross-chain deals* in [102]). The authors came up with a tool, XCHAIN, that takes the graph of transactions and the participants' addresses as input and removes sequences of steps from the input transactions by transforming the graph. Moreover, the tool automatically generates a *Solidity* smart contract for each asset transfer from a high-level description of a cross-chain transaction.

The authors in [68] address the latency problem for transactions within the cross-chain carried out in the blockchain (on-chain). They propose a new payment protocol that reduces the number

[5] A kripke model is an alternative way of representing finite-state machines.

of transactions committed to the blockchain and the transaction confirmation delay. The protocol is built upon off-chain micropayment channels to construct what they call a *duplex micropayment channel*. Off-chain solutions are intended to perform transactions off the chain, increasing the bandwidth simultaneously. As a result, the blockchain is only involved during the setup and closure step of the channel. The duplex micropayment protocol guarantees end-to-end security and allows immediate transfers, unlike Bitcoin transfers, which take minutes to be confirmed. The duplex micropayment uses HTLCs to ensure an end-to-end secure protocol and track money transfers without trust among the participants. A micropayment channel is established between two participants, who make payments to one another, none of which are recorded on the blockchain. The setup consists of two transactions, a 2-of-2 multi-signature transaction and a time-locked refund.

Still based on [101]'s protocol, the authors in [40] propose the concept of *atomic loans* that can be implemented as an extension of atomic swaps. In this work, it is assumed that two participants on different chains communicate through a communication protocol. The protocol allows participants to create loans and enables the trustless transfer of value between various cryptocurrency systems. Atomic loans extend the concept of sharing secrets of HTLC from the atomic swap protocol to enable debt and repayment between participants at different intervals in a loan process. Therefore, as in atomic swap, atomic loans are based on HTLCs. This requirement implies the need for blockchains that implement smart contracts. As a result, blockchains such as Bitcoin are incompatible with implementing atomic loans. When the users agree on loan terms, the loan is issued, and the terms are incorporated into a smart contract.

Through these works, we note that developing a cross-chain algorithm based on HTLC requires the involvement of a blockchain capable of supporting smart contracts. The authors in [195] propose an extension of the common *cross-chain swap* based on HTLC for blockchains that cannot write such contracts. Their protocol is designed for blockchain with smart contracts that only support multi-signature transactions. This extension provides more outstanding capabilities for cross-chain communications without adding any trust assumption among the participants. Those transactions are controlled by multiple private keys and require, to be valid, a certain number of signatures. A different private key generates each signature, and all those signatures need to be explicitly attached to the transaction.

Several projects implement HTLCs differently, providing different correctness guarantees. However, the general algorithm is quite similar in most of the solutions. The protocols based on HTLC require synchronous communication, and the participants must be connected during the entire swap process. This requirement can be a disadvantage because if a swap participant unintentionally disconnects (e.g. due to a failed network connection), the participant may lose assets.

3.2.2 *Cross-Chain Swap* based on Verifiable Proofs

In synchronous solutions [95, 101], based on timed actions, a swap can result in a correct but slow participant being worse off at the end of the swap. Various cross-chain solutions have been developed to counter this constraint. One of the solutions is based on the proof of content to perform the swap [88, 95, 102, 191, 193]; others use *relays* [77] like in [126, 193].

Zakhary *et al.* [191] are the first to propose a protocol in which correct asynchronous participants are never worse off at the end of the swap. The authors have coped with this problem by drawing on a well-known protocol in distributed transactions, namely the *Two-Phase Commit* [37]. By getting close to this algorithm, participants in [191] lock their assets at the beginning of the protocol. Afterwards, a coordinator (a *witness* smart contract) either authorises or aborts all the transfers. This swap is modelled as a directed graph and consists of sub-transactions. Each sub-transaction transfers an asset on some blockchain. They present a solution to the problem of implementing such a swap while aiming to ensure *Atomicity* and *Commitment* properties. *Atomicity* ensures that

either all transactions occur or none of them, and *Commitment* guarantees that once the protocol decides the commitment of the swap, all asset transfers must eventually take place. They guard against behaviour deviating from the protocol by checking the blockchains' content during the swap.

However, the specification does not cover all deviating behaviours. For example, if we consider a swap between A and B. A transfers *bitcoins* to B, which in turn transfers *ethers* to A. If the swap is authorised to commit, A safely retrieves the transferred *ethers*. However, imagine that B crashes just before being able to retrieve the transferred *bitcoins*. The sub-transaction that characterises the *bitcoins* transfer will never occur. Thereby, we face the violation of the *Atomicity* and *Commitment* properties.

Such as Herlihy's protocol, the execution logic of the swap is handled by smart contracts (immutable and permanent in the blockchain), and assets are put in escrow thanks to smart contracts. Smart contracts have a state that can change depending on the protocol. These changes represent proofs of content used in the protocol to unlock assets. The protocol has two mutually exclusive events (that never co-occur) to satisfy the atomicity: *redeem* and *refund*. Each event value depends on the coordinator state characterised by the *witness* smart contract. The protocol proposes a way to ensure that the proof of content is in a block, deep enough to have a negligible probability of having a fork (see Section 2.2.5). Therefore, The protocol is atomic with a probability of $1-\epsilon$, with ϵ the probability of having a fork.

In [102], the authors provide a new definition problem of atomic transactions across distributed ledgers, namely *cross-chain deals*. The cross-chain deals is modelled as a matrix M where $M_{i,j}$ characterises a transfer of some asset from participant i to participant j. They illustrate the implementation of the cross-chain deals into two different protocols: a protocol that assumes synchronous communication based on HTLC (similar to [101]) and another assuming partially synchronous communication based on certified blockchain and verifiable proofs. The atomic cross-chain swap inspires cross-chain deals. However, in [102], the authors detail why a swap is considered a special case of deals and that deals are more powerful and flexible. For example, the atomic cross-chain swap does not handle indirect transfers, as mediated by a broker, but the cross-chain deals do. A deal is divided into five phases:

1. The clearing phase: the deal's setup (creating the matrix).
2. The escrow phase: the locking of assets willing to be transferred.
3. The transfer phase: the potential asset transfer.
4. The validation phase: the participants check whether the created deal (at the transfer phase) corresponds to the created matrix (at the clearing phase).
5. The commit phase: the participants vote for the commit or the abort of the potential transfers.

In this part, we focus on the so-called CBC protocol (certified blockchain) inspired by the *Two-Phase Commit* protocol to perform the deal. As in [191], the authors in [102] use a particular blockchain to endorse the role of the coordinator. Each deal participant votes on the CBC to abort or commit the deal. The result of the vote is stored in the CBC, and any participant can extract this information that will correspond to proof of action. If the vote is a commit vote, the proof extracted from CBC will serve for claiming the assets, while an abort vote will be for refunding the assets. Smart contracts are deployed to verify the extracted proofs to unlock the escrowed assets for the recipients.

Some protocols are developed to perform cryptocurrency exchanges between specific blockchains, as in [95]. The protocol describes achieving atomic swaps between two untrusted Bitcoin and Monero blockchains participants. Untrusting any central authority ensures that their funds are safe if both participants follow the protocol. The protocol does not require timelocks on the Monero blockchain or script capabilities but requires two cryptographical proofs and synchronous communication. In addition, the protocol is based on the private key generation and addresses, making this scheme blockchain agnostic. The authors argue that the protocol can be adapted to any other cryptocurrencies that are Monero-like blockchain and Bitcoin-like blockchain.

Briefly, the protocol works as follows: the participant on Monero generates a private spend key that is split into two secrets. Then, it moves the funds into a specific address where each participant controls half of the private spend key. Depending on who reveals their half of the private spend key, the locked Monero changes ownership. Therefore, to achieve an outcome, one participant must gain knowledge of the entire private spend key at the end of the protocol execution, either for a completed swap or for an aborted swap. If the swap takes place, the participant on Monero owns the *bitcoins* by revealing its private key share, thus allowing the user on Bitcoin to own the locked *moneros*.

Some approaches use relays to achieve a cross-chain swap [126, 193]. A relay is an untrusted component that relays block headers between two blockchains. The authors in [126] propose Horizon, a *gas*-efficient cross-chain protocol to transfer assets from a *Byzantine Fault Tolerant (BFT)* [125] blockchain to another blockchain. The protocol requires the first blockchain of the protocol to be *BFT* because the blockchain ensures *finality blocks* (see Section 2.2.4). The second blockchain may be any other blockchain but must provide the possibility of writing smart contracts, for example, Ethereum. The authors construct a super-light client that relies on cryptographic proof of content, allowing a client to prove that a transaction has been recorded on a *BFT* chain. The system consists of (*i*) a client who wants to perform cross-chain transactions to transfer some x tokens from a blockchain A to a blockchain B, (*ii*) a relay that periodically submits information about A's chain to B, and (*iii*) a full node [6] that maintains an up-to-date copy of A's chain. Moreover, a smart contract is deployed on blockchain B to verify the proof that guarantees that the transaction is recorded correctly on the *BFT* chain (the blockchain A).

The transaction consists of two on-chain transactions, T_{burn} and T_{unlock}, recorded on A (the *BFT chain*) and B, respectively. T_{burn} translates the transfer of x coins in blockchain A to an empty address, i.e. deleting the coins. T_{unlock} represents the proof to unlock the asset from blockchain B.

Horizon protocol is divided into two parts. The first part is the synchronisation between the relay and the contract. The relay sends every 24 hours the most recent block of blockchain A which contains sufficient and necessary information for the contract on blockchain B to verify the inclusion of a burn transaction submitted by the client. The second step is the cross-chain transaction initiated by the client. Once the burn transaction, T_{burn}, is validated in A (i.e. added to the chain by the *validators* of A), the client sends a request to the full node. Once the request is received, the full node finds the block that includes T_{burn}. Then, it generates the proof of burn Π_{burn} that depends on the transaction T_{burn}. The full node sends Π_{burn} to the client; then, the client creates a transaction T_{unlock}. The latter transaction is submitted to the contract of blockchain B. The contract verifies the validity of Π_{burn}, and if the proof is valid, the contract unlocks the token x on B. The verification of verifying the inclusion of T_{burn} submitted by the client and the consistency of the information extracted from Π_{burn}, e.g. the amount in Π_{burn} equals x.

XCLAIM [193] proposes protocols for issuing, transferring, swapping and receiving cryptocurrency-backed assets securely in a non-interactive manner on existing blockchains. XCLAIM constructs a publicly verifiable audit log of participants' actions on blockchains and employs collateralisation and punishments to enforce the correct behaviour of participants. Thereby, XCLAIM follows a proof-of-punishment method, i.e. participants must proactively prove commitment to system

[6] A full node is a component that holds a copy of the entire blockchain.

rules. XCLAIM is a framework for performing cross-chain exchanges in a trustless environment. A smart contract on each chain controls the exchanges between the two chains (e.g. Bitcoin and Ethereum) and penalises malicious parties by taking their collateral in favour of honest parties. The structure of XCLAIM consists of three main components: A participant who wishes to perform the assets' exchange, e.g. from Bitcoin to Ethereum, a vault for locking the Bitcoin assets received from the participant, and an Ethereum relay contract called BTCRelay [77], which stores Bitcoin block headers to allow verification of SPV (Simple Payment Verification) proofs [79]. The protocol starts with locking sufficient collateral on the Ethereum smart contract in the vault. The participant then sends its *bitcoins* to the vault and submits proof to the contract showing that the transaction has been recorded on the Bitcoin blockchain. Consequently, the chain relay verifies this proof and confirms that the lock has been executed rightly to the contract. This last step allows releasing Ethereum assets to the participant. Although the example presented in the paper is between Bitcoin and Ethereum, the authors claim that all blockchains are compatible with the protocol.

Zendoo [88] is a system that performs decentralised cross-chain payment between Bitcoin-like blockchains. The protocol defines a mainchain (a parent blockchain) and a set of sidechains (child blockchains). A sidechain [89] is a mechanism for two existing blockchains to interoperate where one blockchain (mainchain) considers another blockchain as an extension of itself (the sidechain). Nodes from the sidechain can observe the mainchain's state, but the mainchain can only watch the sidechains via cryptographically authenticated certificates. They use zero-knowledge cryptography called Zk-SNARKSs [194] that enables the authentication, validation, and integrity of the information provided by the sidechains via verifiable proofs. Such proofs are used to generate certificate proofs for the mainchain, enabling a secure verification scheme.

However, the authors in [43] showed that, in practice, it is not possible to verify the existence of specific data on one blockchain from within another blockchain. To verify the presence of particular data on one blockchain, one must pull the blockchain up by its roots, i.e. one must verify the entire chain up to the genesis block, to achieve definite certainty over the presence of the data in the blockchain. The authors in [43] formalise the cross-blockchain proof problem and describe the concept of a cross-blockchain asset transfer protocol using claim-first transactions. This protocol allows for the decentralised transfer of assets between blockchains despite the cross-blockchain proof problem by avoiding the necessity of proving that an asset is spent when claiming it. Conversely, the protocol relies on eventual spending on the source blockchain by rewarding parties.

3.2.3 Alternative *Cross-Chain Swap* Solutions

So far, we have seen algorithms that rely on HTLC and verifiable proofs. However, other solutions do not use these solutions and perform cross-chain transactions between different blockchains. The authors in [176] introduce a new specification formalism called *Asynchronous Networks of Timed Automata* (ANTA) to formalise cross-chain payments. Their article ensures the protocol termination within a known time-bound and a protocol that works correctly in the presence of clock skew between participants. Moreover, the ANTA formalism provides an algorithm that solves the cross-chain payment problem only by assuming partial synchrony and in the presence of Byzantine failures. ANTA simplifies the representation of cross-chain payment to a participant (a customer) automaton and an escrow automaton that describes states from which outgoing transitions are immediately enabled and conditional upon some predicates. An escrow is a specific process that can handle values, and customers (named Alice and Bob), are the participants wishing to make the transfer of a particular value (a payment). Moreover, intermediary customers, called *connectors*, are involved in transferring the value from Alice to Bob as being the intermediaries between them. Each escrow is connected to two customers (whether a customer or a connector), and each connector (intermediary customers) is connected to two escrows. The customers and connectors must trust their escrow.

To achieve a transfer, participants must send to customers or escrow connected to it a promise to send or receive some data. This data can be either some value or a certificate. The receiver of the

payment (i.e. Bob) generates a signed certificate to attest the receipt of the payment and sends it to the escrow to which it is connected.

CAPER [19] is a permissioned blockchain system designed to support a set of non-trusting collaborating distributed applications. Each application runs on a disjoint subset of asynchronous nodes. The CAPER system is Byzantine fault-tolerant as nodes might be faulty. The purpose of CAPER is to allow both internal and cross-application transactions. Internal transactions are visible only to the application generating the transaction, and cross-application transactions are public and accessible to all applications in the system. The CAPER's ledger is formed as a directed acyclic graph (DAG) where nodes of the graph are transactions and edges enforce the order of transactions. Applications do not maintain the blockchain ledger, but each application maintains its local view of the ledger, including its internal and cross-application transactions. Unlike the Bitcoin or Ethereum blockchain, the CAPER ledger is the union of all the applications views, and there is no single version of the mainchain. A CAPER system can be defined as a blockchain of blockchains. As CAPER is a permissioned blockchain system, it makes it a perfect solution for trading within an organisation that wishes to keep certain information private.

The authors in [99] propose a novel cross-chain mechanism to provide interconnection using *tunnels* between different blockchains using plug-ins as nodes added to applications. They introduce the concept of membranes to describe the cross-chain mechanism easily, where a membrane is a plane of the blockchain. The cross-chain principle of the paper is to project the blockchain networks involved in the cross-chain protocol onto a plane. Thus, several blockchains are several planes, i.e. several membranes. Asset transfers are done between the membranes using plug-ins added to the network nodes. Between two nodes with a plug-in from two different membranes, a connection tunnel will allow transfers to be made. The protocol is designed to perform transfers between blockchains (membranes) but not to perform asset swaps.

Table 3.3 draws a parallel between the different approaches presented so far concerning *cross-chain swap* algorithms. The table is divided into three parts, corresponding to the various algorithm approaches: HTLC-based algorithms, verifiable proof algorithms and alternative solutions. The table informs whether the algorithms can be applied in a non-blockchain environment, as for the articles [102, 176]. Furthermore, through this table, it is defined which article provides a problem specification that is independent of the implemented protocol. Indeed, the articles define a specification (a set of properties) that requires knowledge of the underlying protocol, making it a non-generic specification. In [102], the authors assert 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 [176]. Conversely, the specification in [191] is completely protocol-agnostic. The analysis of these articles revealed whether the Byzantine participants' presence was taken into account. Some papers do not mention it like in [43, 88, 95], but knowing that the algorithm assumes synchronous communication (i.e. HTLC's approach), we can consider that there a non-BFT protocols. In addition, the author in [101] assumes synchronous communication by timelocks defined in the HTLCs. Thereby, if a rational user is slow because of a problem in the network, the user may end up worse. The authors in [195] mentioned their protocol vulnerability against attacks such as the Eclipse attack [100]. The attack consists of controlling a sufficient number of Bitcoin addresses to monopolise the connections of those addresses.

The authors in [102, 176] contradict what is said in [191] and show that atomicity in a system in the presence of Byzantine participants cannot be atomic. They define an algorithm that can perform cross-chain transfers through intermediaries without asserting atomicity. The authors in [102, 191] describe the protocol in natural language without applying a formal approach, unlike [176], where an automaton defines the protocol. 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. It is necessary to define the steps of an algorithm precisely via a formal description to avoid any ambiguity and prove the correctness of the approaches.

Approachs	References	Blockchain agnostic	Protocol agnostic specification	BFT protocol	Formal description
HTLC	*Herlihy* [101]	✗	✓	✗	✗
	Decker et al. [68]	✗	✗	✗	✗
	Shadab et al. [169]	✗	✗	✗	✓
	Zie et al. [195]	✗	✓	✗	✓
	Imoto et al. [108]	✗	✗	✗	✓
Verifiable Proofs	*Zakhary et al.* [191]	✗	✓	✗	✗
	Herlihy et al. [102]	✓	✗	✓	✗
	Gugger [95]	✗	✗	–	✓
	Lan et al. [126]	✗	✗	✓	✓
	Zamyatin et al. [193]	✗	✗	✓	✓
	Garoffolo et al. [88]	✗	✗	–	✗
	Borkowski et al. [43]	✗	✗	–	✗
Alternative solutions	*Amiri et al.* [19]	✗	*	✓	✓
	He et al. [99]	✗	✗	✗	✗
	van Glabbeek et al. [176]	✓	✗	✓	✓

✓: the criteria is fulfilled

✗: the criteria is not fulfilled

–: not specified

*: the specification depends on the system

Table 3.3 – Various *cross-chain swap* algorithms

3.2.4 Applying Formal Methods on *Cross-Chain Swap* Algorithms

To date, very little work has focused on the formal verification of such protocols hindering their safe application [3], more so with Byzantine participants. 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 with formal tools [113].

The authors in [101, 191] prove their presented algorithm by applying the method of pen and paper proof without using any automatic or semi-automatic tools. In the current literature, to our knowledge, the only proof approach using formal methods is that of [175]. The authors verify atomic swap smart contracts using model-checking. The model considers two participants, Alice and Bob, in the swap, and they study the possible strategy of the participants to terminate the swap. The paper describes the specification of an atomic swap step by step by considering Alice and Bob's possible strategy to reach the desired goal, assuming that both participants can act dishonestly. They use the model checker MCK [87] to perform their verification. In this paper, the authors highlight the insufficient verification of smart contracts alone and the need to reason about user strategies in a multi-agent environment. The authors use Herlihy's swap model with one contract for each exchange.

3.2.5 Our Contribution On *Cross-Chain Swap* Algorithms

This thesis specifies a *cross-chain swap* problem formally, resilient to Byzantine failures. We define safety and liveness properties that guarantee no correct participant will be worse off in an asynchronous system. The formal specification separates the swap problem from the protocol in a clear way. In addition, we provide an abstract swap protocol formally proved, inspired by [191], that satisfies the swap specification. The protocol relies on an abstraction that we call "*proof-of-action*" to cope with Byzantine participants related to verifiable proofs defined in [102]. The definition of

the specification and the protocol is in Chapter 6. The protocol formal proof is found in Chapter 7, where it consists of performing a semi-automatic tool called TLA$^+$ [120] on the protocol.

Moreover, the defined protocol abstracts the blockchain enough to suit other distributed ledger frameworks aiming to perform a *cross-chain swap*. We illustrate, in Chapter 8, how the described abstract protocol can be instantiated in a blockchain system.

These contributions have been presented and published in the proceedings of peer-reviewed conferences [147].

3.3 Conclusion

This chapter has reviewed several articles that are related to our research topics. Of course, we have not made an exhaustive list of all the articles dealing with verifying smart contracts and the different algorithms executing cross-chain transactions. We wanted to show the readers a shallow overview of the existing approaches in these two fields, smart contract verification and *cross-chain swaps*. The main subject of this thesis is the formalisation and verification of blockchain systems, and we focus on smart contracts and cross-chain applications systems. The verification approach of smart contracts is described in Chapter 5. This chapter highlights the advantage of using a deductive approach to verification rather than model-checking. As a result, more precise modelling of smart contracts is desirable to address more ambitious verification and validation issues since model-checking faces combinatorial explosions. This conclusion led us to consider applying deductive verification, which has the advantage of being less dependent on the size of the state space. Although more complicated for the developer as it is asked to write the invariants.

In addition, in this thesis, we provide a *cross-chain swap* protocol that allows the transfer of assets across different distributed ledgers in the presence of Byzantine participants. The description of the protocol is defined in Chapter 6. Protocols based on HTLC require that participants be connected during the swap process, unlike protocols based on proofs. Moreover, a very slow participant who follows its protocol is considered faulty in protocols using HTLC that require synchronous communication. Our approach does not allow such a result and guarantees that a slow participant will never be worse off. As a result, our protocol follows the approach of the verifiable proofs to be satisfied in a system that assumes asynchronous communication. Unlike several articles like [108] and [101], which guarantee the atomicity property in a synchronous system, our swap problem does not argue to be atomic.

Chapter 4

Tools

" Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less. "

— Marie Curie

Contents

This chapter aims to provide the reader with technical notions helpful in understanding the rest of the thesis. It includes the definition of mathematical notations such as logical operators, temporal logic, and temporal properties formulation (Section 4.1). In addition, this chapter introduces the tools that we use in Chapters 5 and 7, which are Why3 (Section 4.2) and TLA$^+$ (Section 4.3). A tutorial approach is used to understand how each work and their verification approach. When reading the technical chapters, the reader is encouraged to refer to this.

4.1 Mathematical Logic Notations

4.1.1 Modal Logics

Modal logic [107] refers to the enrichment of standard formal logic where the standard operations (and, or, not, implication) are accompanied by certain extra operations – called modal operators. The language of basic modal logic is that of propositional logic, defined as follows.

Propositional logics. In propositional logic [107], an expression is represented by a symbol whose relationship with other expressions is defined via a set of logic operators. A logical operator is a symbol or word used to connect two or more expressions. This type of logical expression is also known as a boolean expression because they create a boolean answer or value when evaluated.

Let b a proposition value. b has one of two possible values denoted *true* and *false*, $b \in \{true, false\}$. The negation of b is denoted by $\neg b$. If $b = true$, then $\neg b = false$; thus, the statement $\neg b$ is true if and only if b is false. We define two propositions values b_1 and b_2. The statement $b_1 \wedge b_2$ is true if b_1 and b_2 are both true; otherwise, it is false. The statement $b_1 \vee b_2$ is true if b_1 or b_2 (or both) are true; if both are false, the statement is false. The implication is denoted by $b_1 \implies b_2$. The implication is false if and only if b_1 is true and b_2 is false; otherwise, it is true. Table 4.5 gives the truth table of the operators, negation 4.1, conjunction 4.2, disjunction 4.3 and implication 4.4.

Table 4.1 – Negation

b_1	$\neg b_1$
false	*true*
true	*false*

Table 4.2 – Conjunction

b_1	b_2	$b_1 \wedge b_2$
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Table 4.3 – Disjunction

b_1	b_2	$b_1 \vee b_2$
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*true*

Table 4.4 – Implication

b_1	b_2	$b_1 \implies b_2$
false	*false*	*true*
false	*true*	*true*
true	*false*	*false*
true	*true*	*true*

Table 4.5 – Truth tables of logical operators

Predicate logic. The predicate logic, also called *First-Order Logic* [107], extends propositional logic with *quantification*. Quantification is the ability to assert that a specific property holds for all elements or some element. In propositional logic, the expressions are either $true$ or $false$ variables. An expression in predicate logic is a predicate that asserts a relation between variables. For example, the expression $P(x)$ evaluates the proposition x, where P is a predicate. In predicate logic, expressions that reason about *every* or *some* variables are possible using the *quantifiers*. The symbol $\forall$ is used to indicate a *universal quantification*. $\forall n \in \mathbb{N} : P(n)$ means $P(n)$ is true for all natural numbers n. The symbol $\exists$ is used to indicate *existential quantification*. $\exists n \in \mathbb{N} : P(n)$ means at least one natural number n such that $P(n)$ is true.

We can combine logical operators with quantifiers to express expressions in predicate logic. For example, let us define the sentence "*Not all animals can swim*" in predicate logic. We need to define two predicates, A and S, that have one argument: "$A(x) : x$ *is an animal*" and "$S(x) : x$ *can swim*". Therefore, the sentence can be formally defined as $\neg(\forall x(A(x) \rightarrow S(x)))$.

The following grammar defines a predicate logic formula (φ):

$$\varphi ::= \quad P(t_1, t_2, ..., t_n) \quad | \quad (\neg\varphi) \quad | \quad (\varphi \wedge \varphi) \quad | \quad (\varphi \vee \varphi) \quad | \quad (\varphi \rightarrow \varphi) \quad | \quad (\forall x\ \varphi) \quad | \quad (\exists x\ \varphi)$$

where x is a variable, P is a predicate symbol and t_i are terms over a set of function symbols.

Temporal logic. Temporal logics [107] are special cases of modal logic, which are formalised in several ways. The idea of temporal logic is that a formula is not statically *true* or *false* in a model, as it is in propositional and predicate logic. Temporal logic is a logic for specifying properties over time like the behaviour of a finite-state system. Temporal logic is widely used in formal verification [156], where the basic technique is essentially model-checking. For example, it can express that a dangerous event must not occur until a particular safety condition is satisfied. There exist various kinds of logic, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [107].

Linear temporal logic. LTL formula is properties that refer to the future over a single computation path. An LTL temporal logic formula is built with propositional variables, logical operators and temporal operators. The following grammar defines the well-formed LTL formulas (φ):

$$\varphi ::= \quad p \quad | \quad (\neg\varphi) \quad | \quad (\varphi \wedge \varphi) \quad | \quad (\varphi \vee \varphi) \quad | \quad (\varphi \rightarrow \varphi) \quad | \quad (X\varphi) \quad | \quad (F\varphi) \quad | \quad (G\varphi) \quad | \quad (\varphi\, U\, \varphi)$$

where p is a propositional variable and X, F, G, and U are temporal operators. X stands for *next*, F for *eventually* (finally), G for *always* (globally) and U for *Until*. X, F, and G are also defined by the symbols: $\bigcirc$, $\Diamond$, and $\Box$. Let us define, in the following, examples of LTL propositions over a sequence of states:

- $\Box p$ means that p will always hold, at any time, and on the entire subsequent path (Figure 4.1).
- $\Diamond p$ means that p eventually has to hold, some time in the future, along some subsequent path (Figure 4.2).
- $\bigcirc p$ means that p holds at the next state (Figure 4.3).

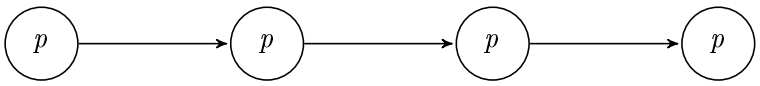


Figure 4.1 – p holds on the entire path

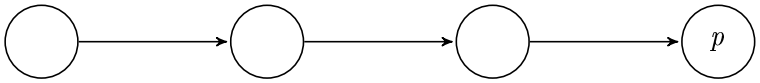


Figure 4.2 – p holds some time in the future

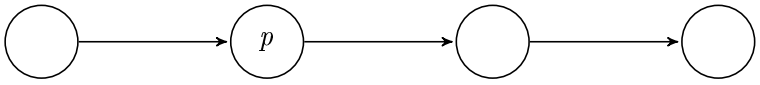


Figure 4.3 – p holds at the next state

Computation tree logic. CTL formulas are properties expressed over a tree of all possible executions meaning that there are different paths in the future. It is built with all LTL operators in addition to path quantifiers (∀ and ∃). Thus it is possible to combine temporal operators with quantifiers and obtain the CTL formula grammar:

$$\varphi ::= p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (AX\varphi) \mid (EX\varphi) \mid (AF\varphi) \mid (EF\varphi) \mid (EG\varphi) \mid (A[\varphi U \varphi]) \mid (E[\varphi U \varphi])$$

where p is a propositional value, and the five first symbols are similar to LTL grammar. The symbols A and E are another way of expressing ∀ and ∃ meaning, respectively, "along all paths" and "along at least one path". Examples of such properties are the following:

- AGp (or $\forall\Box p$) means that the property p holds along all computation paths starting from the state where AGp holds (Figure 4.4).
- EGp (or $\exists\Box p$) means that the property p holds along at least one path starting from the state where EGp holds (Figure 4.5).

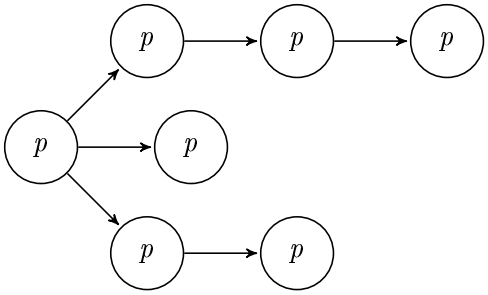


Figure 4.4 – p holds in all possible path execution

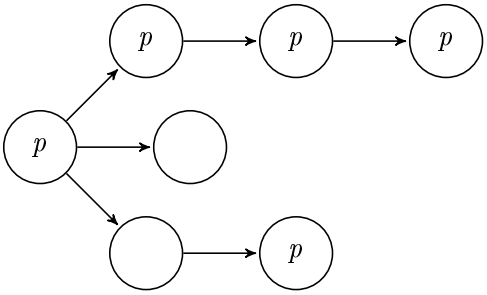


Figure 4.5 – p holds at least in one path execution

4.1.2 Hoare Logic and Weakest Precondition Calculus (WP)

Hoare logic uses the first-order logic formula to present the program logic and express the program's properties. The main feature of Hoare logic is the Hoare triple denoted $\{P\}s\{Q\}$ where P and Q are logic propositions and s a statement. P is called *precondition*, and it characterises a condition that must be true before beginning a function within the program. Q is a *postcondition* and says what is true at the end of the program s. Thus, the Hoare triple means that when the precondition P is met, executing the statement s establishes the postcondition Q. Hoare logic is relational such that for each Q, there are many P, and for each P, there are many Q. Using standard Hoare logic ensures *partial correctness* of programs, i.e. termination cannot be proved. Partial correctness can be considered a weak requirement since any program that does not terminate satisfies the postcondition, e.g. infinite loops. Partial correctness says what must happen if the program terminates. The property that requires the program to terminate is the *total correctness*. We say that the Hoare triple satisfies the total correctness if for all states in which s is executed and satisfies the precondition P, s is guaranteed to terminate, and the resulting state satisfies the postcondition Q.

The Weakest Precondition Calculus is a technique that comes from Dijkstra [69] to prove imperative programs' properties. WP is about evaluating a function; thus, given a statement s and a postcondition Q, the goal is to find the unique precondition P – the weakest precondition for s and Q.

The weakest precondition for s to Q is an assertion that is true for precisely those initial states from which s must terminate and execute s must produce a state satisfying Q. Unlike Hoare logic, WP is functional, and for each Q, there is precisely one assertion P that equals $wp(s, Q)$.

Consequently, WP does respect Hoare logic and the formula $\{wp(S, Q)\}S\{Q\}$ is true. WP ensures total correctness and proves the termination of the program.

Here is an example to understand the difference between the two logic. We consider the statement $x := x + 1$ with its postcondition $x > 0$. Considering a Hoare triple, a valid precondition is $x > 0$, so the following formula is true:

$$\{x > 0\}x := x + 1\{x > 0\}$$

Another valid precondition is $x > -1$ that also satisfies the following formula:

$$\{x > -1\}x := x + 1\{x > 0\}$$

We can conclude that the precondition $x > -1$ is weaker than $x > 0$ because $x > 0 \implies x > -1$. As a result, the precondition $x > -1$ is the weakest precondition of the program defined by the formula:

$$wp(x := x + 1, x > 0) = x > -1$$

Invariant loops. A function that defines a loop in its program cannot be proven by defining pre- and postconditions alone. In addition, we cannot mechanically generate the weakest precondition. Instead, we must reason about the loop inductively and define an inductive proof. The proof shows that each time the loop executes, we get one step closer to the final result and that when the loop terminates, we obtain the expected result. This inductive proof is called *loop invariants*.

When proving a program using Hoare logic [105], guessing the appropriate loop invariants is a significant difficulty. In formal verification, especially in Hoare logic, loop invariants are logical predicates used to prove the correctness of algorithms. As a result, one needs to discover a suitable loop invariant to prove the pre- and postconditions of our program. Remark that the knowledge of completeness gives only hints on effectively determining a suitable loop invariant when required.

A sound loop invariant should satisfy three properties:

1. *Initialisation*. It should be true before the first iteration of the loop.
2. *Preservation*. If the invariant is true before an iteration of the loop, it must be true after the iteration.
3. *Termination*. When the loop terminates, the invariant must give useful information to show that the algorithm is correct.

4.2 *Why3*

Why3 tool [83] is a platform for deductive program verification. It provides a rich language for specification and programming, called *WhyML*, and can be used as an *intermediate* language. An intermediate language is platform-independent; thus, it can be run in any computer environment that has a runtime engine for the language. Moreover, the logical language of *Why3* does not depend on the programming language. It can serve as a standard format for theorem proving problems, readily suitable (via *Why3*) for multiple automated and interactive provers, such as Alt-Ergo [41], CVC4 [30], Z3 [67] and Coq [62]. *Why3* comes with a standard library[1] of logical theories and programming data structures.

The logic of *Why3* is first-order logic with polymorphic types and several extensions: recursive definitions, algebraic data types and inductive predicates. In addition, first-order language is extended, both in terms and formulas, with pattern matching, let-expressions, and conditional (*if-then-else*) expressions. This approach defines properties as preconditions, postconditions, asserts and invariants. The development of *Why3* is mainly motivated by the necessity to model the behaviour of programs and formally prove their properties.

[1] http://why3.lri.fr/

Pure logical definitions, axioms and lemmas are organised in collections called *theories*. The standard library of *Why3* contains numerous theories describing integer and real arithmetic, lists, binary trees, mappings, and abstract algebraic notions. In *WhyML*, a type, a function, or a predicate can be given a definition or just declared abstract symbols and then axiomatised. A *WhyML* file contains modules, and each module contains declarations. This declaration can be program data types, logical declarations as types, functions, predicates, axioms, lemmas and constructs as sequences, loops, and exceptions.

Why3 allows expressing ghost expressions in a program by using the keyword `ghost`. It marks the expression as ghost code added for verification, i.e. only for the specification or proof. Ghost code is removed from the code intended for execution (it is not part of the executable code). Thus, it cannot affect the computation of the program results nor the content of the observable memory. As a consequence, ghost code cannot interfere with regular code in the following sense:

- Ghost code cannot modify regular data, but it can access it in a read-only way.
- Ghost code cannot modify the control flow of regular code.
- Regular code cannot access or modify ghost data.

Why3 **provers.** The *Why3* tool is equipped with an intuitive graphical interface that allows applying certain operations such as splitting a proof and verifying a goal's validity by calling the desired prover. The principal activity of *Why3* can be described as processing proof tasks. A task is a logical context: a list of declarations followed by one goal, that is, a formula. Tasks are extracted from the various theories. When a goal is sent to a prover that does not support some language features, *Why3* apply a series of encoding transformations, for example, to eliminate pattern matching or polymorphic types. Another example is if the target prover is Z3, *Why3* will apply a transformation to remove inductive predicates since Z3 does not handle it. *WhyML* functions are annotated with pre- and postconditions for normal and exceptional termination, and *WhyML* loops are annotated with invariants. While-loops and recursive functions can be given variants (i.e. values that decrease at each recursive iteration or call) to ensure termination. We can insert assertions (statically checked) at arbitrary points in a program. Verification conditions are generated using a standard weakest precondition procedure. Functions, predicates and pure types introduced in the logical language can be used in the program. For example, the type of integers and basic arithmetic operations are shared between specifications and programs.

4.2.1 Structure of a *Why3* Program

Modules. Program declarations and theories are grouped into modules. *Why3* depicts a standard library where a set of theories are defined. It is possible to use a theory by using `use import` (or `use` depending on the language version) following its name. For example, the standard library contains a module `Fact`, where a factorial function is defined. The module is as follows:

```
1  module Fac
2   use Int
3
4  let rec fact_rec (x:int) : int
5   requires { x ≥ 0 }
6   variant  { x }
7   ensures  { result = fact x }
8    = if x = 0 then 1 else x * fact_rec (x-1)
9
10 end
```

A module starts with `module` and ends with `end`. It begins by importing theory `Int` for the integer numbers, then defines the factorial function. If we want to use the recursive function `fact_rec`, we must import the logical theory `use import int.Fact` into the current context that is theory `Fact` from *Why3* standard library [2].

[2] http://why3.lri.fr/stdlib/int.html

In *WhyML* syntax, we can define recursive functions using the `let rec` construct like the function `fact_rec`. The function defines a precondition that the function's input must be positive or null. Within the postcondition, the variable `result` stands for the value returned by the function. To prove the termination of this function, we must define a variant that is a term that decreases at each recursive call for a well-founded order relation.

The *Why3* standard library contains many theories and modules, thus enabling the modularity that avoids rewriting certain types or expressions. Another useful module from the standard library of *WhyML* is the module `Ref`. The module provides references enabling mutable variables. A reference is created with function `ref`, we access the contents of reference `x` with `!x` and assign it with `x := e`. Notice that the same symbol (!) is used for both a pure access function and a program function. Since program symbols cannot appear in specifications, `!r` in pre-and postconditions can only refer to the pure function. In the program code, `!r` will refer to the *WhyML* function. An exception is made for logic functions and predicates specified directly on program types. These functions and predicates have uncontrolled access to ghost components of program types; therefore, they can only be used in specifications.

Machine integers. Previously, we said that *Why3* could be used as an intermediate language using *WhyML* when needed. So far, we have shown the possibility of using arithmetic from *Why3* standard library, with the type `int` of mathematical integers with unbounded precision. However, assume we need to model machine arithmetic (like, signed 32-bit integers) to show the absence of arithmetic overflow in a program or reason about possible overflows. The main difficulty is that we do not want to lose the arithmetic capabilities of SMT solvers (which only know about mathematical arithmetic). One way to do this is to introduce a new, uninterpreted type `int32` for machine integers "`type int32`", together with a function giving the corresponding value of type `int`: "`function toint int32 : int`". The idea is to use only type `int` in program annotations, that is, to apply function `toint` systematically around sub-expressions of type `int32`. If our purpose is to build a model to prove the absence of arithmetic overflow, we need a function to build a value of type `int32` from a value of type `int` with a suitable precondition.

Therefore, the standard library provides a generic theory in module `Bounded_int` [3] that can instantiate integers to n-bit signed and unsigned integers by giving them a minimal and a maximal value. The instantiation is possible thanks to the specifications modularity. In *WhyML*, we can refer to a module by the mean of "cloning".

Notice an important difference between `use` and `clone`. If we use a theory, say `List`, twice, there is no duplication; hence, there is still only one type of list and a unique pair of constructors. On the contrary, a `clone` declaration constructs a local copy of the cloned module, possibly instantiating some of its abstract symbols. Despite having the same names, the newly created symbols are different from their originals. This cloning mechanism is very useful since we can define a module as general as possible. Then we can implement it and verify it only once and then reuse it in different contexts.

The theory `Bounded_int` defines a non-interpreted type `t` with two constants, `min` and `max`. The declarations inside the theory are dependent on the type `t`. Moreover, `Bounded_int` defines a set of functions for mapping a value `int` to a value `t` and conversely. The module also defines classical integer functions, like the addition and subtraction, between two `t` values. For example, the following `val` function

```
1 val to_int (n:t) : int
2  ensures { result = n }
```

corresponds to a value given as input (`n`) to an integer (the function `result`). Thus, if we wish to instantiate a machine integer type, we must clone the theory `Bounded_int` in the current context

[3] http://why3.lri.fr/stdlib/mach.int.html

with the necessary information. If we take the example of the `Int32` integer, the cloning is done as depicted in Listing 4.1.

```
module Int32

  use int.Int

  type int32 = < range -0x8000_0000 0x7fff_ffff >

  let constant min_int32 : int = - 0x8000_0000
  let constant max_int32 : int =   0x7fff_ffff

  clone export Bounded_int with
    type t = int32,
    constant min = min_int32,
    constant max = max_int32,
end
```

Listing 4.1 – Example of cloning mechanism

Types. A type can be an algebraic data type, an alias for a type expression or non-interpreted. For example, the type of polymorphic binary trees is introduced as follows:

```
type tree α = Leaf | Node (tree α) α (tree α)
```

The symbol (`|`) represents an enumeration. Built-in types include integers (`int`), real numbers (`real`) and polymorphic tuples. Record types are a particular case of algebraic types with a single unnamed constructor and named fields. Here is a definition of a generic queue with two fields:

```
type queue α = { front: list α;
                 rear: list α }
```

In *Why3* standard library, we find the following algebraic data types:

```
type bool = True | False
type option α = None | Some α (in option.Option)
type list α = Nil | Cons α (list α) (in list.List)
```

`None` stands for the absence of value, `Some` stands for an entry α, `Nil` for an empty list, and `Cons` for an entry list α.

Why3 standard library provides arrays in module `array.Array` [4]. This module declares a polymorphic type `array` α, an access operation written `a[e]`, an assignment operation `a[e1] <- e2`, and various operations such as `create`, `length`, `append`, `sub`, or `copy`. The type being abstract in programs, we cannot implement operations over this type, but we can declare function prototypes to provide a usable interface. For example, the access operation of an array is declared as follows:

```
val ([]) (a: array α) (i: int) : α
requires { 0 ≤ i < length a }
ensures { result = a[i] }
```

This function takes `a` and `i` as arguments, together with a precondition to ensure array access within bounds. It returns a value of type α, and the postcondition states that the returned value is the value contained in the array at the index `i`.

In addition, the assignment operation is declared in a similar way:

```
val ([]←) (a: array α) (i: int) (v: α) : unit writes { a }
requires { 0 ≤ i < length a }
ensures  { a.elts = Map.set (old a).elts i v }
ensures  { a = (old a)[i ← v] }
```

The main difference is that the annotation `writes {a}`, which indicates that a call to this function modifies the content of `a`. The modification is allowed since the field `elts` was declared to be mutable. The term `((old a).elts)` in the postcondition refers to the pre-call value of the field `a.elts` before it is modified by `([]<-)`.

[4] http://why3.lri.fr/stdlib/array.html

Function symbol	Definition
`let`	A program function, with prototype, contract, and body.
`val`	A program function, with prototype and contract only.
`let function`	A pure program function which can be used in specifications as a logical function symbol.
`let predicate`	A pure boolean program function which can be used in specifications as a logical predicate symbol.
`val function`	A pure program function which can be used in specifications as a logical function symbol.
`val predicate`	A pure boolean program function which can be used in specifications as a logical predicate symbol.
`function`	A logical function symbol which can be used as a program function in ghost code.
`predicate`	A logical predicate symbol which can be used as a boolean program function in ghost code.
`let lemma`	A special pure program function which serves not as an actual code to execute but to prove the function's contract as a lemma.

Table 4.6 – Functions declaration from the *Why3* manual [185]

Functions in *WhyML*. *WhyML* language allows several different functions to be defined with varying declarations. Functions introduced by the "`function`" keyword are pure functions[5] that can be used in both specification and program, whereas functions introduced by "`let`" can only be used in the program. Another difference is that functions introduced by the `function` keyword cannot be annotated, but the provers can access their body. In contrast, functions introduced by the `let` keyword are black boxes that are only seen by provers through their specification. Every function or predicate symbol in *Why3* has a (polymorphic) type signature. For example, an abstract function that merges two integer trees can be declared as follows:

```
function merge (tree int) (tree int) : tree int
```

Moreover, functions and predicates can be given definitions, possibly mutually recursive. For instance, a recursive function that can calculate the height of a tree is defined as follows:

```
function height (t: tree α) : int =
match t with
| Leaf → 0
| Node l _ r → 1 + max (height l) (height r)
end
```

And, the definition of a recursive predicate `mem` checking for the presence of an element `x` in a list `l`:

```
predicate mem (x: α) (l: list α) = match l with
| Nil → false
| Cons y r → x = y  ∨  mem x r
end
```

Table 4.6 defines the different functions that can be expressed in a *WhyML* program.

4.2.2 Proving Euclidean Division Using *Why3*

In the following, we proceed step by step by building the conventional *Euclid's algorithm* to illustrate how *Why3* works. The algorithm's goal is to calculate the division of two integers producing a quotient and a remainder [6]. First, we create a module and name it `Division`. This module will contain the program.

[5] In the sense that they cannot mutate some state or perform any side effects. There are many other limitations on what functions can be defined using the function keyword. For example, they cannot feature loops.

[6] The example can be found in http://toccata.lri.fr/gallery/division.en.html.

```
1 module Division
```

We import the theory of integers `int.Int` from *Why3* standard library — the prefix `int` indicates the file in the standard library containing theory `Int`. Then, we need the usual operations on references for mutable variables. Therefore, we import the theory `Refint`.

```
1   use int.Int
2   use ref.Refint
```

Now that we have all the necessary modules to write our program, we can define the Euclid algorithm. The program takes two integers, `a` and `b`, as parameters and returns an integer; the quotient `q`. We initialise two local variables, the quotient `q` to 0 and the remainder `r` to the value of `a`. To calculate the quotient, we need a loop on the program. While the remainder is greater than the divisor `b`, we increment the quotient and subtract to `r` the value of `b`.

```
1     while r ≥ b do
2       incr q;
3       r -= b
4     done;
5     q
```

The function `incr` comes from the theory `Refint`. The quotient is calculated once the remainder is less than `b`, and the program returns its value (line 5).

We have built the program; now, we have to prove it. Euclidean division is based on the following theorem: "*Given two positive integers* `a` *and* `b`, *with* `b` $\neq 0$, *there exist unique integers* `q` *and* `r` *such that:* $a = b * q + r$ *and* $0 \leq r < |b|$, *where* $|b|$ *denotes the absolute value of* b." This theorem is used to define the program's precondition, noted `requires`, and postcondition, noted `ensures`.

```
1     requires { 0 ≤ a && 0 < b }
2     ensures  { exists r. result * b + r = a && 0 ≤ r < b }
```

Moreover, we need to define a loop invariant, introduced with the keyword `invariant`, to prove that a property is true before and after each loop iteration. It can help to prove the correctness of the program. In the example of Euclid, the property to hold is the conservation of the relationship between the four integers, and the remainder must be positive. Proving that a loop always terminates is a common requirement when verifying software. The usual approach provides a loop variant function introduced with `variant`. The `variant` declaration tells *Why3* to check for structural decreasing, as it does for logical definitions. Usually, an integer expression that decreases on every loop iteration is used to define a variant declaration. In the example, the integer that decreases is `r`. The complete program of the Euclid division algorithm is in Listing 4.2.

```
1  module Division
2
3    use int.Int
4    use ref.Refint
5
6    let division (a b: int) : int
7      requires { 0 ≤ a && 0 < b }
8      ensures  { exists r. result * b + r = a && 0 ≤ r < b }
9    =
10     let ref q = 0 in
11     let ref r = a in
12     while r ≥ b do
13       invariant { q * b + r = a && 0 ≤ r }
14       variant   { r }
15       incr q;
16       r -= b
17     done;
18     q
19
20 end
```

Listing 4.2 – Euclidean division in *WhyML*

We can prove the program by launching the *Why3* GUI, as illustrated in Figure 4.6. We can see from the figure that we have a tree view on the left side. It shows the properties we have proved (green check). On the right side of the figure is the program written in *WhyML*.

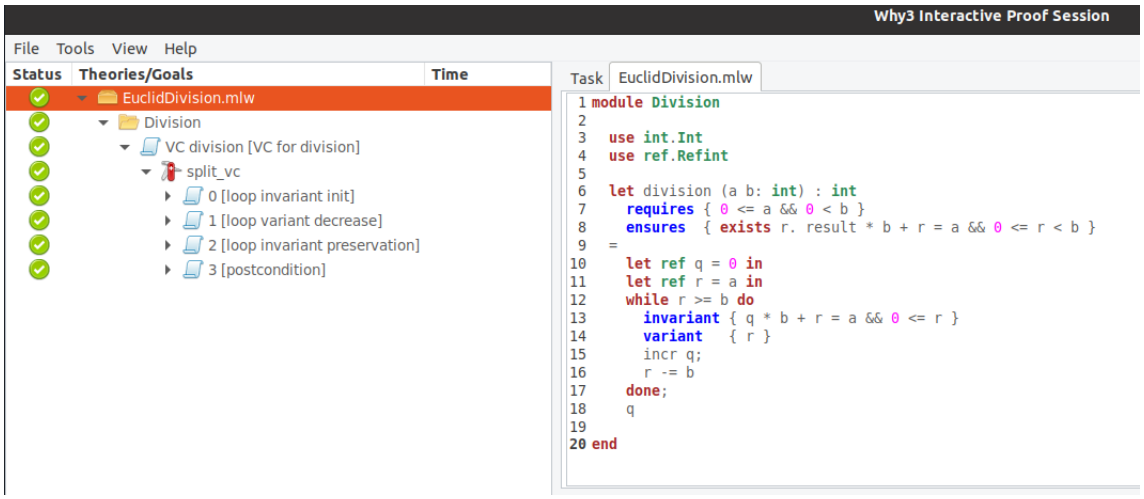


Figure 4.6 – The *Why3* GUI

We invite the reader to refer to the project web page (`http://why3.lri.fr`) for a complete presentation of *Why3* and *WhyML*, which provides a detailed introduction and a large collection of examples.

4.3 TLA$^+$

TLA$^+$ is a specification language based on the Temporal Logic of Action (TLA) [119] and Zermelo-Fraenkel's (ZF) set theory [141]. On the one hand, this combination brings the possibility to describe dynamic behaviours of state-transition systems through TLA and, on the other hand, a way to specify the system's data structure through ZF. Zermelo-Fraenkel's set theory with the axiom of choice is considered the standard foundation for mathematics.

Leslie Lamport designed TLA$^+$ in the 90s to specify and model concurrent and distributed systems. TLA$^+$ is a well-known tool and is frequently used by the industry as Amazon [150], Intel [33], Microsoft [2] and OpencomRTOS [179].

A TLA$^+$ specification is structured as a module that can extend other modules using the keyword $EXTENDS$. A system is composed of variables $x_1, ..., x_n$. The system is represented as actions that, when carried out, move it from one state to another. A state is assigning a value to the system's set variables. Thus, an action is a relation between two states through a predicate over variables $x_1, ..., x_n$ and $x'_1, ...x'_n$. Unprimed variables refer to the value of variables in the current state, and the primed variables refer to the value of variables in the next state of the system. Therefore, a TLA$^+$ predicate describes a behaviour of the system when the predicate evaluates to *true*.

A TLA$^+$ system is specified as $Spec = Init \wedge \Box[Next]_{vars}$. 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 $\Box[Next]_{vars}$ means it is always true that either one of the actions defined in $Next$ is executed or $vars$ is in a state of stuttering. 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 at each step, either an action is true, and the state changes or $vars$ stutters.

4.3.1 The TLA$^+$ Verification Tools

The model checker TLC. To verify the specification of a system written in TLA$^+$, the tool relies on a model checker called TLC [190]. TLC is an explicit-state model checker that performs a breadth-first search to traverse the state graph for checking invariance properties. To reduce the combinatorial explosion problem, well known in model-checking, TLC uses a state compression

method by using fingerprints. This method reduces the amount of space required during the model-checking process, thereby reducing space complexity.

When *TLC* is launched, it first generates the initial states that satisfy the specification and verifies all invariant properties. The execution stops when all state transitions lead to states already discovered. If *TLC* faces a state that violates a system property, the execution halts, and *TLC* returns a counter-example that traces the violation's path.

The proof system TLAPS. TLA$^+$ was extended, in 2012, by the introduction of a proof system *TLAPS* [56]. *TLAPS* is an interactive proof environment where users can deductively verify safety properties written in TLA$^+$. Verification by theorem proving, provided with *TLAPS*, meets the need to strengthen the correctness property of an inductive invariant. For example, proving that a safety property is an invariant of the system comes down to defining an inductive invariant that implies the safety property. The proof system uses declarative notation to write hierarchical proofs that are mechanically checked by generating proof obligations and passes them to back-end verifiers like Isabelle [184], Zenon [42] and SMT solvers. Z3 [67] is the SMT solver distributed with *TLAPS*. However, it is possible to add other SMT solvers by downloading and installing them, for example, CVC4 [30]. By default, *TLAPS* does not reprove an obligation that it has already proved. The Proof Manager computes a fingerprint of every obligation. The fingerprint is a compact canonical representation of the obligation and the relevant part of its context.
TLAPS allows the user to decompose a complex proof into smaller proofs until they can be provable by the available back-end provers. The *TLAPS* proof language is prover-independent, and all reasoning is done at the TLA$^+$ level. Therefore, users do not need to have any knowledge of back-end provers.

A proof in *TLAPS* is built on the specification written in TLA$^+$. A TLA$^+$ module contains declarations, assertions and definitions. Assertions state valid facts which do not need to be proven. Assertions can be expressed through AXIOM or ASSUME keywords. Other formulas such as theorems THEOREM and lemmas LEMMA can be expressed. They assert that facts are provable in the current context.

A hierarchical proof is either a *leaf* proof established by elementary steps that indicate the known facts and definitions of the desired goal or *sequences of assertions* followed by QED . Each proof in the hierarchy ends with a QED step that asserts the proof's goal. Note that definitions and facts must be cited explicitly for *TLAPS* to use them. The *TLAPS* standard module defines some operators that are used when writing proofs to be checked by the *TLAPS* proof system like *PTL* (for Propositional Temporal Logic), *SMT* and other back-ends.

4.3.2 PlusCal

TLA$^+$ language can become challenging if we have no background in TLA$^+$ formalism. In order to make it easy for inexperienced users to use TLA$^+$, PlusCal [121] has been proposed. It is a high-level language for describing concurrent and distributed algorithms in the form of pseudo-code. The PlusCal language expresses simple statements while being quite powerful and allows interesting features such as non-determinism, procedures, and grain of atomicity, i.e. atomic actions. PlusCal is a handy language for those who do not want to master TLA$^+$ specification but still want to use the underlying technology of TLA$^+$ as the model-checking. However, using PlusCal for checking a model or proving a program requires a minimum understanding of TLA$^+$ language. Indeed, properties to verify are written according to the TLA$^+$ specification.

PlusCal is designed to express concurrent systems, allowing multiprocessor algorithms, each with its own definition. The PlusCal language provides interesting features to design algorithms. We find familiar imperative language constructs such as **while** loop to express repetitive algorithm instruction, **either -or** and **with** to express non-deterministic behaviours and **if - then -else** instructions. The language also provides constructs that allow conditioning the behaviour of a process using **when** or **await** statement. This construct can impose a synchronisation of processes that wait for a condition to be true.

Once written, a PlusCal code is parsed with the PlusCal translator, automatically generating a TLA$^+$ specification. The PlusCal translator inserts the generated TLA$^+$ specification between the *BEGIN* and *END* translation comment lines. Although the tool tries to preserve the variable names, it can create new names if necessary. For example, the translator adds the variable *pc*, for "*program control*", to explicitly track the point of execution of the program that corresponds to which label the process is currently on.

Labels. The grain of atomicity is possible by using labels. A grain of atomicity is the ability to ensure that a block of instruction executes without interleaving other process statements. In PlusCal, as many instructions as possible can be labelled. Instructions between two labels are executed atomically and constitute one action. In addition, it is possible to jump from one label to another using the keyword **goto** followed by the label name. However, one must be careful in the way of using labels. The more labels the program has, the more exact it will be, but significantly increasing possible states. Conversely, few labels reduce the number of possible states but decrease the exactitude of the program. It is a trade-off between clarity and performance. There are specific rules in the placement of labels. It is mandatory to place a label in the following locations:

- At the first line of a process.
- Before a **while** statement.
- Right after a **call** , **return** or **goto** statement. **call** and **return** are used in a procedure context. Their role is to, respectively, call a procedure in the program of a process and to mark the end of a procedure. Although it may look like, **return** does not return any value.
- If one possible branch of a **if** or **either** statement has a label, then the whole control structure must follow with a label.

Note that it is impossible to put a label in a **with** statement or assign a variable more than once in a label.

Fair process. A system satisfies a liveness property under fairness assumptions on actions. Expressing fairness in a concurrent system is an important assumption. In PlusCal, it is expressed using the keyword **fair** before the process definition. In a PlusCal algorithm, each label corresponds to an action. An action is enabled if, and only if it can be executed, i.e. a **fair process** cannot stop at that action. If two actions are enabled, the executed action is non-deterministically chosen. Omitting the word **fair** makes the process unfair and has no fairness assumptions on its actions, which can also reflect a crash process's behaviour. There is two kinds of fairness assumptions: *weak fairness*, using the keyword **fair** ensures that the transition must occur if it remains continuously enabled, and *strong fairness*, using the keyword **strong fair** ensures that a transition must occur if it is repeatedly enabled from time to time.

4.3.3 The *Two-Phase Commit* Protocol in TLA$^+$

Several examples in the literature of the *Two-Phase Commit* protocol are implemented in TLA$^+$. This section defines a version of it. First, we define the algorithm in the PlusCal language, and then we translate it using the PlusCal translator to generate the TLA$^+$ specification.

1. The PlusCal algorithm. The algorithm follows the steps defined in Section 2.1.1. Before we start describing the algorithm in PlusCal, we name the module, MODULE $TwoPhaseCommit$. Because we assign an integer identifier to participants, we need to extend the *Integers* module from the TLA$^+$ library. The total number of participants is defined by a constant N:

```
EXTENDS Integers
CONSTANT N
```

We reason about the states of the participants to build the algorithm, namely the coordinator and the participants (also called followers). Hence, we define a set of possible states in which the participants can be:

```
CStates ≜ { "init", "pre-commit", "commit", "abort" }
PStates ≜ { "working", "committed", "aborted", "prepared" }
```

$CStates$ is the set of the coordinator's states. "init" represents the initial state of the coordinator, "pre-commit" represents the state where the coordinator sends the participants the query to commit, "commit" represents the decision of the commit and "abort" the decision of the abort. $PStates$ is the set of the participants' states with "working" the initial state, "prepared" representing the *yes* vote for the commit, "committed" representing the acknowledgement of the commit decision, and "aborted" representing the *no* vote for the commit.

Participants are given an identifier to track their behaviours and actions efficiently. A unique identifier must identify each participant from the following sets:

```
CoordinatorID ≜ 0
Participants ≜ 1..N
```

The identifier of the coordinator is 0. The identifiers of the other participants are assigned so that there is no participant with the same identifier as the coordinator. Therefore, $Participants$ is the set of all numbers starting from 1 to N, and each number represents a participant identifier.

The PlusCal algorithm is enclosed as a comment in TLA$^+$ using $*..*$, and we give it the name $TwoPhaseCommit$. The algorithm defines and initialises three global variables: the coordinator's state $cState$, the participants' state $pState$, and $abortFlag$. The variables are respectively initialised to "init", "working", and FALSE. $abortFlag$ is used to detect if a participant has voted *no* for the commit.

```
--algorithm TwoPhaseCommit
variables cState = "init",
pState = [p ∈ Participants ↦ "working"],
abortFlag = FALSE ;
```

Remark. The addition of the boolean $abortFlag$ serves to avoid the existential quantifier $\exists$. Indeed, we could have defined a predicate that is true if: $\exists p \in Participants : abortFlag[p] = \text{TRUE}$. It can be complicated to reason with the existential quantifier in proofs [76]. Therefore, we replace it with a true boolean when a participant aborts the commit.

The participants' state $pState$ is defined as a function with $Participants$ as its domain. In TLA$^+$, a function is different from other programming languages. It is closer to hashtables or dictionaries with values in the state space of algorithms. Hence, the domain of the function can be seen as the index set of a dictionary or hashtable. The declaration of variables allows them to be initialised as well.

We choose to model the message sent between participants by updating the state of their variable $cState$ and $pState$. That is to say, if $cState$ is set to "pre-commit", then this translates into "*the coordinator has sent a commit request to all participants*".

The PlusCal language has an optional **define** statement for inserting TLA$^+$ definitions in algorithms. It must come before the definition of the process algorithm. It permits predicates defined in terms of variables to be used in the algorithm's expressions. In our algorithm, $allPCommit$ and $atLeastOneAbort$ are predicates that inform the coordinator on which decision to make.

define {
$allPCommit \triangleq \forall p \in Participants : pState[p] =$ "prepared"
$atLeastOneAbort \triangleq abortFlag =$ TRUE **}**

The predicate $allPCommit$ is valid if all the participants agree to commit. In the classical *Two-Phase Commit*, the agreement is a *yes* sending message. In our PlusCal algorithm, the *yes* message is represented by updating the state $pState$ to "prepared". Conversely, $atLeastOneAbort$ is valid if at least one participant changes the state of the variable $abortFlag$ and assigns it to TRUE. In the following, we describe the behaviour of the coordinator and the participants defined as **process** .

The coordinator process. The coordinator, defined in **Definition** 1, is identified by its signature **fair process** ($Coordinator = CoordinatorID$), with $Coordinator$ the name of the process. The process is assumed fair thanks to the **fair** keyword. The coordinator starts at the label "c0" with a conditional statement using the **await** construct. The condition is that the coordinator must be in the initial state. Once the condition is satisfied, the coordinator moves to the second statement and faces a non-deterministic construct. The coordinator can either execute action $c1$ and abandon the commit by changing its state from "init" to "abort" or query the participant for a prepared to commit by changing its state from "init" to "pre-commit". Note that if the coordinator decides to take the **either** branch, it reaches the end of its program. Indeed, after the $cState$ variable changes state, the program has no further action to perform.

However, suppose it decides to query the participants. The coordinator executes action "c2" after updating its state to "pre-commit". The label "c2" defines a non-deterministic conditional action. The coordinator waits until one of the two predicates defined in the **define** statement is valid. Either $allPCommit$ is valid, meaning that all participants are prepared to commit, or the predicate $atLeastOneAbort$ is valid, meaning that at least one participant has decided to abort. In the first case, the state of the variable $cState$ changes to "commit", which ends the coordinator program. The second case returns the coordinator to the label "c1", which represents the abort state of the coordinator and does not allow the commit to taking place. The last statement uses the keyword **goto** that enables the program to jump to the label "c1". Three possible atomic actions define the coordinator program: $\{c0, c1, c2\}$.

Definition 1 (The coordinator's PlusCal program)**.**

```
fair process (Coordinator = CoordinatorID){
c0:  await cState = "init";
     either {
c1:        cState := "abort"; }
     or {
           cState := "pre-commit";
c2:        either {
                 await allPCommit;
                 cState := "commit"; }
           or {
                 await atLeastOneAbort;
                 goto c1; } ; } ;
} ;
```

The participants' process. Similarly to the coordinator, we define in **Definition** 2 a process for the participants. It is identified by the signature **fair process** ($Participant \in Participants$), with $Participant$ as the name of the process. The participant starts the process with the label $p0$. The label represents a conditional action and asks the participant that executes $p0$ to be in the "working" state. $self$ represents the process identifier that executes the code.

According to Figure 2.1 in Chapter 2, the participants have no choice but to wait for the first action of the coordinator. Suppose the coordinator decides to abort from the beginning of the algorithm. In that case, the participants will have no choice but to abort and execute the action on label $p1$. Suppose the coordinator decides to query the participant for a commit, i.e. $cstate =$ "pre-commit". It that case, each participant can make a non-deterministic choice of either voting *yes* to the commit or *no*. The *no* vote is represented in the label $p1$, and the participant's state changes from "working" to "aborted" and sets the boolean $abortFlag$ to TRUE. This action validates the predicate $atLeastOneAbort$ giving information to the coordinator for an abort decision. Moreover, the participants reach the end of their program when executing $p1$. Conversely, if they decide to accept the commit, characterised by the *yes* vote, their $pState$ value will change from "working" to "prepared".

Once the participant sends its agreement to the commit (updating their state), it executes the label $p2$. This label is a non-deterministic conditional action representing the waiting for the coordinator's decision. If all participants agree to commit, then the predicate $allPCommit$ is satisfied and is set to TRUE. As a result, the coordinator can change its state from "pre-commit" to "commit", representing its decision. Conversely, suppose one of the participants wishes to abandon the commit and decides to abort. In that case, the coordinator's decision will be "abort", and its state will change from "pre-commit" to "abort". Whatever the decision, the participant sends an acknowledgement to the coordinator by changing their state to "committed" if the decision is "commit" and "aborted" if the decision is "abort".

Definition 2 (The participants' PlusCal program)**.**

```
fair process ( Participant ∈ Participants ) {
p0:   await pState[self] = "working";
      either {
p1:           await cState ∈ {"pre-commit", "abort"};
              pState[self] := "aborted";
              abortFlag := TRUE; }
      or {
              await cState = "pre-commit";
              pState[self] := "prepared";
p2:           either {
                      await cState = "commit";
                      pState[self] := "committed"; }
              or {
                      await cState = "abort";
                      goto p1; } ; } ;
    } ;
```

Consequently, the participant has three possible atomic actions: $\{p0, p1, p2\}$. Note that the participant can take the label $p1$ for two reasons. It decides to abort voluntarily at the beginning of the process or because the coordinator has decided to abort.

Remark. $Done$ is a label defined in PlusCal that designates the end of the process. Therefore, it is possible to write **goto** $Done$, which jumps the program at the end of the process. Even if it is

not visible in the processes algorithm, be aware that in addition to the $\{p0, p1, p2, c0, c1, c2\}$ labels, there is $Done$.

2. The PlusCal translation. Once the PlusCal code is written, we can translate it to generate the TLA$^+$ specification. The translation is done through a command line or the TLA$^+$ toolbox. The program looks like the following[7]:

```
MODULE TwoPhaseCommit
EXTENDS Integers, TLAPS
CONSTANT N
Participants ≜ 1 .. N
CoordinatorID ≜ 0
CStates ≜ {"init", "pre-commit", "commit", "abort"}
PStates ≜ {"working", "committed", "aborted", "prepared"}
(* --algorithm TwoPhaseCommit {

fair process (Coordinator = CoordinatorID){ } ;

fair process ( Participant ∈ Participants ) { } ;
 }
*)
  (* BEGIN TRANSLATION *)
Generated TLA+ specification here
  (* END TRANSLATION *)
```

The generated TLA$^+$ specification contains the same variables as the PlusCal code ($cState$, $pState$ and $abortFlag$). An additional variable has been created during the translation; pc, the program control variable that tracks which label a process is currently on. All variables are gathered in a tuple $vars$. The translation preserves the **define** statements that include the predicates $allPCommit$ and $atLeastOneAbort$.

```
  (* BEGIN TRANSLATION *)
VARIABLES cState, pState, abortFlag, pc


allPCommit ≜ ∀ p ∈ Participants : pState[p] = "prepared"
atLeastOneAbort ≜ abortFlag = TRUE

vars ≜ ⟨cState, pState, abortFlag, pc⟩
  (* END TRANSLATION *)
```

The translation also generates a definition; $ProcSet$, which is the set of the algorithm's processes. In our example, $ProcSet$ is the union set of $CoordinatorID$ and $Participants$. The set $CoordinatorID$ contains one element (itself), and the set $Participants$ contains N elements:

$$ProcSet \triangleq \{CoordinatorID\} \cup (Participants)$$

As a recall, a TLA$^+$ system is specified as $Spec = Init \wedge \Box[Next]_{vars}$. With $Init$, the initial states; $Next$, the next action to execute and the predicate $Spec$ is the desired protocol behaviour. In the *Two-Phase Commit* example, $Init$ is defined in **Definition** 3.

[7] Please refer to the Appendix A.1 for the complete PlusCal code.

Definition 3 (The initial predicate $Init$)**.**

$$
\begin{aligned}
Init \triangleq\ & \land\, cState = \text{“init”} \\
& \land\, pState = [p \in Participants \mapsto \text{“working”}] \\
& \land\, abortFlag = FALSE \\
& \land\, pc = [self \in ProcSet \mapsto CASE\ self = CoordinatorID \rightarrow \text{"}c0\text{"} \\
& \qquad\qquad \square self \in Participants \rightarrow \text{"}p0\text{"}]
\end{aligned}
$$

TLA$^+$ syntax is based on conjunctions and disjunctions. Typically, a TLA$^+$ expression starts with a conjunction symbol or disjunction symbol. This kind of expression gives a better overview of the hierarchical structure of the logical formula. **Definition** 3 initialises the variables according to the information given by the PlusCal code. The program control initialises the coordinator's state (using the $CASE$ statement, similar to a pattern-matching) to $c0$, which is its first action and all the participants to $p0$. The specification of a TLA$^+$ system is a set of predicates representing a possible action that the system can execute. Each defined label in the PlusCal code refers to an action in TLA$^+$.

The coordinator's actions. The possible actions of the coordinator are $c0$, $c1$ and $c2$. The content of these actions is derived from the PlusCal code. Let us start with the $c0$ action in **Definition** 4.

Definition 4 (The first action of the coordinator)**.**

$$
\begin{aligned}
c0 \triangleq\ & \land\, pc[CoordinatorID] = \text{“c0”} \\
& \land\, cState = \text{“init”} \\
& \land\, \lor\, \land\, pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“c1”}] \\
& \qquad \land \text{ UNCHANGED } cState \\
& \quad \lor\, \land\, cState' = \text{“pre-commit”} \\
& \qquad \land\, pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“c2”}] \\
& \land \text{ UNCHANGED } \langle pState, abortFlag \rangle
\end{aligned}
$$

The action $c0$ can be executed if the program control of the coordinator is in $c0$ (the first statement of the conjunction), and the state of the variable $cState$ must be in "init" (the second conjunction). The third conjunction represents the **either -or** statement. Either the next state of the program control (represented by the prime) is $c1$, and the state of $cState$ does not change, i.e. stays in the "init" state, or the next state of pc is $c2$, and the variable $cState$ changes to "pre-commit". Finally, the last conjunction states that when the action $c0$ is executed, variables $pState$ and $abortFlag$ are unchanged and keep their current states.

Remark. TLA$^+$ introduce the notation: f EXCEPT $![e_1] = e_2$. It means that the resulting function, say f', is equal to the function f except at the point e_1, where its value is replaced with e_2, namely $f'[e_1] = e_2$.

The second possible action of the coordinator is $c1$ defined in **Definition** 5.

Definition 5 (The second action of the coordinator)**.**

$$
\begin{aligned}
c1 \triangleq\ & \land\, pc[CoordinatorID] = \text{“c1”} \\
& \land\, cState' = \text{“abort”} \\
& \land\, pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“Done”}] \\
& \land \text{ UNCHANGED } \langle pState, abortFlag \rangle
\end{aligned}
$$

Action $c1$ represents the abort decision. It can be executed from the previous action, $c0$ if the coordinator aborts voluntarily, and from the following action $c2$ if the predicate $atLeastOneAbort$ is valid. The program control of the coordinator must be equal to $c1$ to execute the action (the first conjunction). The execution sets the next state of the variable $cState$ to "abort" (the second conjunction). The program of the coordinator ends when the next state of its program control is set to "Done" (the third conjunction).
Definition 6 introduces the last possible action of the coordinator, $c2$.

Definition 6 (The third action of the coordinator)**.**

$$
\begin{array}{l}
c2 \triangleq \ \land\, pc[CoordinatorID] = \text{"c2"} \\
\quad \land\, \lor\, \land\, allPCommit \\
\qquad\quad \land\, cState' = \text{"commit"} \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"Done"}] \\
\qquad \lor\, \land\, atLeastOneAbort \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"c1"}] \\
\qquad\quad \land\, \text{UNCHANGED } cState \\
\quad \land\, \text{UNCHANGED } \langle pState, abortFlag \rangle
\end{array}
$$

This action is the decision step after the query for commit to the participants. This action can be reached only from the action "c0". When $c2$ is executed, either $allPCommit$ is valid (the first disjunction) or $atLeastOneAbort$ is valid (the second disjunction). The former case sets the next state of $cState$ to "commit", and the program of the coordinator ends, and the latter case sets the next state of the program control to "c1". In TLA$^+$, the behaviour of a process is expressed by associating actions with disjunctions or conjunctions. Thus, the behaviour of the coordinator is defined by the expression $Coordinator$, which consists of the disjunction of the three actions presented in **Definition** 4, 5 and 6:

$$Coordinator \triangleq c0 \lor c1 \lor c2$$

If two or more actions are enabled, then the choice is made in a non-deterministic way.

The participants' actions. The process remains the same for the participants, with a few differences. The actions $c0$, $c1$, and $c2$ are actions that only the coordinator can execute. However, we have N participants who can execute a participant's possible actions concurrently. The actions are "p0", "p1", and "p2" and take as argument the participant's identifier executing the action. Thereby it is possible to track each participant's behaviour. By default, $self$ represents the variable that characterises the participant's identifier.
Definition 7 represents the participants' first action expressed by $p0(self)$.

Definition 7 (The first action of the participants)**.**

$$
\begin{array}{l}
p0(self) \triangleq \ \land\, pc[self] = \text{"p0"} \\
\qquad\quad \land\, pState[self] = \text{"working"} \\
\qquad\quad \land\, \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"p1"}] \\
\qquad\qquad\quad \land\, \text{UNCHANGED } pState \\
\qquad\qquad \lor\, \land\, cState = \text{"pre-commit"} \\
\qquad\qquad\quad \land\, pState' = [pState \text{ EXCEPT } ![self] = \text{"prepared"}] \\
\qquad\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"p2"}] \\
\qquad\quad \land\, \text{UNCHANGED } \langle cState, abortFlag \rangle
\end{array}
$$

The action is executed by $self$ and represents the first action that the participant $self$ can execute. The executor of the action must be in its initial state to execute the action (second conjunction). If this condition is satisfied, two possible choices can be executed by the participant $self$ (the two disjunctions). Either abandon the commit and set the next state of the program control to $p1$ (first disjunction) or wait for the coordinator to send the request, characterised by ($cState$ = "pre-commit") and go to the next action, $p2$. Once executed, the action "p0" does not change $cState$ and $abortFlag$ variables.
Definition 8 represents the second action of $self$, which is $p1(self)$.

Definition 8 (The second action of the participants)**.**

$$
\begin{aligned}
p1(self) \triangleq\ & \land\ pc[self] = \text{"p1"} \\
& \land\ cState \in \{\text{"pre-commit"}, \text{"abort"}\} \\
& \land\ pState' = [pState \text{ EXCEPT } ![self] = \text{"aborted"}] \\
& \land\ abortFlag' = \text{TRUE} \\
& \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
& \land \text{UNCHANGED } cState
\end{aligned}
$$

This action can be executed as long as the coordinator aborts or queries for the commit (second conjunction). $p1(self)$ can be reached from $p0(self)$ if the participant decides to abort from the beginning or from the $p2(self)$ action if the coordinator decides to abort. Whatever the case, the action sets the next state of $abortFlag$ to TRUE, the state of the $self$ participant to "aborted" (third conjunction) and the program control to "Done". The execution of this action terminates the participant's program.
Definition 9 represents the third and last action of a participant, which is $p2(self)$.

Definition 9 (The participant third action)**.**

$$
\begin{aligned}
p2(self) \triangleq\ & \land\ pc[self] = \text{"p2"} \\
& \land\ \lor\ \land\ cState = \text{"commit"} \\
& \qquad\ \land\ pState' = [pState \text{ EXCEPT } ![self] = \text{"committed"}] \\
& \qquad\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
& \quad\ \lor\ \land\ cState = \text{"abort"} \\
& \qquad\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{"p1"}] \\
& \qquad\ \land \text{UNCHANGED } pState \\
& \land \text{UNCHANGED } \langle cState, abortFlag \rangle
\end{aligned}
$$

The action $p2(self)$ is reached only from $p0(self)$. If $self$ is executing $p2(self)$, that means that $self$ has voted *yes* for the commit. If the coordinator gives a commit decision, then the next state of $pState$ is "committed", and its program control is set to "Done". On the contrary, an abort decision sets the next state of $self$'s program control to "p1". Consequently, the possible behaviour of a participant $self$ is defined by $Participant(self)$, which consists of the disjunction of the three actions defined in **Definition** 7, 8, and 9:

$$Participant(self) \triangleq p0(self) \lor p1(self) \lor p2(self)$$

We have the set of the coordinator and the participants' possible actions. We can build the $Next$ predicate of the TLA$^+$ specification from these expressions. $Next$ is the disjunction of all actions that we just defined and is represented as follows:

$$Next \triangleq Coordinator \vee (\exists self \in Participants : Participant(self)) \vee Terminating$$

With $Terminating$, the predicate that allows infinite stuttering to prevent deadlock on termination is defined as follows:

$$\begin{aligned} Terminating \triangleq\ & \wedge \forall self \in ProcSet : pc[self] = \text{“Done”} \\ & \wedge \text{UNCHANGED } vars \end{aligned}$$

This first step has designed the specification of the *Two-Phase Commit* algorithm defined by $Spec = Init \wedge \Box[Next]_{vars}$. We can apply formal methods to the specification and verify the model according to defined properties, either with TLC or $TLAPS$.

4.3.4 Methodology of the *Two-Phase Commit* Proof of Correctness

This section describes the *Safety* proof methodology of a *Two-Phase Commit* algorithm by using the proof system of TLA$^+$; $TLAPS$[8]. Accordingly, the module must extend the $TLAPS$ standard module. The methodology consists of three distinct steps that are detailed in the following.

Step 1. The definition of the *Safety* property. A safety property of the *Two-Phase Commit* is that the possible decisions that the coordinator can take are mutually exclusive; i.e. it is not possible to have two participants, one in an aborted state and the other in a committed state. The property formula obtained in TLA$^+$ is defined in **Definition** 10.

Definition 10 (The safety property)**.**

$$\begin{aligned} Safety \triangleq\ & \forall a, b \in Participants : \neg \wedge pState[a] = \text{“aborted”} \\ & \qquad\qquad\qquad\qquad \wedge pState[b] = \text{“committed”} \end{aligned}$$

The property is a conjunction and the symbol ($\neg$) represents the negation in TLA$^+$.

Step 2. The definition of the inductive invariant. The safety property is verified through *the inductive proof of invariants*. The most subtle part of the verification approach is searching for an appropriate inductive invariant that implies the required property and is inductively preserved for all behaviour states. An invariant is sound according to a program if:

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. Safety is valid in all reachable states.

The resulting invariant rule is:

$$\frac{Init \implies Inv \quad Inv \wedge [Next]_{vars} \implies Inv' \quad Inv \implies Safety}{Init \wedge \Box[Next]_{vars} \implies \Box Safety} \tag{4.1}$$

Inductive invariants contain interesting implementation information about the model and represent the overall correctness idea. Thus, Inv must be sufficiently complete to manage the proof of the system.

[8]The deductive proof method for TLA$^+$ is more recent than the TLC model checker. However, $TLAPS$ can be combined with TLC to quickly find minor errors using TLC and prove the system using $TLAPS$.

Type correctness. The characteristic of TLA$^+$ is that it is an untyped language. It is not possible to distinguish an integer from a non-integer expression. The authors of TLA$^+$ assume this choice because an untyped language brings flexibility for writing specifications and does not restrict the expressiveness of a specification like a typed language can do [123]. Checking the type correctness of a specification is not mandatory by the language. However, it is customary to prove that all system variables belong to a set of values throughout all reachable states. Although TLA$^+$ is an untyped language, one can define four basic types in TLA$^+$, namely; number, string, boolean and model value.

Definition 11 defines the type correctness property.

Definition 11 (The type invariant predicate)**.**

$$
\begin{aligned}
TypeOk \triangleq\ & \land\ cState \in CStates \\
& \land\ pState \in [Participants \rightarrow PStates] \\
& \land\ abortFlag \in \text{BOOLEAN} \\
& \land\ pc \in [ProcSet \rightarrow \{\text{“c0”}, \text{“c1”}, \text{“c2”}, \text{“p0”}, \text{“p1”}, \text{“p2”}, \text{“Done”}\}] \\
& \land\ pc[CoordinatorID] \in \{\text{“c0”}, \text{“c1”}, \text{“c2”}, \text{“Done”}\}
\end{aligned}
$$

The property defines the type constraints for all the system's variables. The variables consist of the coordinator's state and participants' state, where $pState$ is a function that maps from the set of participants to the set of participants' possible states. Besides, the type invariant constrains the program control variable to the set of available labels of the system.

The coordinator's correctness. We have chosen to build the invariant according to the coordinator because it has a central role in the system's progression, and it is assumed to be correct. Thus, we build a predicate for each possible action of the coordinator that establishes the state of the system's variables. The resulting invariant is represented by **Definition** 12.

Definition 12 (The coordinator invariant)**.**

$$
\begin{aligned}
IInv \triangleq\ & \land\ pc[CoordinatorID] = \text{“c0”} \implies cInit \\
& \land\ pc[CoordinatorID] = \text{“c1”} \implies Abort \\
& \land\ pc[CoordinatorID] = \text{“c2”} \implies PreCommit \\
& \land\ pc[CoordinatorID] = \text{“Done”} \land cState = \text{“commit”} \implies doneCommit \\
& \land\ pc[CoordinatorID] = \text{“Done”} \land cState = \text{“abort”} \implies doneAbort
\end{aligned}
$$

In the following, we explain each conjunction of the invariant. The invariant is constructed to set the system's state in each possible case of the coordinator's program control. As a reminder, the pc can be $\{c0, c1, c2, Done\}$. It is, therefore, necessary to describe and define an invariant for each case, which provides five predicates. The case of "Done" is split into two different predicates, one for the commit decision and one for the abort.

1. The coordinator is in "c0". The first conjunction $pc[CoordinatorID] = \text{“c0”} \implies cInit$ describes the state of the system when the coordinator is in the initial state. It can be read as "when the program control of the coordinator is in "c0" state that implies that $cInit$ is true". The predicate $cInit$ is defined in **Definition** 13.

Definition 13 (The $cInit$ predicate)**.**

$$cInit \triangleq \forall p \in Participants : \begin{array}{l} \wedge\, cState = \text{“init”} \\ \wedge\, pState[p] = \text{“working”} \\ \wedge\, abortFlag = \text{FALSE} \\ \wedge\, pc[p] \in \{\text{“p0”}, \text{“p1”}\} \end{array}$$

For the predicate $cInit$ to be valid, the system variables must be in their initial state. Note that the program control of the participants can be in $p0$ or $p1$. A participant can be in the initial state, i.e. “p0”, or “p1”, if it decides to abandon the commit before waiting for the coordinator’s decision (without changing its state yet, $pState$ remains unchanged).

2. The coordinator is in “c1”. The second conjunction, $pc[CoordinatorID] = \text{“c1”} \implies Abort$, describes the system’s state where the coordinator decides to abort the system. The predicate $Abort$ is defined in **Definition** 14.

Definition 14 (The $Abort$ predicate)**.**

$$Abort \triangleq \forall p \in Participants : \begin{array}{l} \wedge\, cState \in \{\text{“init”}, \text{“pre-commit”}\} \\ \wedge\, pc[p] \in \{\text{“p0”}, \text{“p1”}, \text{“p2”}, \text{“Done”}\} \\ \wedge\, pc[p] = \text{“Done”} \implies \begin{array}{l} \wedge\, abortFlag = \text{TRUE} \\ \wedge\, pState[p] = \text{“aborted”} \end{array} \\ \wedge\, pc[p] = \text{“p2”} \implies pState[p] = \text{“prepared”} \\ \wedge\, pc[p] \in \{\text{“p0”}, \text{“p1”}\} \implies pState[p] = \text{“working”} \end{array}$$

The coordinator can be in two states because the action “c1” is reached from “c0” and “c2”. If $cState$ is in “init”, then “c1” is executed because the coordinator decided to abort the system at the beginning of the program. If $cState$ is “pre-commit”, the abort wish comes from one of the participants. Besides, if a participant p has finished its program, i.e. $pc[p] = \text{“Done”}$, then p has decided to abort, changing the variable $abortFlag$ to TRUE.

3. The coordinator is in “c2”. The third conjunction, $pc[CoordinatorID] = \text{“c2”} \implies PreCommit$), describes the system’s state where the coordinator has to query the participants to commit and waits for their vote. The predicate $PreCommit$ is defined in **Definition** 15.

Definition 15 (The $PreCommit$ predicate)**.**

$$PreCommit \triangleq \forall p \in Participants : \begin{array}{l} \wedge\, cState = \text{“pre-commit”} \\ \wedge\, pc[p] \in \{\text{“p0”}, \text{“p1”}, \text{“p2”}, \text{“Done”}\} \\ \wedge\, pc[p] = \text{“Done”} \implies \begin{array}{l} \wedge\, abortFlag = \text{TRUE} \\ \wedge\, pState[p] = \text{“aborted”} \end{array} \\ \wedge\, pc[p] = \text{“p2”} \implies pState[p] = \text{“prepared”} \\ \wedge\, pc[p] \in \{\text{“p0”}, \text{“p1”}\} \implies pState[p] = \text{“working”} \end{array}$$

The participants’ behaviour is identical to the $Abort$ predicate. However, in **Definition** 15, the coordinator cannot be in an initial state, which means it cannot spontaneously abort the system. Because the system is concurrent, each participant may have different behaviour from the other.

Their program control may be in all possible states. A pc in "Done" translates the participant's termination with an abort vote and sets $abortFlag$ to TRUE. A pc in "p2" represents a participant willing to commit. The pc in "p0" represents a participant in the initial state, and in "p1", a participant who decides to abandon the commit but has not yet updated its state.

4. The coordinator is in "Done" with a commit decision. The fourth conjunction, $pc[CoordinatorID] = \text{"Done"} \land cState = \text{"commit"} \implies doneCommit$, describes the system's state where the coordinator finishes its program with a commit decision. The predicate $doneCommit$ is defined in **Definition** 16.

Definition 16 (The $doneCommit$ predicate)**.**

$$
\begin{aligned}
doneCommit \triangleq \forall\, p \in Participants : \; & \land\, pc[p] \in \{\text{"p2"}, \text{"Done"}\} \\
& \land\, pState[p] \in \{\text{"prepared"}, \text{"committed"}\} \\
& \land\, pc[p] = \text{"Done"} \implies pState[p] = \text{"committed"} \\
& \land\, pc[p] = \text{"p2"} \implies pState[p] = \text{"prepared"}
\end{aligned}
$$

This system's state is only possible if all participants responded positively to the commit. Participants have no choice but to be in "p2" if they have not yet noted the coordinator's decision or in "Done" if they have taken note of it and finished their program by updating their state to "committed".

5. The coordinator is in "Done" with an abort decision. The fifth conjunction, $pc[CoordinatorID] = \text{"Done"} \land cState = \text{"abort"} \implies doneAbort$, describes the system's state where the coordinator finishies its program with an abort decision. The predicate $doneAbort$ is defined in **Definition** 17.

Definition 17 (The predicate $doneAbort$)**.**

$$
\begin{aligned}
doneAbort \triangleq \forall\, p \in Participants : \; & \land\, pc[p] \in \{\text{"p0"}, \text{"p1"}, \text{"p2"}, \text{"Done"}\} \\
& \land\, pc[p] = \text{"p2"} \implies pState[p] = \text{"prepared"} \\
& \land\, pc[p] = \text{"p1"} \implies pState[p] \in \{\text{"working"}, \text{"prepared"}\} \\
& \land\, pc[p] = \text{"p0"} \implies pState[p] = \text{"working"} \\
& \land\, pc[p] = \text{"Done"} \implies \land\, pState[p] = \text{"aborted"} \\
& \qquad\qquad\qquad\qquad\quad \land\, abortFlag = \text{TRUE}
\end{aligned}
$$

Participants can be in all possible states except "committed". They will eventually all be in an "aborted" state when they reach the end of their program.

Remark. When constructing the invariant with the presented methodology, one has to be careful to treat each coordinator case. Suppose the process that is the subject of the invariant definition ends in "Done" by three possible paths. In that case, its invariant must be described by specifying what happens in each case. If we have three possible paths leading to the program's end, the invariant must deal with the three cases. That is why we have the case where the coordinator finishes with the commit and abort decisions.

The invariant defined in **Definition** 12 turns out to be incomplete when launching $TLAPS$. The proof system will notice a missing case corresponding to the case where the coordinator is in "pre-commit" and "init" when $pc[CoordinatorID] = \text{"Done"}$. Therefore, we must add the missing case to the invariant 12, represented by the sixth conjunction of the invariant. The complete coordinator invariant is defined in **Definition** 18.

Definition 18 (The complete coordinator invariant)**.**

$$\begin{aligned} IInv \triangleq\ & \land pc[CoordinatorID] = \text{“c0”} \implies cInit \\ & \ldots \\ & \land pc[CoordinatorID] = \text{“Done”} \land cState \in \{\text{“pre-commit”}, \text{“init”}\} \implies done \end{aligned}$$

The predicate $done$ is described as follows:

$$done \triangleq \land cState \in \{\text{“commit”}, \text{“abort”}\}$$

This last conjunction and predicate are needed to show that the coordinator can be in the “commit” or the “abort” state only when its program control is in “Done”. Otherwise, the provers will try to prove for the remaining cases, namely “pre-commit” and “init”. In this way, the invariant is provided with the necessary information that the coordinator cannot be in these two states when it reaches the end of its program.
The complete formula of the invariant in TLA$^+$ is the conjunction of *type correctness*, expressed by $TypeOk$, and the *coordinator correctness*, expressed by $IInv$. The resulting invariant is as follows:

$$Inv \triangleq TypeOk \land IInv$$

This **Step 2** ends with constructing the inductive invariant Inv, which is sufficiently complete to give the information about the system’s state to prove the safety property defined in **Step 1**.

Step 3. The proof of the resulting invariant. The two previous steps allowed us to define the formulas and predicates necessary for the proof. This third step aims to prove whether the system satisfies the safety property in **definition** 10, using definitions from 11 to 18. A theorem in $TLAPS$ has the following form:

Theorem 1. Structure of a theorem

THEOREM $name \triangleq$
  ASSUME $assumptions$
  PROVE $goalToProve$

It has a name and a set of assumptions. The property to prove is right after the PROVE keyword. Applying the structure of **Theorem** 1 to the inductive invariant Inv, we obtain **Theorem** 2.

Theorem 2. The invariant theorem

THEOREM $Inv \triangleq$
  ASSUME $Init \implies Inv$,
    $Inv \land [Next]_{vars} \implies Inv'$,
    $Inv \implies Safety$
  PROVE $Init \land \Box[Next]_{vars} \implies Safety$

Each assumption statement refers to a component of the invariant rule defined in formula 4.1. The goal of the theorem is to prove that the system's specification ($Spec$) satisfies the $Safety$ according to the defined assumptions. We decompose the theorem to ease the proof, and each component will represent a theorem to prove in $TLAPS$.

The set theorem. A theorem manipulates definitions and facts from the system. In $TLAPS$, one has to cite the definitions that provers need for the proof explicitly. However, this can quickly clutter the theorem because all the definitions must be cited. Moreover, sending a non-negligible volume of definitions to the prover can complicate the proof. In order to lighten the construction and the information sent to the prover, we define **Theorem** 3. It is used to prevent opening the set definitions in the various theorems. A step in a theorem can cite the $setsTheorem$; thus, avoiding the necessity to recall all the definitions used in the theorem.

Theorem 3. Sets theorem

THEOREM $setsTheorem \triangleq$
$\quad \land CStates = \{$"init", "pre-commit", "commit", "abort"$\}$
$\quad \land PStates = \{$"working", "committed", "aborted", "prepared"$\}$
$\quad \land ProcSet = \{CoordinatorID\} \cup (Participants)$
$\quad \land CoordinatorID \notin Participants$
$\quad \land \forall p \in Participants : p \neq CoordinatorID$
BY DEF $CStates, PStates, ProcSet, CoordinatorID, Participants$

The theorem $setsTheorem$ proves the domain of the sets that constitute the system. In addition, it proves that a participant will never have the same identifier as the coordinator. Therefore, instead of citing the definitions, e.g. $CStates$ and $PStates$, in each proof of each theorem, it is sufficient to cite the $setsTheorem$. $setsTheorem$ is a leaf-proof because it consists of elementary steps, and there is no QED step. To be used, a step must be called after the BY keyword.

According to **Theorem** 2, the first theorem to prove is the initial state invariant defined in **Theorem** 4.

Theorem 4. The theorem $Init \implies Inv$

THEOREM $InitImpliesInv \triangleq$
ASSUME $Init$
PROVE Inv
$\quad \langle 1 \rangle$ USE DEF $Init$
$\quad \langle 1 \rangle 1.\ TypeOk$
$\quad\quad$ BY $setsTheorem$ DEF $TypeOk$
$\quad \langle 1 \rangle 2.\ IInv$
$\quad\quad$ BY $setsTheorem$ DEF $cInit, IInv$
$\quad \langle 1 \rangle 3.$ QED
$\quad\quad$ BY $\langle 1 \rangle 1, \langle 1 \rangle 2$ DEF Inv

The structure of a $TLAPS$ proof is hierarchical. Each step has a level, and the more we divide the proof into sub-proofs, the more the level increases. In the first example, the proof has only one level, characterised by $\langle 1 \rangle$. The theorem assumes that the predicate $Init$ is true and proves Inv. In $TLAPS$, it is possible to extend a definition to all the proof using the keyword USE DEF .

The theorem consists of three sub-proofs; the first is for the $TypeOk$ conjunction of the invariant. Proving the invariant demands citing the $setsTheorem$ theorem, which will be a known fact for the sub-proof. The definition of $TypeOk$ must be cited. The second sub-proof concerns the coordinator invariant. As for the $TypeOk$, the $setsTheorem$ is necessary for the proof. The definitions that must be cited are the invariant itself, $IInv$, and the predicate $cInit$, which represents the system's initial state when the coordinator is in the initial state. Finally, the third sub-proof is the QED step. This step closes the proof and checks if the cited known facts are sufficient to prove the theorem.

Figure 4.7 is a screenshot of the TLA$^+$ toolbox representing the theorem $Init \implies Inv$. The formula turns green when the theorem is proved, as depicted in Figure 4.7a. Conversely, a non-proven step is highlighted in red (see Figure 4.7b). In the latter figure, we have removed from the known facts of the QED the level $\langle 1 \rangle 2$. In that case, the QED step will not be proven, as the cited known facts are not sufficient to prove the theorem. It is common to start by proving the QED step to see if there are any missing steps in the theorem.

```
THEOREM InitImpliesInv ==
ASSUME Init
PROVE Inv
  <1> USE DEF Init
  <1>1. TypeOk
    BY setsTheorem DEF TypeOk
  <1>2. IInv
    BY setsTheorem DEF cInit, IInv
  <1>3. QED
    BY <1>1, <1>2 DEF Inv
```

(a) Proven theorem

```
THEOREM InitImpliesInv ==
ASSUME Init
PROVE Inv
  <1> USE DEF Init
  <1>1. TypeOk
    BY setsTheorem DEF TypeOk
  <1>2. IInv
    BY setsTheorem DEF cInit, IInv
  <1>3. QED
    BY <1>1 DEF Inv
```

(b) Omitting the level $\langle 1 \rangle 2$ from the QED

Figure 4.7 – The theorem $Init \implies Inv$ represented in TLA$^+$ toolbox

The second component of formula 4.1 is $Inv \land Next \implies Inv'$, the invariant for any behaviour state. The formula can be decomposed in $TypeOk \land IInv \land Next \implies Inv'$. Before proving the inductive invariant, let us prove the type correctness $TypeOk$ invariant; $TypeOkInvariant$ (**Theorem** 5) and then we prove the $IInv$ invariant; $InvInvariant$ (**Theorem** 6). The theorem $TypeOkInvariant$ assumes $TypeOk$ and $Next$ and proves the preservation of the $TypeOk$ predicate in any next behaviour. As for the theorem of the initial state, we split the proof according to the $Next$ predicate.

Theorem 5. The type correctness invariant

THEOREM $TypeOkInvariant \triangleq$
ASSUME $TypeOk, Next$
PROVE $TypeOk'$
  $\langle 1 \rangle$ USE DEF $TypeOk$
  $\langle 1 \rangle 1$.CASE $Coordinator$
    BY $\langle 1 \rangle 1$, $setsTheorem$ DEF $c0, c1, c2, Coordinator$
  $\langle 1 \rangle 2$.CASE $\exists self \in Participants : Participant(self)$
    BY $\langle 1 \rangle 2$, $setsTheorem$ DEF $p0, p1, p2, Participant$
  $\langle 1 \rangle 3$.CASE $Terminating$
    BY $\langle 1 \rangle 3$, $setsTheorem$ DEF $Terminating, vars$
  $\langle 1 \rangle 4$. QED
    BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3$ DEF $Next$

The theorem defines four sub-proofs, each for a component of the $Next$, by using the CASE structure. First, we extend the definition of $TypeOk$ to all the proof and avoid the redundancy in

each sub-proof. For the case of the coordinator, the proof needs the $setsTheorem$ and the definition of the coordinator's actions $\{c0, c1, c2\}$ along with the $Coordinator$ definition. We proceed with the same approach for the two remaining cases; $Participant$ and $Terminating$. The theorem ends with the QED step needed to verify the sufficiency of the known facts to prove the theorem.

The theorem $InvInvariant$ defined in **Theorem** 6 assumes the predicate $IInv$, $Next$ and $TypeOk$, and proves Inv'. The theorem defines seven sub-proofs, and each represents a conjunction of the invariant $IInv$. Level ⟨1⟩ expands $IInv$, $TypeOk$ and Inv definitions to the theorem.

Theorem 6. The inductive invariant

```
THEOREM InvInvariant ≜
ASSUME IInv, Next, TypeOk
PROVE Inv'
  ⟨1⟩ USE DEF IInv, TypeOk, Inv
  ⟨1⟩1.CASE pc[CoordinatorID] = "c0"
  ⟨1⟩2.CASE pc[CoordinatorID] = "c1"
  ⟨1⟩3.CASE pc[CoordinatorID] = "c2"
  ⟨1⟩4.CASE pc[CoordinatorID] = "Done" ∧ cState = "commit"
  ⟨1⟩5.CASE pc[CoordinatorID] = "Done" ∧ cState = "abort"
  ⟨1⟩6.CASE pc[CoordinatorID] = "Done" ∧ cState ∈ {"init","pre-commit"}
  ⟨1⟩7. QED
    BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6, setsTheorem DEF Coordinator, Participant
```

If we decompose the theorem's step according to the predicate $Next$, each step is split into sub-proofs of the level ⟨2⟩. The following represents the first two levels of **Theorem** 6.

```
⟨1⟩1.CASE pc[CoordinatorID] = "c0"
  ⟨2⟩1.CASE Coordinator
  ⟨2⟩2.CASE ∃self ∈ Participants : Participant(self)
  ⟨2⟩3.CASE Terminating
  ⟨2⟩4. QED
    BY ⟨2⟩1, ⟨2⟩2, ⟨2⟩3 DEF Next
```

```
⟨1⟩2.CASE pc[CoordinatorID] = "c1"
  ⟨2⟩1.CASE Coordinator
  ⟨2⟩2.CASE ∃self ∈ Participants : Participant(self)
  ⟨2⟩3.CASE Terminating
  ⟨2⟩4. QED
    BY ⟨2⟩1, ⟨2⟩2, ⟨2⟩3 DEF Next
```

We will not detail the proof of all theorem steps as the methodology is the same. We detailed the first case, step ⟨1⟩1. In TLA$^+$, CASE F is equivalent to saying ASSUME F PROVE G with G the current goal of the step. The formula above can be read as "*we assume the case where the coordinator's program control is in* $c0$ *(without executing it), and we prove the four sub-proofs in the lower level of the tree*". Each sub-proof represents a disjunction formula of $Next$. To lighten the proof, we decompose the tree again, according to the $Coordinator$ predicate, and obtain third level steps defined in the following structure (we give the case of the step ⟨2⟩1):

```
⟨1⟩1.CASE pc[CoordinatorID] = "c0"
  ⟨2⟩1.CASE Coordinator
    ⟨3⟩ USE DEF Coordinator
    ⟨3⟩1.CASE c0
      BY setsTheorem, ⟨2⟩1, ⟨3⟩1, TypeOkInvariant DEF c0, cInit, PreCommit, Abort
    ⟨3⟩2.CASE c1
      BY setsTheorem, ⟨2⟩1, ⟨3⟩2 DEF c1, doneAbort, Abort
    ⟨3⟩3.CASE c2
      BY setsTheorem, ⟨2⟩1, ⟨3⟩3 DEF c2, PreCommit, doneCommit, allPCommit, Abort
    ⟨3⟩4. QED
      BY ⟨2⟩1, ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, setsTheorem
```

Each sub-proof is independent of the other. From this example, what is given to the back-end provers are four obligations of proof. The first is; ASSUME ⟨1⟩1, ⟨2⟩1 PROVE ⟨3⟩1. The second is; ASSUME ⟨1⟩1, ⟨2⟩1 PROVE ⟨3⟩2. The third is; ASSUME ⟨1⟩1, ⟨2⟩1 PROVE ⟨3⟩3. The fourth is; ASSUME ⟨1⟩1, ⟨2⟩1 PROVE ⟨3⟩4. Each sub-proof represents a disjunction of the predicate *Coordinator*. The step cites the necessary defintions for the proof. For example, CASE "c1" represents the action that allows the coordinator to abort. In order to prove this step, the provers need information from the system about the abort behaviour. Therefore, the necessary definitions to cite are the predicates *doneAbort* and *Abort*. If a step seems too complicated for the provers to prove, we continue the decomposition of the sub-proofs. As a result, the level ⟨3⟩1 can be decomposed into a fourth level of sub-proofs according to the predicate *IInv'*. The result of the split of the step ⟨3⟩1 is as follows:

```
⟨1⟩1.CASE pc[CoordinatorID] = "c0"
 ⟨2⟩1.CASE Coordinator
  ⟨3⟩ USE DEF Coordinator
  ⟨3⟩1.CASE c0
   ⟨4⟩1. (pc[CoordinatorID] = "c0" ⟹ cInit)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0
   ⟨4⟩2. (pc[CoordinatorID] = "c2" ⟹ PreCommit)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0, PreCommit, cInit
   ⟨4⟩3. (pc[CoordinatorID] = "Done" ∧ cState = "commit" ⟹ doneCommit)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0
   ⟨4⟩4. (pc[CoordinatorID] = "Done" ∧ cState = "abort" ⟹ doneAbort)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0
   ⟨4⟩5. (pc[CoordinatorID] = "c1" ⟹ Abort)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0, cInit, Abort
   ⟨4⟩6. (pc[CoordinatorID] = "Done" ∧ cState ∈ {"init", "pre-commit"} ⟹ done)'
     BY setsTheorem, ⟨2⟩1, ⟨3⟩1, ⟨1⟩1 DEF c0
   ⟨4⟩7. QED
     BY ⟨4⟩1, ⟨4⟩2, ⟨4⟩3, ⟨4⟩4, ⟨4⟩5, ⟨4⟩6, TypeOkInvariant
```

Remark. In a *TLAPS* proof, one has to be careful about opening the definitions. Using USE DEF at the beginning of the proof extends the definitions to the entire proof. However, too much information may be given to the provers. In that case, the QED may struggle to prove the proof since it has to manage too many unnecessary definitions.

The third component of formula 4.1 is $Inv \implies Safety$. The theorem, defined in **Theorem** 7, assumes the predicate *Inv* and proves the property *Safety*. The theorem has seven sub-proofs of level ⟨1⟩. Each step represents a conjunction of the predicate *IInv*, with the last step the QED step.

Theorem 7. The theorem $Inv \implies Safety$

THEOREM $InvImpliesSafety \triangleq$
  ASSUME Inv
  PROVE $Safety$
⟨1⟩ USE DEF $IInv, Inv, Safety$
⟨1⟩1.CASE $pc[CoordinatorID] =$ "c0"
 BY ⟨1⟩1, $setsTheorem$ DEF $cInit$
⟨1⟩2.CASE $pc[CoordinatorID] =$ "c2"
 BY ⟨1⟩2, $setsTheorem$ DEF $PreCommit$
⟨1⟩3.CASE $pc[CoordinatorID] =$ "c1"
 BY ⟨1⟩3, $setsTheorem$ DEF $Abort$
⟨1⟩4.CASE $pc[CoordinatorID] =$ "Done" $\wedge$ $cState =$ "commit"
 BY ⟨1⟩4, $setsTheorem$ DEF $doneCommit, allPCommit$
⟨1⟩5.CASE $pc[CoordinatorID] =$ "Done" $\wedge$ $cState =$ "abort"
 BY ⟨1⟩5, $setsTheorem$ DEF $doneAbort, atLeastOneAbort$
⟨1⟩6.CASE $pc[CoordinatorID] =$ "Done" $\wedge$ $cState \in$ {"pre-commit", "init"}
 BY ⟨1⟩6, $setsTheorem$ DEF $done$
⟨1⟩7. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6, $setsTheorem$ DEF $TypeOk$

The invariant Inv is complete enough for **Theorem** 7 to be proved. The proven theorems 4, 6 and 7 can be used to prove the fourth and last component of the formula 4.1. The theorem, defined in **Theorem** 8, has four steps, and each represents a component of the formula 4.1.

Theorem 8. The theorem $Init \wedge \Box[Next]_{vars} \implies Safety$

THEOREM $InductiveInvariant \triangleq Spec \implies \Box Safety$
⟨1⟩1. $Init \implies Inv$
  BY $InitImpliesInv$ DEF Inv
⟨1⟩2. $Inv \wedge [Next]_{vars} \implies Inv'$
  BY $InvInvariant$ DEF $Inv, vars, TypeOk, IInv, Abort, doneAbort, doneCommit,$
  $PreCommit, cInit, done$
⟨1⟩3. $Inv \implies Safety$
  BY $InvImpliesSafety$ DEF $Safety, Inv$
⟨1⟩4. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, PTL DEF $Spec$

The step ⟨1⟩1 is proven using the fact $InitImpliesInv$ (**Theorem** 4), the step ⟨1⟩2 is proven using the fact $InvInvariant$ (**Theorem** 6), and the step ⟨1⟩3 is proven using the fact $InvImpliesSafety$ (**Theorem** 7). The QED step cites PTL, for Propositional Temporal Logic, from the $TLAPS$ standard module because the theorem uses temporal symbols (□). The complete proof of the *Two-Phase Commit* algorithm is in the following GitHub link [9].

4.4 Conclusion

This chapter provides the reader with the technical background necessary to understand the thesis. This chapter does not entirely describe the logic and the two tools presented, namely *Why3* and

[9] https://github.com/ZeinabYeong/2PC-Proof/blob/master/TwoPhaseCommit.tla

TLA$^+$. However, we hope that the notions will be clear and sufficient to make this thesis as self-contained as possible.
In addition, the methodology applied to the *Two-Phase Commit* example will be helpful in Chapters 6, 7 and 8, where we will repeat the three proof steps using $TLAPS$ on a second algorithm of the same structure.

Part III

A Formal Language for Writing Smart Contracts

Chapter 5

Using Deductive Verification on Smart Contracts

" Every great advance in science has issued from a new audacity of imagination. "

– John Dewey

Contents

A bug or error is a common problem that any software or computer program may encounter. It can occur from a poorly written program, a typing error or bad memory management. However, errors can become a significant issue if the unsafe program is used for critical systems. Therefore, formal methods for these kinds of systems are significantly required.
This chapter proposes a language dedicated to deductive verification, WhyML, as a new language for writing formal and verified smart contracts. The purpose is to avoid attacks exploiting such contract execution vulnerabilities. Because they manipulate cryptocurrency and transaction information, serious consequences can happen if a bug occurs in such programs, such as loss of money. This chapter shows that a language dedicated to deductive verification like WhyML can be suitable for writing correct and proven contracts.
The methodology, introduced in Section 5.1, is first to write a WhyML smart contracts program; then, formulate specifications to be proved as functional properties and the absence of RunTime Errors (RTE). Next, we verify the program's behaviour using the Why3 tool. Finally, we compile the WhyML contracts to the well-known Ethereum Virtual Machine (EVM) (Section 5.3). Moreover, we provide a set of generic mathematical statements that verify functional properties suited to any smart contracts holding cryptocurrency, showing that WhyML can be a suitable language for writing smart contracts. We describe a real industrial use case to illustrate the methodology approach in Section 5.2.

5.1 A New Approach to Writing and Verifying Smart Contracts Using *Why3*

This section shows the expressiveness of the *WhyML* language [83] that allows for writing smart contracts in a formal and verified way. To highlight the advantages of *WhyML*, we choose to compare this language with the most widespread smart contract language, *Solidity* [78].
We focus on *Solidity* because it is the most well-known and used language for smart contracts; thus, drawing the parallel between the *Solidity* contracts and the *WhyML* contracts seems relevant to study. Moreover, through this section, we want to highlight that using the *WhyML* language instead of *Solidity* is to be considered because *Solidity* changes very frequently. As a result, *Solidity* contracts face several attacks, and their semantics is not clear enough to directly apply proving methods to the source code. We believe *WhyML* could be a language for writing smart contracts while proving their correctness and absence of bugs.

5.1.1 *Solidity*

Solidity is a high-level, object-oriented language for implementing smart contracts. This language is designed to target the Ethereum Virtual Machine (EVM). Listing 5.1 gives a simple *Solidity* contract example, "`contract Recording`". The function consists of a variable "`owner`" and a mapping "`dataBalance`" that maps an address "`address`" with an unsigned integer "`uint`". Moreover, the contract defines a "`modifier`" function, "`onlyOwner`", and a function "`recordData`".

The `modifier` primitive in line 5 is a primitive feature in the *Solidity* language. Its role is to restrict access to a function and can be used to model the states and guard against incorrect usage of the contract. An exception is thrown if a function does not meet the `modifier` condition. The example of `onlyOwner` says that the modifier limits the access of a function to the `owner` user. Moreover, *Solidity* defines primitive variables intended to be assigned to information included in a transaction. For example, `msg.sender` is for the sender address; for a transaction that characterises a function call, `msg.sender` will be the address of the function caller, `msg.value` is for the amount of transferred *ethers* (Ethereum's cryptocurrency), and `msg.data` is for storing the function arguments data (acting as a memory).
Thereby, `recordData` is restricted to being executed by the owner address (i.e. `msg.sender` – the calling contract); otherwise, a `throw` is raised (line 6), representing an exception. The `recordData` function assigns an unsigned integer (`_amount`) to an address (`_usersID`) each time `owner` calls

```
1  contract Recording {
2    address owner;
3    mapping (address => uint) public dataBalance;
4
5    modifier onlyOwner () {
6        if (msg.sender != owner){throw;}
7             _;}
8
9    function recordData(address _userID, uint _amount) onlyOwner returns (bool) {
10       if (_userID == address(0x0)) return false;
11       if (_amount == 0) return false;
12       dataBalance[_userID] = _amount;
13       return true;}}
```

Listing 5.1 – A *Solidity* contract example

Level	Cause of vulnerability
Solidity language	Call to the unknown Gasless send Exception disorders Reentrancy

Table 5.1 – An extract from [24] on taxonomy of vulnerabilities in Ethereum contracts

it. The function returns `false` when the user address is invalid (line 10) or there is no amount to record (line 11); otherwise, it returns `true`.

Since its creation in 2014 [187], the *Solidity* language has provided a suitable area for malicious users desiring to take advantage of vulnerabilities that smart contracts encounter [24, 133], e.g. to earn an amount of money through a flaw found in a contract. Hence, the language has constantly evolved and developed to create a secure environment against potential attacks. *Atzei et al.* in [24] have established a summary of common programming pitfalls and identified two major causes of vulnerabilities: problems in the *Solidity* language and poor documentation of weaknesses. Moreover, they give a taxonomy of vulnerabilities in Ethereum smart contracts. We are interested in the vulnerability category related to the *Solidity* language presented in [24]. Table 5.1 summarises some causes of vulnerabilities that we are interested in this study. As explained before, a contract can invoke a function of another contract and send money to a user or another contract using primitive functions such as `send()` and `call()`, or by direct call (i.e. as a traditional function call of an imperative language). However, the primitives can be a source of bugs, and in the following, we explain how these bugs can occur:

- *Call to the unknown*: if a function that does not exist is called, the EVM will still try to execute it. What happens is that the non-existing function signature will not match any of the available signatures in a *Solidity* contract, thus, triggering the *fallback* function. A fallback is a function without a signature (no name, no parameters); `function() { x = 1;}` is an example. Some *Solidity* primitive functions, such as `send()`, always trigger the fallback function of the target address if it exists. The role of `send()` function is to send *ethers*; therefore, if a malicious user calls this function using a contract address instead of a user address, and its fallback function implements an infinite loop, the user can block the process.

- *Gasless send*: the *gas* is a unit that measures the amount of computational effort it will take to execute certain operations in Ethereum. It is a source of bugs that can lead us to a phenomenon of running out-of-gas when a function consumes more *gas* than it should. This exception can occur if we transfer *ethers* to a contract address using the `send()` function. As a result, the fallback function will be triggered, and extra *gas* consumption can occur depending on the execution code of the fallback function.

- *Exception disorders*: *Solidity* uses state-reverting exceptions to handle errors. Such an exception (e.g. not enough *gas*) will undo all changes made to the state in the callee contract and warn the caller by returning false if an error occurs. However, if the call is made via the `send()` instruction, the caller contract should explicitly check the return value to verify that the call has been executed correctly. Furthermore, if a chain of nested calls is made, the exception in the callee contract may or may not be propagated to the caller. This inconsistent exception propagation leads to many cases where exceptions are not handled correctly.

- *Reentrancy*: reentrancy occurs when a function can be called repeatedly before the first invocation of the function is finished. It is also a consequence of the *Call to the unknown* vulnerability. For example, if a function calls itself, we have a recursive call; hence the function is called an n-th time before the (n-1)-th invocation is completed. This scenario causes a reentrancy phenomenon. This vulnerability can cause repeated withdrawals of the balance, which was the source of a previously mentioned attack, "*the DAO*" hack [24].

5.1.2 A Library Model for Encoding *Solidity* Primitives into the *WhyML* language

Previously, we presented the reasons that led us to consider *Solidity* as a language that is not safe enough for critical applications. In this section, we show how, based on the principles of the *Solidity* language, we have designed an approach to write smart contracts using the *WhyML* language. To fully understand this section, we invite the reader to refer to Section 4.2 for notions about the *WhyML* language.

Solidity is an imperative object-oriented programming language characterised by static typing. It provides several elementary types that can be combined to form complex types such as booleans, signed, unsigned, fixed-width integers, settings, and domain-specific types like addresses. Moreover, the address type has primitive functions able to transfer *ethers* (`send()`, `transfer()`) or manipulate cryptocurrency balances (`.balance`). *Solidity* contains elements that are not part of the *WhyML* language. One could model these as additional types or primitive features. Let us take a simple example to illustrate how *WhyML* could model *Solidity* primitives. The following *Solidity* function transfers an integer `amount` from the address `owner`, defined in Listing 5.1 (line 2), to the address `x`:

```
1 function _transfer (address x, int amount) onlyOwner {
2     if (owner.balance >= amount) x.send(amount); }
```

It is necessary to perform some type encoding to express the above example in *WhyML*. Indeed, types like `uint256` [1] and `address` do not exist in *WhyML*; thus, we must introduce them as new *WhyML* abstract types: `type uint256` and `type address`.

Machine integers model. We define several machine integers types, and for each one created, we define a corresponding module. For example, the type `Int160`, `Uint160` (which characterises type `int` and `uint` in *Solidity*) are defined, respectively, in the module "`module Int160`" and "`module Uint`". The latter module defines the type `uint160` and bounded it by setting it to a range of values: `type uint160 = < range 0 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF >`
Then, it defines the unsigned integer type, which is equal to: `type uint = uint160`. However, specifications in *WhyML* use only mathematical integers, e.g. `int` type; hence, we cannot introduce partial functions in the logic, such as an `uint160` addition or subtraction. If an addition appears within a specification, it should be the usual addition over mathematical integers. Thus *WhyML* implicitly maps values of type `uint160` to the corresponding value in type `int`. We introduce a logical function, `to_int`, that maps a value of type `uint160` to its corresponding value in type `int`:

```
function to_int (x : uint160) : int = uint160'int x
```

[1] Integers in *Solidity* are of various sizes (from `uint8` to `uint256`)

In the module `Uint`, we use the cloning to instantiate a module defined in the *Why3* library named "`module Unsigned`" which itself clones the module `Bounded_int`. The instruction `clone export mach.int.Unsigned with` in module `Uint` allow to instantiate the abstract type `t` to `uint160` and set the maximum value `max` to `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF >`.

Applying the same approach, we have defined abstract types for representing integers `int160` and `uint256`. The type `uint256` is used to represent the value of exchanged *ethers* and we define its range of values as:

```
type uint256 = <range 0 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF>
```

The module `Uint256`, defined in Listing 5.2, clones the `Unsigned` module (line 8) and, as for `Uint160`, it instantiates the abstract type `t` to `uint256` and `max` to its maximum value (lines 9 and 10).

```
1  module UInt256
2    use int.Int
3    use Uint
4
5    type uint256
6    constant max_uint256 : int = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
7
8    clone export mach.int.Unsigned with
9      type t = uint256,
10     constant max = max_uint256
11
12   val v_of_uint (n : uint) : uint256
13     ensures {to_int result = Uint.to_int n}
14
15   val v_to_uint (n : uint256) : uint
16     ensures {Uint.to_int result = to_int n}
17
18 end
```

Listing 5.2 – The module `Uint256`

We define two `val` functions to map a `uint` value to a `uint256` value (line 12) and conversely (line 15). Note that in the two `val` functions, we use the function `to_int` instantiated by the module `Unsigned` and the module `Uint256`. To avoid confusion and distinguish them, we add a prefix `Uint` to the function `to_int` from the module `Unsigned`. This way, we indicate from which module the function comes.

Address and hashtable model. Based on the same reasoning, we also model the `address` type and its members as the `msg.sender` primitive and the `send()` function. We choose to encode the private storage (`balance`) by an interface close to a hashtable having as a key-value an address and the associated value a `uint256` value.
The module that mimicks hashtables is "`module Hashtbl`". It is defined generically, and it defines the following abstract types:

```
1 type key
2 type t α = abstract { mutable contents: map key α;
3                       mutable defined: S.fset key;}
```

Line 2 and 3 say that the content of the type can be modified (`mutable`), and each key is mapped to a set of values. Moreover, the module defines functions to `create` and `clear` a hashtable and to `add`, `find` and `remove` an element from a hashtable.

As a result, the module `Address` clone the `Hashtbl` module, named `Bal`, to define its private storage `balance`:

```
1 clone import Hashtbl as Bal with type key = address
2 val balance : Bal.t uint256
```

The current value of the balance of addresses is `balance[address]`. In addition, the `send` member is translated by a `val` function, which performs operations on the `balance` hashtable. In *Solidity*, the `send()` function can fail (return `False`) due to an out-of-gas, e.g. an overrun of 2300 units of

gas, because in some cases, the transfer of *ethers* to a contract involves the execution of the contract fallback; therefore, the function might consume more *gas* than expected (more than 2300).

In *WhyML*, we chose to modify the *Solidity* `send()` function. The *WhyML* `send()` function, Listing 5.3, does not allow fallback execution; it only transfers *ethers* from one address to another. Thus, the *WhyML* `send()` function does not return a boolean because we assume that the transfer must occur if the function is executed. It is assumed that the function never fails, but it could have been encoded to accept the failure of the function by introducing a boolean, as is the case for the *Solidity* version. In our case study, we made this choice for the sake of simplification. Knowing that this choice is not satisfactory, in the case of a future study, it would be considered to adapt the `send()` function to manage execution failures.

```
1 let address_send (amount : uint256) (from_ : address) (to_ : address) : unit
2  requires { uniqueAddress from_ to_ }
3  requires { acceptableEtherTransaction balance from_ to_ amount }
4  ensures  { etherTransactionCompletedSuccessfully (old balance) balance from_ to_ amount}
5   =
6     balance[from_] ← balance[from_] - amount;
7     balance[to_] ← balance[to_] + amount
```

Listing 5.3 – *WhyML* send function

The function `address_send` defines a set of specifications with preconditions and postconditions to ensure proper execution. The specification requires to have a different user for the sender (`from_`) and the receiver (`to_`), using the predicate: `predicate uniqueAddress (a: address) (b: address) = a ≠ b` from the module `Address`, and that the transaction must be acceptable (line 3). An acceptable transaction requires that the sender has a sufficient amount of *ethers* and that the transfer does not cause an overflow at the receiver.
Moreover, the `address_send` function ensures a successful transaction in line 4. It ensures that the receiver gets the amount it wants, and the balance between the sender and receiver remains after the function's execution. The formal definition of the two predicates is defined in Section 5.1.4.

Gas model. We give a *gas* model, "`module Gas`", to specify the maximum amount of *gas* needed for each defined function. We define a value "`gas`" to express the amount of *gas* consumed and a value "`alloc`" to express the memory allocation. Both are defined as mutable integers.

```
1 val gas: ref int
2 val alloc: ref int
```

Moreover, we introduce a "`val add_gas`" function (Listing 5.4) that adds to the variable `gas` the amount of *gas* consumed each time the function is called (the input `g`). The same goes for the variable `alloc`:

```
1 val add_gas (g:int) (a:int) : unit
2  requires { 0 ≤ g }
3  requires { 0 ≤ a }
4  ensures { !gas = old !gas + g }
5  ensures { !alloc = old !alloc + a }
6  writes { gas, alloc }
```

Listing 5.4 – `add_gas` function

The function has no body and is defined only by its set of properties (see Table 4.6). The function requires non-negative inputs (`g` and `a`) values and ensures the modification of `gas` and `alloc` variables. The number of allocations is essential because the real *gas* consumption of EVM integrates the maximum quantity of volatile memory used.

5.1.3 Functions in Smart Contracts

Oracles. Often, developing smart contracts relies on the concept of *Oracles* [4]. An oracle can be seen as the link between the blockchain and the "real world". In the context of blockchain, the real

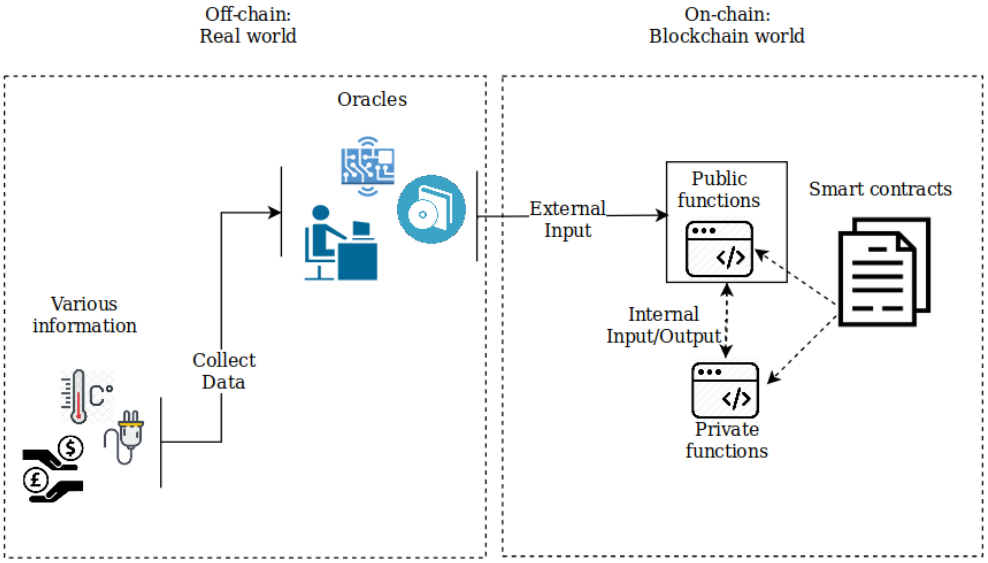


Figure 5.1 – Data routing process between on-chain and off-chain

world is everything outside the blockchain, also called *off-chain*. On the other hand, the blockchain world is everything that comes under the blockchain, i.e. *on-chain*.
Some smart contract functions have arguments that are external to the blockchain. However, the blockchain does not have access to information from an untrusted off-chain data source. Accordingly, the oracle provides a service responsible for entering external data into the blockchain, having the role of a trusted third party. However, questions arise about the reliability of such oracles and the accuracy of information. Oracles can have unpredictable behaviour, e.g. a sensor that measures the temperature might be an oracle but might be faulty; thus, one must account for invalid information from oracles. Figure 5.1 illustrates the three steps to provide various information from outside the blockchain to the blockchain:

Step 1: *Off-chain raw data collection*. The first step is to collect the off-chain data (produced in the real world) used by the blockchain application. The data can be of various kinds, depending on the application. For example, as we will see in more detail in Section 5.2, energy consumption data can be used by the blockchain in the context of a local energy consumption application. The entity that collects this data are oracles, and they can be a person, a sensor or software.

Step 2: *Provide data to the blockchain*. The second step is to provide the blockchain with the collected data that the application requires. This action is applied by the oracles who have access to the off-chain and on-chain world. Providing the data can be done via an interface or a directly linked platform to the smart contracts responsible for receiving this data.

Step 3: *Data processing*. The smart contracts responsible for processing this data have functions designed to have input arguments that accept data from outside the blockchain. Once the data is assigned in the smart contract, it becomes on-chain data, and other smart contracts can use it.

Private and public functions. Considering these different steps of the data routing process and the distinction between the real world and the blockchain world, we defined two types of functions involved in contracts. We noted that some functions are called internally by other smart contracts functions; they are called "*private functions*". In contrast, others are called externally by oracles, and they are called "*public functions*". The proof approach of the two types is different.

For the *private* functions, one defines preconditions and postconditions, and then we prove that no error can occur and that the function behaves as it should. It is thus not necessary to define exceptions to be raised throughout the program; they are proved never to occur.
Conversely, the *public* functions are called by oracles, the behaviour of the function must take into account any input values, and it is not possible to require conditions upstream of the call. In contrast, exceptions are necessary; we use so-called *defensive proof* to protect ourselves from errors that oracles can generate. No constraints are applied to postconditions. However, since specifications in *WhyML* are modular when a function calls another, it must verify and satisfy the preconditions of the callee function before the call.

Let us take an example of a `send_data()` function (see Listing 5.5) that is intended to transfer some data from one address to another, which in its current state can generate data overflow.

```
1 function send_data(address receiver, uint _data) onlyOwner {
2  if (balance[owner] < _data) return;
3   balance[owner] -= _data;
4   balance[receiver] += _data; }
```

Listing 5.5 – Simple *Solidity* function of sending data

The variable `owner` and the modifier `onlyOwner` are those of the contract defined in Listing 5.1. To understand the difference in proof approach according to the type of function, we apply the two proof approaches to the function defined in Listing 5.5. Therefore, if we qualify this function as a private function, we obtain the result in Listing 5.6.

```
1 let private_send_data (data: uint256)(receiver: address): uint
2     requires {data > 0 ∧ balance[owner] ≥ data}
3     requires {msg_sender ≠ receiver ∧ msg_sender == owner}
4     requires {balance[receiver] + data < max_uint256}
5     ensures {balance[msg_sender] = old (balance[msg_sender]) - data}
6     ensures {balance[receiver] = old (balance[receiver]) + data }
7 =
8     balance[owner]← balance[owner] - data;
9     balance[receiver]← balance[receiver] + data;
```

Listing 5.6 – `send_data` private function in *WhyML*

The specification states that we require positive data to send and that the owner owns at least the amount of `data` (line 2). Moreover, the address that will receive the data cannot be the caller of the function, i.e. `msg_sender` (line 3). The third precondition in line 4 states an absence of integer overflow; this part is detailed in Section 5.1.4. We appreciate from the postconditions (lines 5-6) that, at the end of the function execution, data is transferred from one address to another (lines 8-9). A complete set of specifications must provide sufficient information to know the role of a function without even looking at the program. The properties of `private_send_data()` are sufficient to prove the function and interpret it without seeing the program.

The second proof approach is applied to public functions. We take the example of Listing 5.5 and assume this function to be public. The result is shown in Listing 5.7.

```
1  exception InvalidData, InvalidAddress, OnlyOwner, NoData, Overflow
2
3  let public_send_data (data: uint256)(receiver: address): uint
4      raises {InvalidData → data = 0}
5      raises {InvalidAddress → msg_sender == receiver}
6      raises {OnlyOwner → msg_sender != owner}
7      raises {NoData → balance[owner] < data}
8      raises {Overflow → balance[receiver] + data > max_uint256}
9      ensures {balance[msg_sender] = old (balance[msg_sender]) - data}
10     ensures {balance[receiver] = old (balance[receiver]) + data }
11 =
12     if msg_sender != owner then raise OnlyOwner;
13     if data = 0 then raise InvalidData;
14     if msg_sender == receiver then raise InvalidAddress;
15     if balance[owner] < data then raise NoData;
16     if balance[receiver] > max_uint256 - data then raise Overflow;
17       balance[owner]← balance[owner] - data;
```

```
18      balance[receiver]← balance[receiver] + data;
```

Listing 5.7 – `send_data` public function in *WhyML*

Following the modelling rules, we define exceptions (lines 1 and 4-8) to be raised and no defined preconditions. What we have as requirements in Listing 5.6, we have as exceptions in the public case. We raise an exception in the two cases where we have no data to send (lines 4, 13) and if the receiver is also the sender (lines 5, 14). In addition, the `modifier` primitive can be modelled in two ways in our approach. In Listing 5.6, it is modelled as a precondition (line 3), and in Listing 5.7, it is modelled as an exception to handle (lines 6, 12). Lines (8-16) refer to the integer overflow property discussed in the following.

5.1.4 Functions Properties in a Smart Contract

In a *WhyML* contract, we distinguish two types of properties to be proven: the absence of *runtime errors* and *functional properties*.

Runtime Errors (RTE). RTE are an annoying and frustrating experience for users. RTE are errors that are only detected when running the program. The cost of such bugs can be very high, and many methods have been proposed to reduce these failures [132, 135]. There are so many runtime errors that they are not always easy to diagnose. We focus on the principal RTE, namely:

1. *Positive values.* The parameters of a function must be valid; therefore, the parameters that express a quantity (*gas*, *ethers*, data, or other) must be positive (in some instances positive or null), and those that represent recorders such as arrays or hashtables must not be empty. Most of the exchanged data express an amount of cryptocurrency; hence, a transferred negative amount can cause damage, i.e. the receiver could lose money instead of receiving. To counter this error, we have encoded the values of *ethers* and *gas* as an unsigned integer (see Section 5.1.2).

2. *Integer overflow and underflow.* In many situations, performing proof of the absence of integer overflow/underflow is extremely difficult and invasive. In our programs, we manipulate machine integers; thus, we fix a maximum and a minimum bound. We must show the absence of arithmetic overflow/underflow by defining bounding preconditions. Hence, such bounds invade specifications throughout the program, resulting in an impractical annotation/proof burden. When manipulating counters, we use the `Peano` number from [59] to avoid such proof.

3. *Index out of array bounds.* If the program manipulates an array, all the requests must use a non-negative index and an index less than the size of the array element. Defining an invariant is enough to prove that the index will never be more than the size of the array. It is also possible to define a precondition to prove the absence of such RTE.

4. *Division by zero.* In computing, a program error may result from an attempt to divide by zero, which may generate positive or negative infinity, generate an exception, generate an error message, or cause the program to terminate.

Recall that properties that express the absence of RTE defined in a function f must be extended and proved to functions that call f to satisfy their preconditions.

Functional properties. Proving that a program does not cause RTE does not mean it behaves as it should. We also want to prove its correctness and termination. A correct program precisely does what its designers and users intend. Furthermore, a formally correct program is one whose correctness can be proved mathematically by specifying what the program is intended to do for all possible values of its input. We provide mathematical formulas that can be used to express *predicates*,

axioms, and *invariants* to define functional properties. For each mathematical formula useful to the proof, we refer to the vulnerabilities stated in Section 5.1.1, resulting from the paper [24].

- *An acceptable transfer*. Sending a quantity of *ethers* via the primitive `send()` or `transfer()` of *Solidity* smart contracts can cause various vulnerabilities. Hence, securing the process of such primitives is fundamental. Therefore, we introduce in *WhyML* contracts a predicate that states an acceptable transaction that must be satisfied when used. This predicate has been introduced previously with the *WhyML* send function 5.3. The predicate `acceptableEtherTransaction` must be satisfied to execute the `address_send` function. The predicate takes four arguments: a hashtable, two addresses and some *ethers* amount. The hashtable of balances `h` is a hashtable that records the balance of users (identified by their address). The two addresses are those of the sender and receiver.

```
1 predicate acceptableEtherTransaction (h: t uint256) (sender : address)(receiver: address) (amount
      : uint256)
2   = h[sender] ≥ amount ∧ h[receiver] + amount ≤ max_uint256
```

  This predicate can be used in a precondition specification of a function that performs a transfer of *ethers*, such as the `send()` primitive of an address. Thus, a user cannot send an amount of *ethers* that it does not have. Note that the predicate also defines the RTE property "`h[receiver] + amount <= max_uint256`". In addition, the *WhyML* `address_send` function does not implement a fallback function, which allows us to avoid *Call to the unknown* and *Reentrancy* vulnerabilities from Table 5.1.

- *A successfully completed transaction.* When executing a function that is supposed to transfer *ethers* or any other data, what is hoped for is that once the function is executed, we are sure that the transfer is successful. The predicate *etherTransactionCompletedSuccessfully* models this expectation to ensure that a transaction is successfully executed.

```
1 predicate etherTransactionCompletedSuccessfully (hBefore : t uint256) (hAfter : t uint256) (
      sender, receiver: address)  (data: uint256)
2   = hBefore[sender] + hBefore[receiver] = hAfter[sender] + hAfter[receiver] ∧ hAfter[sender] =
      hBefore - data ∧ hAfter[receiver] = hBefore[receiver] + data
```

  A transfer completes if the sum of the sender and the receiver balance before and after does not change. That is, there is no loss of money during the sending. Moreover, the receiver receives the quantity of `data` it wants. This predicate avoids the occurrence of *Exception disorder* and *Reentrancy* errors (see Table 5.1).

- *Duplicate recording.* Several defined functions in a contract record data, typically in arrays or hashtables. However, duplicate recording can cause errors in memory management. A `mem` function allows checking a hashtable to determine if a value is already present or not before the execution of the function. `mem` returns `True` if it is the case; otherwise, the function returns `False`:

```
1     requires {¬ mem nameOfHashtable keyValue}
```

  This type of error can cause double expenses if a transfer is recorded twice in the blockchain.

- *Ensuring a maximum consumption of gas*. The vulnerability *Gasless send* can be avoided by the *gas* model. Instructions in *Solidity* consume an amount of *gas*, and they are categorised by level of difficulty; e.g. for the set $W_{verylow} = \{ADD,\ SUB,\ ...\}$, the amount to pay is $G_{verylow} = 3\ units\ of\ gas$, and for a create operation, the amount to pay is $G_{create} = 32000\ units\ of\ gas$ [187]. The price of an operation is proportional to its difficulty. Accordingly, we fix for each *WhyML* function the appropriate amount of *gas* needed to execute it. Thus, at the end of the function instructions, a variable `gas` expresses the total quantity of *gas* consumed during the process. We give more details about the *gas* calculation in Section 5.3.

This first section introduces *WhyML* as a language for writing smart contracts. We elaborate a parallel between the two languages, *Solidity* and *WhyML*, to show interest in using *WhyML* to write contracts in a formal and proven way. We have introduced the encodings made to express values like *ethers* and *gas* and two approaches of proofs according to the type of functions of the contracts.

5.2 Use Case: The BEMP Decentralised Application

This section demonstrates how to apply *WhyML*, as a smart contract writing language, in a case study, "the Blockchain Energy Market Place application" (BEMP) [148]. This blockchain application based on smart contracts, popularised by the Brooklyn microgrid [139], aims at managing peer-to-peer energy exchanges between prosumers (producer and consumer) in a microgrid.

5.2.1 Description of BEMP

Figure 5.2 depicts an energy trade between two users, Alice and Bob, of the BEMP application. In this figure, transfers 1 and 1' are performed continuously and independently of the market, whereas transfers 2, 2' and 3 are performed regularly by smart contracts according to transfers 1 and 1' (users' smart meters are the oracles that feed the BEMP with energy production and consumption data). The figure illustrates:

- (1) Alice produces energy and supplies her excess production to the grid.
- (1') Bob consumes a certain amount of energy pulling from the grid.
- (2) Alice's production is capitalised as "crypto-kilowatt-hours" (crypto-kWh), and her smart meter provides to the BEMP her production data.
- (2') Bob's energy consumption data are collected by his smart meter and sent to the BEMP application.
- (3) Bob pays Alice in *ethers* for his energy consumption.

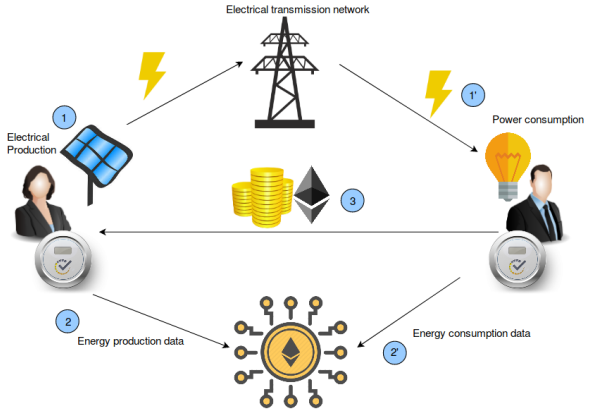


Figure 5.2 – The BEMP Process

The BEMP application allows the payment to be regulated according to consumption. It is a decentralised application, and each user wishing to participate in the market will need to download the application and create an account. The considered application implementation is deployed in the Ethereum blockchain and composed of the following smart contracts coded into *Solidity* language:

- ***Account*** contract. The contract stores the properties of an account (user location, user market and purchasing capacity) and performs the *ethers* payment. One such instance is created for each user.

- ***Accounts*** contract. It stores a mapping (kind of hashtables) of the application user's account and defines a function that records the account into that mapping.
- ***Market*** contract. It receives all sales and orders and ensures the transit of crypto-kWh.
- ***Algorithm*** contract. It determines the best way to satisfy sales and orders.
- ***Registry*** contract. It logs any events occurring in BEMP (account creation, sales and orders, actual transactions).

Once deployed in an Ethereum blockchain, the calling of these contracts' functions is performed by a JavaScript oracle running regularly. In practice, the oracle is connected to the blockchain as a casual client and regularly sends the same sequence of transactions to smart contracts.
The process begins with the market opening via a transaction sent to the *Market* contract. Then, the application records each user's production and consumption amounts. The record is performed with a sent transaction to the *Registry* contract, which triggers three automatic transactions between the contracts *Registry*, *Account* and *Market*. Once all records have been done, the application's algorithm runs. A transaction is sent to the *Algorithm* contract, which triggers four internal transactions between the contracts *Algorithm*, *Registry*, and two instances of *Account* and *Market* contract. A transaction is sent to the *Market* contract to close the market when all the transfers have been completed.

5.2.2 BEMP in *WhyML*

This part shows how our proof methodology described in the previous section can be applied to a real case. In the initial work, we applied our method to a simplified version of the application: a one-to-one exchange (one producer and one consumer, as depicted in Figure 5.2). This first test allowed us to identify and prove RTE such as *overflow* or *index out of array bounds*. The simplicity of the unidirectional exchange model did not allow the definition of complex functional properties to show the importance and utility of the *Why3* tool. Plus, this number of users does not express a real application implementation.

An energy trade application is a suitable case study to show the potential of the formal deductive verification provided by *Why3*. Moreover, in addition to transferring *ethers*, users transfer crypto-kWh to reward consumers for consuming locally produced energy. Hence, the system needs to formulate and prove predicates and functional properties of functions handling various data other than cryptocurrency.
Our approach was first to define which smart contract's function was of the public and private type. Secondly, to express in *WhyML* the functions defined in the various *Solidity* contracts. Then, to establish the specification for each function of the contracts based on the proof rules according to the function types. Therefore, predicates such as `acceptableAmountTransaction` and `amountTransactionCompletedSuccessfully` must be added. The BEMP consists of five different contracts; some only communicate internally, e.g. contract *Account* allowing to create an account, while others only serve to communicate with oracles; e.g. contract *Accounts* allowing to register an account.

Examples of private function from the BEMP application. In this part, we apply the rules of proof and modelling according to the private function type.

```
1 let transferFromMarket (_to : address) (cryptokwh : uint) : bool
2     requires {msg_sender == market ∧ cryptokwh > 0 }
3     requires {acceptableAmountTransaction marketBalanceOf importBalanceOf market _to cryptokwh}
4     ensures {amountTransactionCompletedSuccessfully (old marketBalanceOf) marketBalanceOf (old
       importBalanceOf) importBalanceOf market _to cryptokwh}
5     = (* The program *)
```

Listing 5.8 – Example of a *WhyML* private function from BEMP

Listing 5.8 illustrates a proof example of a private function resulting from the contract *Registery* of the BEMP application. The function transfers a positive amount of `cryptokwh` from the global address market "`market`", defined in the contract, to the `_to` address. This process is internal to the blockchain; there is no external exchange; hence the function is qualified as private. According to the modelling approach, we define complete preconditions and postconditions. The precondition in line 2 expresses the `modifier` primitive discussed in Section 5.1.1; thus, the `tranferFromMarket()` function can only be executed by the market, restricting the function's caller, `msg_sender`, to the market. Note that `marketBalanceOf` is the hashtable that records crypto-kWh balances associated with market addresses, and `importBalanceOf` is the hashtable that records the amount of crypto-kWh intended for the buyer addresses. The second precondition is the condition to guarantee an acceptable transaction; thus, satisfying the following predicate:

```
1 predicate acceptableAmountTransaction (h1 h2: t uint) (sender receiver : address) (data : uint) =
2     h1[sender] ≥ data ∧ h2[receiver] ≤ max_uint - data
```

Moreover, the function ensures a successful transaction if the following predicate is satisfied:

```
1 predicate amountTransactionCompletedSuccessfully (h1_before : t uint) (h1_after : t uint ) (h2_before :
      t uint) (h2_after : t uint ) (sender receiver : address) (data : uint) =
2     h1_before[sender] + h2_before[receiver] = h1_after[sender] + h2_after[receiver] ∧ h1_after[sender]
      = h1_before[sender] - data ∧ h2_after[receiver] = h2_before[receiver] + data
```

It ensures the conservation of the user's balance during the transaction. From the specification set in Listing 5.8, we understand the function's behaviour without referencing the program. Therefore, the function `transferFromMarket` must satisfy the following absence of RTE and functional properties:

- **RTE**: *(1) Positive values*; before executing the function, it is necessary to require (line 2) that there is a valid amount of crypto-kWh to transfer. *(2) Integer overflow*; before the execution of the function, it is necessary to require that no overflow will occur when `_to` receives `cryptokwh` (line 3, see `acceptableAmountTransaction`).

- **Functional properties**: *(1) Acceptable transfer*; before executing the function, it is necessary to require (line 3) that the transfer can be done; hence the market has enough crypto-kWh to send. *(2) Successful transfer*; after executing the function, we ensure (line 4) that the transaction is completed successfully; hence the sum of the two balances (sender + receiver) remains unchanged before and after the execution, and the sender transfers the amount of crypto-kWh intended by the receiver. *(3)* `modifier` *function*; the function can be executed only by the market; thus, we require that the function's caller may be solely the market (line 2).

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

Examples of public function from the BEMP application. Listing 5.9 illustrates a public function from the *Registry* contract of the BEMP application. The function `registerSmartMeters` is identified by a name (`meterID`) and an owner (`ownerAddress`) parameters. Note that all meter owners are recorded in a hashtable `addressOf` associated with a key-value `meterID` of the `string` type. The main bug of the function is to register a meter twice. There are no preconditions following the modelling rules; instead, we define exceptions.

```
1 exception OnlyOwner, ExistingSmartMeter
2
3 let registerSmartMeter (meterID : string) (ownerAddress : address)
4     raises { OnlyOwner→ msg_sender ≠ owner }
5     raises {ExistingSmartMeter → mem addressOf meterID}
6     ensures { (size addressOf) = (size (old addressOf) + 1 ) }
7     ensures { mem addressOf meterID}
8     = (*The program*)
```

Listing 5.9 – Example of a *WhyML* public function from BEMP

`registerSmartMeter` must respect the following absence of RTE and functional properties:

- **RTE**: *Duplicate record*; during the function execution, it is necessary to raise an exception, `ExistingSmartMeter`, if a smart meter and its owner is already recorded (line 1 and 5).
- **Functional properties**: *(1)* `modifier` *function*; the first exception, `OnlyOwner`, is the `modifier` function, which restricts the function execution to the owner – the function's caller (`msg_sender`). The exception is raised when the function's caller is not the owner (lines 1 and 4). *(2) Successful record*; at the end of the function execution, we ensure (line 6) that a record is made. *(3) Existing record*. At the end of the function execution, we ensure (line 7) that the registered smart meter has been recorded correctly in the hashtable `addressOf`.

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

5.2.3 Trading Algorithm in BEMP

In a second step of the use case study, we extended the application to an indefinite number of users and enriched our specifications. The deductive approach is quite suitable for this order of magnitude. The significant aspect that differs between the two versions is consumers' choice between the various offers of sale at their disposal on the market. Indeed, we had only two actors in the first version, so there was no way to choose with whom to make a trade. Alice supplies Bob with electricity, and the price of a kilowatt-hour was fixed in advance. We want to introduce a marketplace environment with sale and purchase offers based on a simple trading algorithm. Therefore, a sixth contract, the *Trading* contract, is added to the five existing contracts to perform the trading function.

Figure 5.3 represents the flowchart of the trading algorithm defined in the *Trading* contract. The algorithm takes as input sell and buy orders arrays. The two arrays are sorted in descending order according to the price. The algorithm goes through the two arrays to match a seller with a buyer and create a list of orders (the output). If the algorithm reaches the end of one of the two arrays, it returns the result of the trading order list. The algorithm starts by taking the first element of each array. The i buyer of the buy order array and the j seller of the sell order array. The algorithm is made so that a seller cannot sell its energy to a buyer willing to pay less than the seller offers. Consequently, if a match is done between a seller and a buyer, we keep the seller's price. As we favour the buyer, the algorithm checks whether the bid to buy the energy of buyer i is higher or equal to that of seller j. Suppose the seller at the top of the table makes a bid too high for i. In that case, we move on to the next seller, return to the beginning of the algorithm, and carry out the same check between buyer i and the price of the seller $j+1$. If, on the contrary, the seller's price is lower than or equal to the buyer's, then we continue.

The third level of decision is whether the seller has enough energy tokens to supply the buyer i. If the seller j does not have the quantity of token requested by the buyer, then the seller sells all of its token (which amounts to setting its token balance to zero). Then, we create a record order with the information necessary for the trading: the seller and the buyer's address, the number of exchanged tokens and the purchase price. That record is added to the order list, the output of the trading algorithm.
Since the current seller j could not provide the entire quantity of token requested by the buyer i, we move on to the next seller to satisfy the requirements of i. That brings us back to the beginning of the loop to repeat the same decision steps as previously explained. On the other hand, if the seller has the number of tokens requested by the buyer, the token balance is updated. The buyer's balance token is reset to zero since the seller has provided the entire quantity of tokens requested by the buyer. Then, we create and add the order record to the order list returned at the end of the algorithm. Once this stage is carried out, one passes to the next buyer $i+1$ and checks if the seller still has tokens to sell. If the seller has exhausted all its tokens, we move on to the next seller;

otherwise, it returns to the loop and returns to the first decision step. This process ends once we reach the end of one of the two arrays, and there is no longer any possibility of making a match between sellers and buyers.

This example of a trading algorithm is straightforward from an order management point of view. However, it is more complex than most smart contract functions and is efficient enough to match a producer (seller) with a consumer (buyer) of our BEMP application. For a first approach to trading, we adapted an order book matching algorithm with the limit orders algorithm to our case study [70].

5.2.4 *Trading* Smart Contract in *WhyML*

The *Trading* contract has the role of matching a buyer with a seller. The function that performs the trading algorithm implements the diagram in Figure 5.3. We assume that all users have smart meters that record their energy consumption and production data. A smart contract is implemented to receive consumption and production data from the real world provided by the smart meters oracles. The smart contract will store and analyse this data to create a consumption and production array. The BEMP application is configured to make these measurements by fixed time intervals to have fixed size consumption and production arrays and not a dynamic one. Indeed, the latter case would generate a more complex algorithm different from the current one. The contract defines two types illustrated in Listing 5.10: `consumption`, which corresponds to the buyer's consumption and `production`, which corresponds to the seller's production. Both types are records that have the following members: the user's address of the type `address`, the smart meter identifier[2], the selling price for a producer `price_s` and the purchase price for a consumer `price_b`, and the amount of energy produced `amount_s` and consumed `amount_b`. We define a `val` function value to represent the array of production `prod_array` and the array of consumption `consum_array`. Once all the smart contracts are implemented in the application deployed on the blockchain, the data `prod_array` and `consum_array` will represent the input data of the *Trading* contract.

```
1  type consumption = {buyer : address;
2                      smb_id: smartMeterID;
3                      price_b: uint;
4                      amount_b: uint}
5
6  type production = {seller : address;
7                     sms_id : smartMeterID;
8                     price_s: uint;
9                     amount_s: uint}
10
11 val consum_array : array consumption
12 val prod_array : array production
```

Listing 5.10 – Consumption and production records encoded in a *WhyML* contract

Implementing the trading algorithm in *WhyML* needs to add some additional types (Listing 5.11). We introduce additional types such as `order` and `order_trading`. The `order` type is a record containing `orderAddress`, a seller or a buyer, `tokens` that express the crypto-Kilowatthours, and `price_order`; the purchase or sale price. The `order_trading` type is a record that contains seller information `seller_index`, buyer information `buyer_index` and the transferred amount `amount_t`. The buyer and seller's information corresponds to their index that places them in their respective order arrays. For example, if the member `seller_index` equals 3, it corresponds to the third seller from the top of the sell order array.

[2] `smartMeterID` is an abstract type created in *WhyML*.

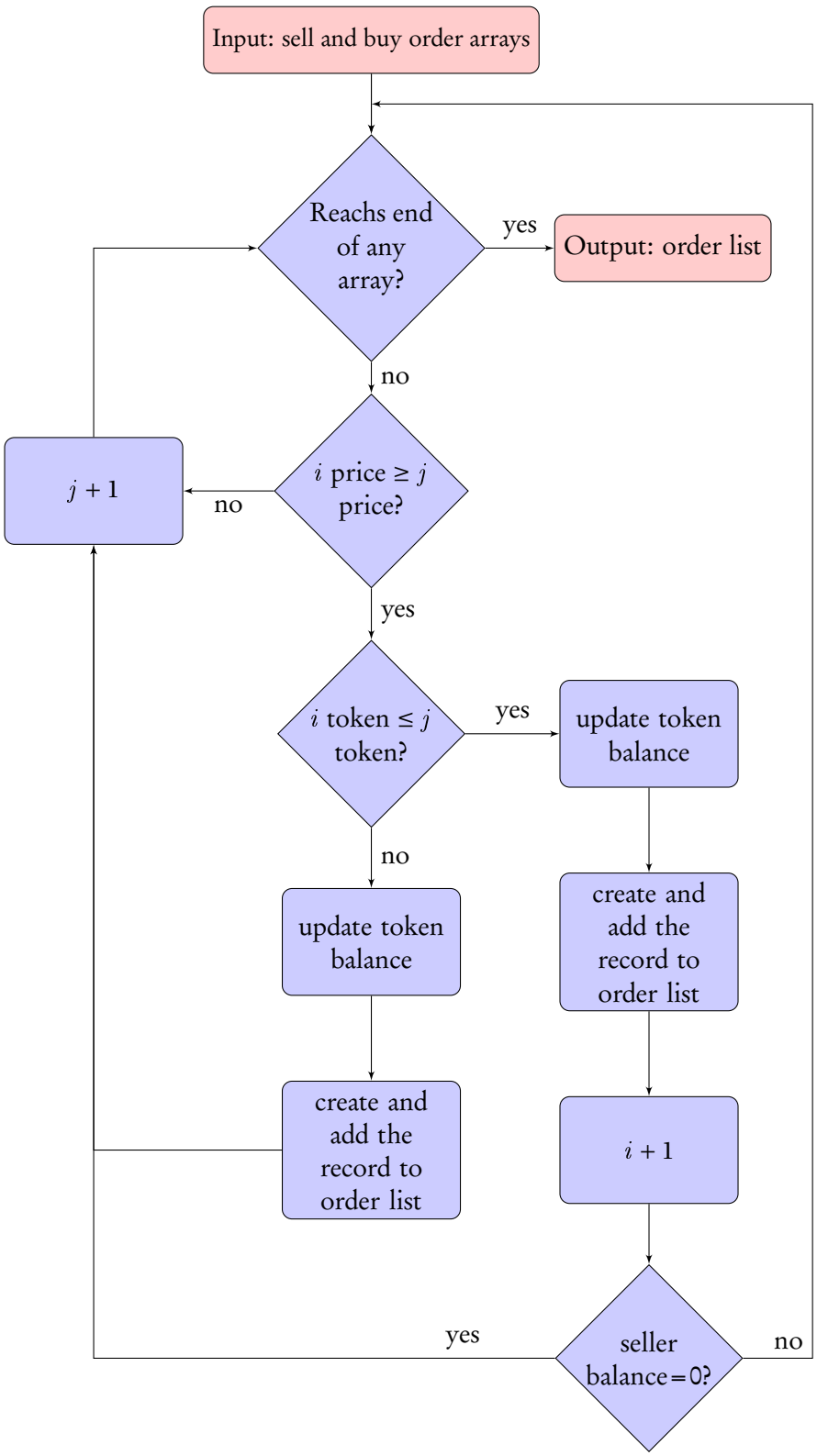


Figure 5.3 – Trading algorithm diagram

```
1 type order = {orderAddress : address;
2               tokens: uint;
3               price_order: uint}
4
5 type order_trading = {seller_index: uint;
6                       buyer_index: uint;
7                       amount_t: uint}
```

Listing 5.11 – Additional types in *WhyML* contract

The trading function takes two `arrays` of `orders` (buy and sell) and outputs a `list` of `order_trading`; the signature is as follows [3]:

```
let trading (buy_order : array order) (sell_order : array order) : list order_trading
```

Properties to prove. The `trading` function matches a potential buyer with a potential seller, recorded in two arrays; `buy_order` and `sell_order`. The function's properties must be respected to obtain an expected result at the end of the execution. `trading` is a private function; thus, no exceptions are defined, but preconditions are. Listings 5.12 illustrates the set of properties defined for the private trading function.

```
1  let trading (buy_order : array order) (sell_order : array order) : list order_trading
2   requires {length buy_order > 0 ∧ length sell_order > 0}
3   requires {sorted_order buy_order}
4   requires {sorted_order sell_order}
5   requires {forall j:int. 0 ≤ j < length buy_order → 0 < buy_order[j].tokens }
6   requires {forall j:int. 0 ≤ j < length sell_order → 0 < sell_order[j].tokens  }
7   ensures { correct result (old buy_order) (old sell_order) }
8   ensures { forall l. correct l (old buy_order) (old sell_order) → nb_token l ≤ nb_token result }
9   ensures {!gas ≤ old !gas + 374 + (length buy_order + length sell_order) * 363}
10   = (*The program*)
```

Listing 5.12 – The `trading` function specification

Moreover, to prove the pre- and postconditions, the function must define loop invariants since the function defines a loop that iterates over the arrays given as inputs (see Figure 5.3). Hence, the `trading` function must respect the following functional properties and the absence of RTE:

- **RTE**: *(1) Positive values*; the parameters of the functions must not be empty (empty array, line 2), in which case the trading can not occur. *(2) Index out of array bounds* (lines 5-6); the corresponding invariants can be defined as follows:

```
1 invariant {forall k:int. !i ≤ k < length (buy_order at Before) → 0 < buy_order[k].tokens}
2 invariant {forall k:int. !j ≤ k < length (sell_order at Before) → 0 < sell_order[k].tokens }
3 invariant {0 ≤ !i ≤ length (buy_order) ∧ 0 ≤ !j ≤ length (sell_order )}
```

The variables `i` and `j` are the ones that iterate on, respectively, the buy array and the sell array order.

- **Functional properties**: the function is intended to match consumers and producers. We favour consumers; thereby, sellers can only provide energy to consumers who make an offer to buy at a price greater than or equal to the selling offer. We defined four properties to prove:

(1) *Sorted arrays*; the inputs must be sorted (lines 3-4); thus, the predicate `sorted_order` must be satisfied both for `buy_order` and `sell_order`. In *Why3*, arrays are defined according to the sequence type from the library `seq.Seq`. As a result, we can apply operations to arrays that require sequence inputs since *Why3* makes the correspondence automatically:

```
1 predicate sorted_order (a: seq order) =
2  forall k1 k2 : int. 0 ≤ k1 ≤ k2 < length a → a[k2].price_order ≤ a[k1].price_order
```

[3] The complete code of the function can be found in the following website http://francois.bobot.eu/fm2019/BEMP.mlw

As an argument, the predicate takes a sequence of orders (`seq order`). It ensures that the sequence elements are sorted in decreasing order according to the price.

(2) *Correct trading*; the trading function must be correct at the end of its execution (line 7) according to its inputs and its output `result`. The postcondition uses a predicate "`correct`" to express the property to prove. Its definition is as follows:

```
1 predicate correct (l:list order_trading) (buy_order: seq order) (sell_order: seq order) =
2 (forall i:uint. 0 ≤ i < length sell_order → sum_seller l i ≤  sell_order[i].tokens) ∧
3 (forall i:uint. 0 ≤ i < length buy_order → sum_buyer l i ≤  buy_order[i].tokens) ∧
4        matching l buy_order sell_order
```

The predicate has three members; the first one expresses that, at the end of the trading, the seller has not sold more tokens than it owned before the trading execution. We define a function that takes as inputs a list of trading orders and an integer that represents the index of the seller and outputs an integer which is the calculated sum. The function is as follows:

```
1 function sum_seller (l : list order_trading) (sellerIndexe : int) : int
2  =
3  match l with
4  | Nil → 0
5  | Cons h t → ( if  h.seller_index = sellerIndexe then h.amount_t
6    else 0 ) + sum_seller t sellerIndexe
7  end
```

`sum_seller` is a recursive function that defines pattern-matching over the list of trading orders. `h` represents the head of the list `l` (the first order element of the list) and `t` the tail (the rest of the list). If the `seller_index` member of `h` is equal to the integer given as a parameter, `sellerIndexe`, then the function returns the `amount_t` of `h`. This returned value is added to the value returned by the `sum_seller` function applied to the list's tail `t`. The execution of this recursive function ends when `l` matches with `Nil`, i.e. we have gone through the whole list. Thus, we obtain the calculation of the sum of tokens that a seller has sold.
The same goes for the second member of the predicate `correct`, that, at the end of the trading, the buyer will not have purchased more tokens than it had requested at the beginning of the trading. The function is as follows:

```
1 function sum_buyer (l : list order_trading) (buyerIndexe : int) : int
2  =
3  match l with
4  | Nil → 0
5  | Cons h t → ( if  h.buyer_index = buyerIndexe then h.amount_t
6    else 0 ) + sum_buyer t buyerIndexe
7  end
```

Similarly to `sum_seller`, the function `sum_buyer` calculates the sum of the `amount_t` value of the index `buyerIndexe`. At the end of the execution, we get the sum of `amount_t` bought by the buyer.
The third member of the predicate ensures a correct matching which means no seller will match a buyer willing to pay less than its price. The recursive `matching` predicate is defined as follows:

```
1 predicate matching (order: list order_trading) (b_order : seq order) (s_order : seq order) =
2       match order with
3       | Nil → true
4       | Cons k l → matching_order k b_order s_order ∧ matching l b_order s_order
5       end
```

The predicate defines a pattern-matching over the list `order`. It uses the predicate `matching_order` on the first element of the list, `k`, and recursively applies `matching` on the rest of the list `l`. The predicate `matching_order` ensures that the price of the order `k` is no more than the price offered by the buyer `k.buyer_index`. Moreover, the predicate ensures a positive order amount `k.amount_t` and that the buyer and seller indexes are in the inputs bounds.

```
1 predicate matching_order (k: order_trading) (b_order :seq order) (s_order : seq order) =
2  s_order[k.seller_index].price_order ≤ b_order[k.buyer_index].price_order  ∧
3  0 ≤ k.buyer_index < length b_order ∧ 0 ≤ k.seller_index < length s_order ∧ 0 < k.amount_t
```

Provers	Number of proofs	Time (seconds)		
		minimum	*maximum*	*average*
Z3 4.6.0	2	0.04	0.49	0.27
Alt-Ergo 2.2.0	205	0.00	1.34	0.07
Alt-Ergo 2.3.0	572	0.00	0.47	0.03
CVC4 1.6	507	0.04	0.96	0.12

Table 5.2 – Statistics per prover applied to BEMP

(3) *Best tokens exchange*:

```
ensures { forall l. correct l (old buy_order) (old sell_order) → nb_token l ≤ nb_token result }
```

We choose to qualify the trading as one of the best if it maximises the total number of tokens exchanged. Whatever the correct trading, our solution will be the most optimal in terms of the number of tokens exchanged. Our solution will always have more or as many tokens exchanged as another correct trading. `nb_token`, defined below, is a recursive function that outputs the sum of traded tokens of the order list given as input.

```
function nb_token (l : list order_trading) : int =
 match l with
 | Nil → 0
 | Cons h t → h.amount_t + nb_token t
 end
```

(4) *Gas consumption*: when a function consumes more *gas* than expected, an out-of-gas exception is raised. The following property

```
ensures {!gas ≤ old !gas + 374 + (length buy_order + length sell_order) * 363}
```

ensures that, at the end of the execution, the trading function consumes precisely or less than the calculation of "`374 + (length buy_order + length sell_order) * 363`". The `gas` consumption depends on the length of both lists. 374 and 363 are constants that have been calculated according to the operations that constitute the trading function. In the next section, we detail how to obtain these calculations.

This second version of the case study allowed it to express more properties than the first version. We have defined a smart contract that allows us to create a list of trading orders and define complex properties. Table 5.2 gives some metrics concerning the writing and the proof of the *WhyML* smart contracts. The updated case study consists of 353 lines of specification code and 327 ligne of implementation code. We notice that the specification part in smart contracts is as important as the code itself. This result supports the interest of *WhyML* in writing formal and proven smart contracts.

5.3 Compiling *WhyML* Contracts and Proving *gas* Consumption

This section aims to describe the approach to compile *WhyML* contracts into EVM. In the first step (Section 5.3.1), we explain how the compilation in EVM works with *Why3*, and in the second step (Section 5.3.2), we explain how to calculate the expected *gas* consumption of a function and the method of proving such consumption.

5.3.1 The Ethereum Virtual Machine (EVM) and *Why3*

The final step of the deductive verification approach is the deployment of *WhyML* contracts. EVM is designed to be the runtime environment for the smart contracts on the Ethereum blockchain [187]. Smart contracts are like regular accounts, except they run EVM bytecode when receiving a transaction, allowing them to perform calculations and further transactions.

The EVM is a stack-based machine (word of 256 bits) and uses a set of instructions called opcodes[4] to execute specific tasks. The EVM features two memories, one volatile that does not survive the current transaction and a second for storage that does survive but is a lot more expensive to modify. Opcodes are encoded to bytecode to be efficiently stored, and each opcode is allocated a byte (for example, the opcode `ADD` is `0x01`).

The compilation[5] in itself is straightforward; it is done in three phases: *(1)* the compilation to an EVM that uses symbolic labels for jump destination and macro instructions; *(2)* computing the absolute address of the labels, it must be done inside a fixpoint because the size of the jump addresses has an impact on the size of the instruction, and *(3)* translating the assembly code to pure EVM assembly and printing it.
Most of the *WhyML* statements can be translated into opcodes. The proof-of-concept compiler (an extraction module of *Why3*) allows using algebraic data types without nesting pattern-matching, mutable records, recursive functions, while loops, and integer bounded arithmetic (32, 64,128, 256 bits). Global variables are restricted to mutable records with fields of integers. It could be extended to a hashtable using the hashing technique of the keys used in *Solidity*. Without using specific instructions, like for C, *WhyML* is extracted to garbage-collected language; here, all the allocations are done in the volatile memory, so the memory is reclaimed only at the end of the transaction.
We have not formally proved the correction of the compilation yet. We only tested the compiler on function examples using a reference interpreter [6] and asserting some invariants during the transformation (*WhyML* code to EVM).

However, we could list the following arguments for the compilation improvement:

1. The compilation of *WhyML* is straightforward to stack machines.
2. The precondition on all the arithmetic operations (always bounded) ensures arithmetic operations could directly use 256bit operations.
3. Raising exceptions are accepted only in *public* function before any mutation, so the fact they are translated into $REVERT$ opcode does not change their semantics.
4. Only immutable data types can be stored in the permanent store. Currently, only integers can be stored; they could be extended to other immutable data by copying the data to and from the store.
5. The send function in *WhyML* only modifies the state of balance of the contracts and requires that the transfer is acceptable and never fails, as discussed previously. So it is compiled similarly to the *Solidity* function send function with a *gas* limit small enough to disallow modification of the store.
6. The *public* functions are differentiated from *private* ones using the attribute `[@ evm:external]`. The *private* functions do not appear in the dispatching code at the contract entry point, so that they can be called only internally.

5.3.2 The Calculation of the *gas* Consumed by a Function

The execution of each bytecode instruction has an associated cost. When sending a transaction, one must pay some *gas*; if there is not enough *gas* to execute the transaction, the execution stops, and the state is rolled back. Therefore, it is essential to be sure that the execution of a smart contract will not require an excessive quantity of *gas* at any later date. The computation of WCET (Worst-Case

[4] https://ethervm.io
[5] The implementation can be found at http://francois.bobot.eu/fm2019/
[6] https://github.com/ethereum/go-ethereum

Execution Time) is facilitated in EVM by the absence of cache. WCET is a software development metric that determines the maximum length of time a task or set of tasks requires on a specific hardware platform.
To track the amount of *gas*, we could use techniques of [17] which annotates in the source code the quantity of *gas* used. However, in our approach, we use the function `add_gas` (defined in Listing 5.4) to calculate the quantity of *gas* consumed.

The following code in Listing 5.13 is a basic *WhyML* contract, `module A`, consisting of a simple function `ite` that takes as parameter a machine integer `x`. The contract needs some external modules to write its function as the module `Int` for integer. The function checks whether the integer is negative or null. If it is, the function returns `0`; otherwise, it returns 1. The function calls in lines 9 and 11 the `add_gas` ghost function defined in the module `mach.evm.Gas`.

```
1  module A
2    use int.Int
3    use mach.int.UInt32
4    use mach.int.Int32
5    use mach.evm.Gas
6
7     let ite (x: int32) : int32
8     =
9      add_gas 59 0;
10     if x ≤ 0 then 0 else
11     (add_gas 10 0; 1)
12 end
```

Listing 5.13 – An example of a simple *WhyML* contract

```
Startgas(0) JUMPDEST(Lsym:ite)
Addgas(59) PUSH1(00) DUP2
DUP2 DUP2 SLT SWAP2 EQ OR
JUMPI(ifthen) Addgas(10)
PUSH1(02) JUMP(ifend)
JUMPDEST(ifthen) PUSH1(00)
JUMPDEST(ifend) SWAP1 POP
SWAP1 JUMPDYN Stopgas(0)
```

Listing 5.14 – The *WhyML* contract in opcode

During the compilation, the *WhyML* code is translated into opcodes. Each opcode has a specific instruction, and each opcode is stacked on the other. The resulted translation of the *WhyML* contract is represented in Listing 5.14. Executing opcodes consumes a quantity of *gas*, and the cost ranges from `0` to over 32000. According to the path taken by the function (either the *if* or the *or* path), the cost of the function will be different. Let us notice that the compiler does not execute `Startgas`, `Addgas` and `Stopgas`; we added them for clarity of code reading. The compilation checks that all the function paths have a cost smaller than the sum of the `add_gas` on it. Paths of a function are defined on the EVM code by starting at the function-entry and loop-head and going through the code following jumps that are not going back to the loop-head.

We assign to each opcode their corresponding *gas* cost. Let us consider the `if` path. The beginning of *gas* calculation starts with `Startgas`. The first instruction is the `JUMPDEST` which costs 1 unit of *gas* and marks a valid destination for jumps. The next instruction is the `Addgas` which consists of an argument of the value 59. This quantity corresponds to the *gas* function consumption estimation for the `if` path that the developer defines. The followed opcodes have a specific action and a fixed amount of *gas* consumed. Table 5.3 gives the action of each along with their *gas* consumption.
Since we consider the `if` path, when the program counter arrives at the opcode `JUMPI(ifthen)`, it will jump to the indicated valid destination, which is `JUMPDEST(ifthen)` and then continue to execute the opcodes until reaching `Stopgas(0)`. Therefore, the following sequence of opcodes are not executed: {`Addgas(10)`, `PUSH1(02)`, `JUMP(ifend)`}, because they refer to the `else` path.
At the end of the function execution, we must obtain `0` or less when we sum all the opcodes values minus the value of *gas* consumption indicated by the developer, i.e. 59.
If we do the same for the `else` path, we will have the call of the `Addgas` function two times. Therefore, instead of consuming 59 units of *gas*, the function must consume 69 units of *gas*. When we sum all opcodes costs minus 69, we must get `0`, which means that the estimation corresponds to the actual cost of the function. The calculation stopped when we met `Stopgas`.

Mnemonic	Cost in *gas*	Description
JUMPDEST	1	Mark a valid destination for jumps.
JUMPI	13	Conditionally alter the program counter.
JUMP	11	Alter the program counter.
PUSH1	3	Place 1 byte item on stack.
DUP2	3	Duplicate 2nd stack item.
SLT	3	Signed less-than comparision.
SWAP2	3	Exchange 1st and 3rd stack items.
EQ	3	Equality comparision.
OR	3	Bitwise OR operation.
SWAP1	3	Exchange 1st and 2nd stack items.
POP	2	Remove item from stack.

Table 5.3 – Extract from [187] of some opcodes with their description and corresponding *gas* consumption

Similarly to the trading specification of *gas* consumption, we ensure that the function will consume exactly or less at the end of the execution, either 59 or 69, according to the function's path. The corresponding postcondition will be:

```
1 ensures { !gas - old !gas ≤ (if x ≤ 0 then 59 else 69) }
```

If a function needs to allocate memory, a postcondition must be defined to ensure that the function will not run out of memory. For example, the following code in Listing 5.15 is a function that takes as a parameter the positive size (line 2) of the list to build and returns it. Through this example, we want to show the memory allocation according to the path taken by the function (`if` path no dependent on `i`, or `else` path dependent on `i`). Since it is a recursive function (`rec` key-word), we need to add a variant (line 2) to prove the termination.

```
1 let rec create_list (i:int32) : list int32
2  requires { 0 ≤ i }
3  ensures { i = length result }
4  ensures { !gas - old !gas ≤ i * 185 + 113 }
5  ensures { !alloc - old !alloc ≤ i * 96 + 32 }
6  variant { i }
7   =
8    if i ≤ 0 then (add_gas 113 32; Nil)
9    else (let l = create_list (i-1) in add_gas 185 96; Cons (0x42:int32) l)
```

Listing 5.15 – A function example to calculate memory allocation

The output list of the function must have the length of the input integer (line 3). Line 4 is a postcondition that ensures a correct *gas* consumption which depends on the value of `i`. As a result, 113 corresponds to the minimum quantity of *gas* that the function consumes. The larger the value of `i`, the greater the amount of *gas* consumed—the same observation for the memory allocation in line 5. The function needs a minimum of 32 units of memory to execute the function.
Once we get the appropriate information about *gas* and allocation, we can lift this information using the *WhyML* specification to prove that a function that given `i` builds a list of length `l` has a cost smaller than $185i+113$ and allocates at most $96i+32$ bytes. Currently, the cost of the modification of storage is over-approximated; we could specify that it is less expensive to use a memory cell already used.

5.4 Conclusion

In this chapter, we applied concepts of deductive verification to a computer protocol intended to enforce some transaction rules within an Ethereum blockchain application. The aim is to avoid errors that could have serious consequences. Reproduce, with *WhyML*, the behaviour of *Solidity* functions showed that *WhyML* is suitable for writing and verifying smart contracts programs. In this theorem proving approach, we define mathematical statements to be proved as preconditions,

postconditions, and invariants. Furthermore, because the *Solidity* language contains elements that are not part of the *WhyML* language, we built a *WhyML* library dedicated to *Solidity* expressions. The presented method was applied to a use case that describes an energy marketplace allowing local energy trading among inhabitants of a neighbourhood. The resulting modelling allows establishing a trading contract to match consumers with producers willing to make transactions. This last point demonstrates that with a deductive approach, it is possible to model and prove the operation of the BEMP application at a realistic scale. We manage to prove the application that matches m consumers with n producers, contrary to model-checking in [148]; thus, verifying more realistic functional properties. However, the user is asked to write the invariants in the presented approach, which can be hard to achieve.

Part IV

Formalisation and Proof of a Blockchain Distributed Algorithm based on Smart Contracts

Chapter 6

Distributed *Cross-Chain Swap* Algorithm

> " *Aerodynamically the bumble bee shouldn't be able to fly, but the bumble bee doesn't know it so it goes on flying anyway.* "
>
> - Mary Kay Ash

Contents

The previous chapter showed an application based on smart contracts, BEMP, that runs sequentially. Therefore, proving the correctness of the smart contract was enough to ensure that the BEMP application works properly. However, some applications based on smart contracts require users' collaboration and need a distributed execution. In that case, proving the smart contract's code is not enough to ensure that the application works properly, as the behaviour of the users comes into account. In addition to the proof of smart contracts' code, it is necessary to prove the underlying distributed protocol.
This chapter addresses these issues of distributed applications exploiting smart contracts. Recently, one application based on smart contracts has gained popularity, namely cross-chain swap applications [99, 101, 102, 104, 169, 176, 195]. *These applications allow users of different blockchains to trade assets in a decentralised manner and without the involvement of an intermediary. In this chapter, we describe the cross-chain swap problem in Sections 6.1 and 6.2, and its formal modelisation in Section 6.3. We proceed in two steps; the first is modelling the protocol into state machines based on verifiable proofs (Section 6.4), and the second is implementing the model into a formal language called TLA$^+$ (Section 6.5).*

6.1 *Cross-Chain Swap* Problem

The studied distributed application in this chapter is the *cross-chain swap*. These applications find their use in blockchain systems. The aim is to achieve the exchange of assets between participants of different blockchains. The underlying protocol's difficulty is to ensure the security and the protection of the assets and the correct participants. Most *cross-chain swap* protocols use smart contracts to secure transactions, especially the exchanged assets. Smart contracts are used to put in escrow assets for the duration of the protocol to avoid double-spending, i.e. sending an asset to two different recipients. Although *cross-chain swap* protocols find their application in blockchains, we propose a protocol that abstracts blockchain implementations. We chose to bring a level of abstraction to our protocol that can be applied to other distributed ledgers than blockchains. The design provides a generic approach to the problem.

6.1.1 System Model

Participants

The system is composed of a set of participants Π and a set of assets Λ. The participants run the system, and they can be of two types, Π_s and Π_r with $\Pi = \Pi_s \cup \Pi_r$. Π_s represents the set of participants transferring assets, and Π_r is the set of participants receiving transferred assets. A participant in Π_s can be in Π_r, and conversely, if it receives and sends one or more assets. Participants have a local clock to timestamp events. We assume that each participant in the model is, by default, asynchronous. They communicate by sending and receiving messages, and all messages are digitally signed; hence we assume that they cannot be forged. We define some assumptions on messages; there are no duplication messages, and a message is received at most once. Moreover, there is no creation message, and no message is received unless some participant did send it. If a participant sends a message to a participant and both are correct, the message is eventually delivered.

Assets

An asset is any entity having a specific value and a unique owner (who is also a participant). It can be a cryptocurrency or a physical asset's ownership certificate. A participant can own assets and can transfer its asset's ownership to another participant. Throughout the study, we use the term "asset" for reasons of clarity, but it should be remembered that it is the "asset's ownership", as defined in **Definition** 6.1, that is transferred and not the physical asset. Moreover, assets' ownership can be tokenised to facilitate the transfer through the network (tokenisation is defined in Section 2.2.2) .

Definition 6.1. (*Asset Ownership*). An asset is defined by its value (economic value or future benefit) and its owner. The ownership of an asset is unique to each asset.

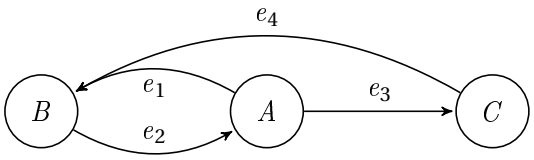


Figure 6.1 – A swap graph S with $\Pi = \{A, B, C\}$ and $E = \{e_1, e_2, e_3, e_4\}$

Definition 6.2. (*Transferring Asset*). The transfer of an asset from participant A to participant B is the attribution of the asset's ownership to participant B.

6.1.2 Swap Model

A *swap* is a distributed transactions [1] model. The objective is to transfer assets between participants across multiple distributed ledgers in a trustless environment without an intermediary. In a swap, the number of participants and assets is finite. A participant runs the system; thus, it is a user of a distributed ledger involved in the swap. Therefore, it can be a *source*, a *recipient* or both. The *source* transfers its asset's ownership, as defined in **Definition** 6.2, and the *recipient* receives a transferred asset's ownership. Moreover, there are no constraints to one source transferring multiple assets and one recipient receiving multiple transferred assets within the same swap.
A swap S is modelled as a directed graph $S = (\Pi, E)$ (see Figure 6.1). S is composed of a set of vertices Π (the set of participants) and a set of labelled edges $E = \{e_1, e_2, ..., e_m\}$. The label of an edge is the transferred asset. Each edge of S transfers a unique asset from the set of assets, Λ, involved in the swap. Consequently, $|E| = m$ represents the total number of transferred assets in S. An edge is defined as $e_i = (s, a_i, r) \in \Pi_s \times \Lambda \times \Pi_r$ with $i \in \{1, .., m\}$, $s \neq r$, and a_i the label of the edge that designates the transferred asset. Moreover, Π_s is the set of participants transferring assets, "*sources*" (vertices with outgoing edges) and Π_r the set of participants receiving transferred assets, "*recipients*" (vertices with incoming edges).
Note that a participant who is both a source and a recipient will have two different representations. For example, participant A in Figure 6.1 is a source for edges $\{e_1, e_3\}$; hence it will be represented by s_A, and is a recipient, represented by r_A, for the edge $\{e_2\}$. A source and a recipient perform actions on assets defined in **Definition** 6.3 and **Definition** 6.4.

Definition 6.3. (*Recovers*). Recovering an asset is an action only performed by a source participant, which means that the source takes over its asset ownership.

Definition 6.4. (*Retrieves*). Retrieving an asset is an action only performed by a recipient participant, which means that the recipient takes ownership of the received asset.

When a recipient retrieves an asset, it receives the transferred asset's ownership and becomes the asset's new owner.

Remark. (*Swap Graph Construction*). We assume that, before the swap, the graph is constructed by all the participants. Thereby, they agree with its configuration. How the graph is built is not part of our study. We will see later that the graph, once constructed, is visible to all participants. Therefore, an error or fraud in the graph construction, for example, the wrong source identifier for an edge, will be identified. Accordingly, one could imagine an algorithm that will construct the graph by having the list of the recipients and the sources along with their assets.

6.2 Problem Definition

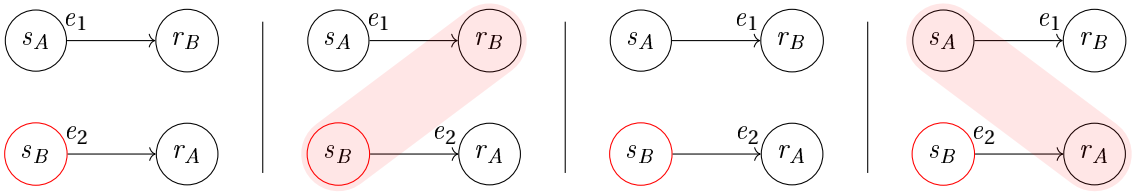
This section introduces the specification of the swap problem. It is defined in three properties, one of safety; *Consistency* and two of liveness; *Ownership* and *Retrieving*.

[1] Following blockchain terminology, a transaction is a payment or set of payments, not an atomic unit of synchronisation as in databases or transactional memory.

The *Consistency* property. A swap must ensure properties to protect the correct participants and the exchanged assets. *Consistency* is a safety property that does not require *synchrony* to be satisfied. The property states that no correct participant will end up worse off. In several *cross-chain swap* protocols, one of the safety properties that often comes up is *atomicity* [101, 191]. This property aims to prove that asset transfers occur in an atomic manner, so either all transfers occur or none. Therefore, it also proves that no correct participant terminates worse off. However, since the system tolerates Byzantine participants, the classical atomicity definition "*all-or-nothing*" cannot be applied, as said in [102]. It is impossible to force a participant to initiate its asset transfer. Moreover, one can have a situation where assets are transferred, by Byzantine sources, even if the swap does not authorise the transfer of the assets. For this reason, safety is intended to be weaker than classical atomicity while ensuring that a correct participant will always terminate safely. Therefore, we define the *Consistency* property as follows:

Definition 6.5. (*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 the swap execution, either s_1 owns a_1 or r_2 owns a_2.*

Reasoning about a correct source and recipient pair is sufficient to extrapolate the property to all pairs of correct participants. Let us take a simple example to illustrate the property by simplifying Figure 6.1. We only consider the transfer of the assets between the participants A and B. Below, we give the four possible combinations of the transfer of the assets. For the example, we consider the participant B, when it is a source, as a Byzantine participant (represented by a red circle), and all other participants are correct. We suppose that the participant B decides to be correct when it is a recipient. In addition, the position of the edge label indicates whether the source or the recipient owns the asset. If the label is near the node, the node owns the asset corresponding to the label. Therefore, among the four possible combinations of transfers, two seem to violate the *Consistency* property (the pair of source-recipient highlighted in red). The first case involves the Byzantine source; thus, this case does not violate the property, and it is an acceptable combination. The second case does not respect the *Consistency* property as both s_A and r_A are correct. This combination is not acceptable by the property and must not occur.



This example shows that considering only a pair of correct source-recipient is sufficient to deduce the property to all pairs of the swap, thus avoiding the limitation of checking the execution completion of all correct participants.

The *Ownership* property. *Ownership* is a property that does not require *synchrony* to be satisfied. Through this property, we wish to provide guarantees on assets' ownership so that they are never lost forever. The *Ownership* is defined as follows:

Definition 6.6. (*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.*

The *Ownership* property comprises 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 will never end up worse off. Thereby, it will always either *retrieve* or *recover* its asset(s) asynchronously.

The conjunction *or* in the property seems unusual in the expression of the properties, and instinctively one would think of the conjunction *and*. The *Ownership* property expresses the ownership of assets according to the outcome of the swap. The first member of the conjunction, which is "*No asset owned initially by a correct source is ownerless forever*", ensures that if the swap does not take place, the correct sources will recover their assets. The second member of the conjunction, which is "*No asset intended to be transferred to a correct recipient is ownerless forever*", ensures that if the swap takes place, the correct recipients will retrieve their assets. As the two swap results cannot happen simultaneously, it is the *or* conjunction that must be chosen. Let us take the example of A and B again. Suppose the second combination results from a swap that does not take place. In that case, the first member of the *Ownership* property is satisfied since the asset corresponding to the label e_1 is initially owned by s_A (a correct source), and the asset of e_2 is initially owned by a Byzantine source s_B.

The *Retrieving* property. *Retrieving* property is a property that requires *synchrony* to be satisfied. The *Retrieving* property state the desired outcome in the case where all participants are correct.

Definition 6.7. (*Retrieving*). *If all participants are correct then all recipients will retrieve their intended assets.*

This property assumes strong assumptions, such as the mode of communication and the participants' behaviour. However, this property allows us to state the ideal case of the protocol and avoids any empty protocols.

6.3 Protocol Specification

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

6.3.1 Representation of Asset's States in a Swap

This part introduces 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 (see Figure 6.2):

- 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 detail this point later.

The participants have operations that, once executed, cause a change in the asset's state. The protocol interacts with assets through trigger[2] events ϵ_i, where $i \in \mathbb{N}$. Triggers make it possible to modify the states of the assets.
An asset can change its state legally (following an action made by a correct or a Byzantine participant; plain edges $\rightarrow$) or illegally (following an action made by a Byzantine participant, dashed edges $\dashrightarrow$). The meaning of the term legally means following the rules of the swap.

[2] In the study context, a trigger is a mechanism that initiates an action when an event occurs.

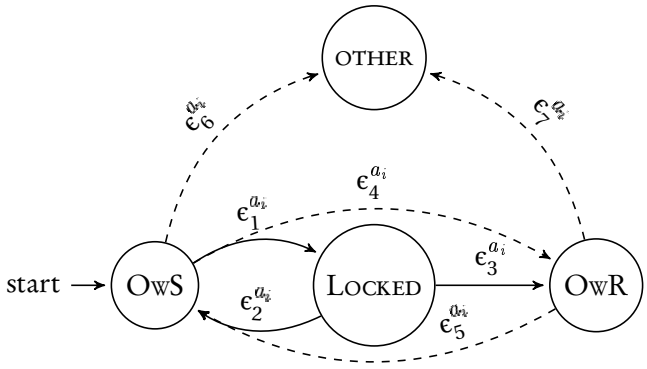


Figure 6.2 – Representation of an asset a_i possible states

Sources have two operations: (1) locking its asset a_i and assigning r_i as the new owner of the asset ($\epsilon_1^{a_i}$ in Figure 6.2), and (2) recovering its asset a_i and becoming again the owner of a_i ($\epsilon_2^{a_i}$). Recipients have one operation, which is retrieving the asset a_i and becoming the new owner of a_i ($\epsilon_3^{a_i}$).
Moreover, a Byzantine participant has actions that can also change the assets' state. Their actions are the following (see Figure 6.2):

- $\epsilon_4^{a_i}$: a Byzantine source transfers its asset directly to the recipient without passing through the swap.
- $\epsilon_5^{a_i}$: a Byzantine recipient, once it retrieves its asset, can send back the asset to the original owner, the source.
- $\epsilon_{\{6,7\}}^{a_i}$: a Byzantine source or recipient can transfer its asset to an unknown participant or lock it somewhere or perform all other actions not recognised by the swap.

We can see from Figure 6.2 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 significant information since the outgoing edges would cancel the incoming edges.

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

The abstract protocol, $\mathscr{P}_{swap}$, is modelled as a set of state machines that influences the assets' state introduced in Section 6.3.1.

Overview of the Protocol $\mathscr{P}_{swap}$

$\mathscr{P}_{swap}$ is inspired by the defined protocol in [191]. The idea is similar to the well-known *Two-Phase Commit* protocol [37] (defined in Section 2.1.1). 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. A special entity, known as a coordinator, is required for a *Two-Phase Commit* to take place. The coordinator decides whether to commit or abort the transaction and communicates the result to all the participants.
In $\mathscr{P}_{swap}$ the coordinator is defined as a public entity. We assume a communication channel between the coordinator and each participant, but we do not assume direct communication among the participants during the swap. The behaviour of each participant is independent of the others. On the other hand, the coordinator's behaviour influences the participants and vice versa. We make no assumptions about participants' behaviour; thus, they can behave arbitrarily, i.e. be a Byzantine participant.

After constructing the swap graph, all correct participants have a local copy of it. All sources must lock their asset(s) to prove their wish to commit the swap. The coordinator has the role of authorising the swap or not by giving a decision to the swap participants. Only the coordinator's decision can unlock the assets. The possible decisions are the *redeem* decision to authorise the swap or the *refund* decision to abort the swap.

The coordinator correctness. The coordinator has a central role in this protocol since it is inspired by the *Two-Phase Commit* algorithm. $\mathscr{P}_{swap}$ wants to be tolerant to possible Byzantine attacks from the participants but also from the coordinator. Thus, we assume that correct participants can evaluate the correctness of the coordinator. If the coordinator is Byzantine, the swap could not start in the first place because correct participants will abandon the swap. From this premise, if the swap starts, we assume the coordinator is correct. Therefore, to simplify the description of the protocol, it is necessary to assume a correct coordinator.

Proof-of-Actions

The protocol is tolerant to an unbounded number of Byzantine participants. Thus, our properties must hold despite their presence. The protocol uses a method to withstand Byzantine attacks that allow countering their behaviour called *proof-of-actions*. A *proof-of-action* means proving to someone that a particular action has been performed. A *proof-of-action*, once provided, cannot be forged, even if a Byzantine participant provides the proof. In cryptography, similar methods are used, like the *zero-knowledge protocol* [91]. In addition, in the context of blockchains, a transaction stored in a block can be a reliable *proof-of-action*. Since blockchain data is unforgeable, it is easy to prove whether or not an action has been carried out. In the case of $\mathscr{P}_{swap}$ protocol, the coordinator and participants can verify executed actions in the swap by using *proof-of-action*. The coordinator and participants can produce proof that a given action or state change is correctly done. This proof cannot be falsified. If any proof is false, then it will be detected. If a given action is correctly done, the proof is valid.

6.3.3 Participants State Machines

$\mathscr{P}_{swap}$ interacts with participants of the swap, consisting of three kinds of participants; a *publisher*, a *coordinator*, *sources*, and *recipients*. Their behaviour is represented by a state machine structured with the following elements $(\Gamma,\ Q,\ \Sigma,\ \delta,\ q_0,\ F)$ (see Section 2.1.2 for more details about the participant's state machine).
In addition, Σ contains three parts (each one is optional), written $q \xrightarrow{\epsilon;\sigma;\omega} q'$ with 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. Actions and operations can contain arguments. The symbol $\oslash$ is used where the label does not contain one of the three parts.
We recall that participants have a local clock and a timeout for each step where the participant waits for a coordinator's action.

The Publisher. The publisher is a participant in Π, represented by Figure 6.3 and defined by the elements of Table 6.1. Its role is to publish the swap graph $swap$ to the coordinator with the action ϵ_1^p. A publisher can also be a source or a recipient. How the publisher is selected is not part of our study. Although this step could be done through a leader election algorithm or randomly. We assume that for each swap, only one publisher is selected. However, in the following, we explain how we maintain the swap properties in the case of several publishers. Furthermore, we explain why the selection method does not violate the swap properties.

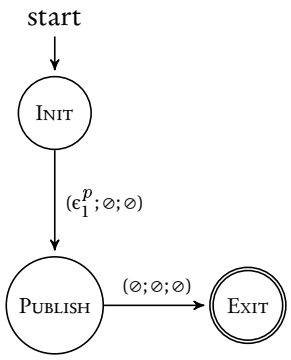


Figure 6.3 – State machine of the publisher

Q_p	$= \{\text{INIT}, \text{PUBLISH}, \text{EXIT}\}$
Σ_p	$= \{(\epsilon_1^p;\varnothing;\varnothing), (\varnothing;\varnothing;\varnothing)\}$
δ_p	$= \begin{cases} \text{INIT} \times (\epsilon_1^p;\varnothing;\varnothing) \longmapsto \text{PUBLISH} \\ \text{PUBLISH} \times (\varnothing;\varnothing;\varnothing) \longmapsto \text{EXIT} \end{cases}$
q_{0_p}	$= \{\text{INIT}\}$
F_p	$= \{\text{EXIT}\}$
ϵ_1^p	$= publish(swap)!$

Table 6.1 – Elements of the publisher

The Coordinator. The role of the coordinator is to coordinate the evolution of the swap. It is represented by Figure 6.4 and defined by the elements of Table 6.2. 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. In ϵ_1^c, the coordinator waits for the publisher to execute the $publish(swap)!$ action. Then, in ϵ_2^c, the coordinator waits for the participants to ask for a *refund* or a *redeem* decision. If σ_3^c is true (resp. σ_4^c), it satisfies $\sigma_4^{s_i}, \sigma_5^{r_i}$ (resp. $\sigma_6^{s_i}, \sigma_3^{r_i}$) from Figures 6.5 and 6.6. We define a predicate ValidTransfer as the conjunction of the swap's conditions to allow the transfer of assets. The predicate is conditioned by a valid *proof-of-action*, $Proof_{lock}$, given as a parameter (the definition of $Proof_{lock}$ is detailed later). 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. Both predicates, ValidTransfer and AbortTransfer, are mutually exclusive. We define the two predicates in more detail in the next section.

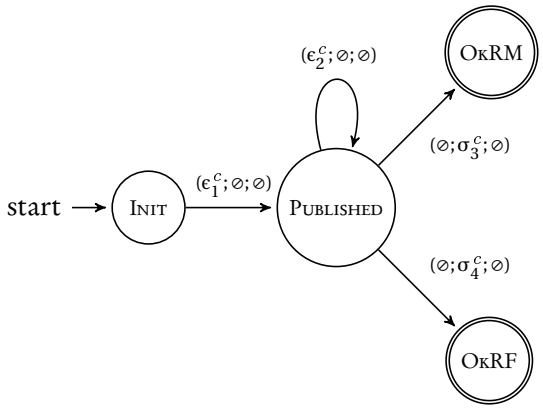


Figure 6.4 – State machine of the coordinator

Q_c	$= \{\text{INIT}, \text{PUBLISHED}, \text{OKRM}, \text{OKRF}\}$
Σ_c	$= \{(\epsilon_1^c;\varnothing;\varnothing), (\epsilon_2^c;\varnothing;\varnothing), (\varnothing;\sigma_3^c;\varnothing), (\varnothing;\sigma_4^c;\varnothing)\}$
δ_c	$= \begin{cases} \text{INIT} \times (\epsilon_1^c;\varnothing;\varnothing) \longmapsto \text{PUBLISHED} \\ \text{PUBLISHED} \times (\epsilon_2^c;\varnothing;\varnothing) \longmapsto \text{PUBLISHED} \\ \text{PUBLISHED} \times (\varnothing;\sigma_3^c;\varnothing) \longmapsto \text{OKRM} \\ \text{PUBLISHED} \times (\varnothing;\sigma_4^c;\varnothing) \longmapsto \text{OKRF} \end{cases}$
q_{0_c}	$= \{\text{INIT}\}$
F_c	$= \{\text{OKRM}, \text{OKRF}\}$
ϵ_1^c	$= publish?$
ϵ_2^c	$= askRM? \vee askRF?$
σ_3^c	$= \text{ValidTransfer}(Proof_{lock})$
σ_4^c	$= \text{AbortTransfer}()$

Table 6.2 – Elements of the coordinator

Sources. The role of the source is to transfer assets to recipients. A source is represented by Figure 6.5 and defined by the elements in Table 6.3. Let us introduce the four predicates of the source's protocol.

- CorrectSwap: it takes the *proof-of-action* $Proof_{publish}$ as a parameter that proves the swap graph publication to the coordinator. To be valid, the predicate must satisfy the following two

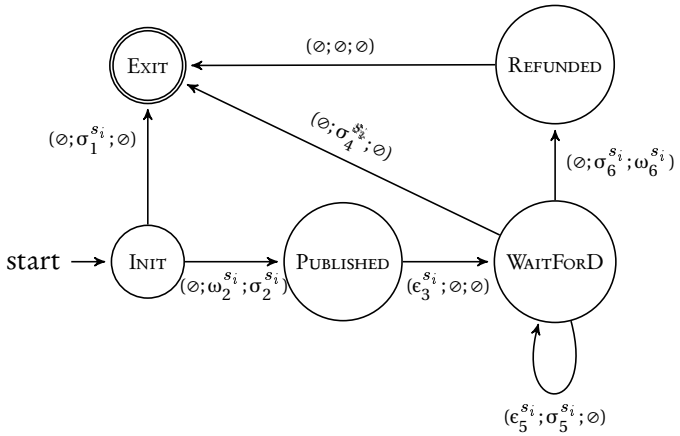


Figure 6.5 – State machine of a source s_i

$$
\begin{aligned}
Q_{s_i} &= \{\textsc{Init}, \textsc{Published}, \textsc{WaitForD}, \textsc{Refunded}, \textsc{Exit}\}\\
\Sigma_{s_i} &= \{(\varnothing;\sigma_1^{s_i};\varnothing),(\varnothing;\sigma_2^{s_i};\omega_2^{s_i}),(\epsilon_3^{s_i};\varnothing;\varnothing),(\varnothing;\sigma_4^{s_i};\varnothing),\\
&\quad (\epsilon_5^{s_i};\sigma_5^{s_i};\varnothing),(\varnothing;\sigma_6^{s_i};\omega_6^{s_i}),(\varnothing;\varnothing;\varnothing)\}\\
\delta_{s_i} &= \begin{array}{l}
\textsc{Init} \times (\varnothing;\sigma_1^{s_i};\varnothing) \longmapsto \textsc{Exit}\\
\textsc{Init} \times (\varnothing;\sigma_2^{s_i};\omega_2^{s_i}) \longmapsto \textsc{Published}\\
\textsc{Published} \times (\epsilon_3^{s_i};\varnothing;\varnothing) \longmapsto \textsc{WaitForD}\\
\textsc{WaitForD} \times (\epsilon_5^{s_i};\sigma_5^{s_i};\varnothing) \longmapsto \textsc{WaitForD}\\
\textsc{WaitForD} \times (\varnothing;\sigma_6^{s_i};\omega_6^{s_i}) \longmapsto \textsc{Refunded}\\
\textsc{WaitForD} \times (\varnothing;\sigma_4^{s_i};\varnothing) \longmapsto \textsc{Exit}\\
\textsc{Refunded} \times (\varnothing;\varnothing;\varnothing) \longmapsto \textsc{Exit}
\end{array}\\
q_{0_p} &= \{\textsc{Init}\}\\
F_p &= \{\textsc{Exit}\}\\
\sigma_1^{s_i} &= \neg\ \mathsf{CorrectSwap}(\mathit{Proof}_{\mathit{publish}})\\
\sigma_2^{s_i} &= \mathsf{CorrectSwap}(\mathit{Proof}_{\mathit{publish}})\\
\sigma_4^{s_i} &= \mathsf{AuthoRM}()\\
\sigma_5^{s_i} &= \mathsf{NoDecision}()\\
\sigma_6^{s_i} &= \mathsf{AuthoRF}()\\
\epsilon_3^{s_i} &= askRM(\mathit{Proof}_{\mathit{lock}})!\\
\epsilon_5^{s_i} &= askRF!\\
\omega_2^{s_i} &= LockAsset(a_i, r_i)\\
\omega_6^{s_i} &= RecoveringAsset(a_i, \mathit{Proof}_{\mathit{refund}})
\end{aligned}
$$

Table 6.3 – Elements of the source s_i

conjunctions; (1) the source's local copy of the graph and the graph located in $\mathit{Proof}_{\mathit{publish}}$ are identical, and (2) the source's local timeout is not reached.

- NoDecision: it is valid if the coordinator has not yet decided after the source's timeout.
- AuthoRM: it is true when the coordinator state machine is in "OkRM" state.
- AuthoRF: it is true when the coordinator is in "OkRF" state.

The source starts by checking the status of the graph. If the graph does not satisfy CorrectSwap, i.e. an invalid $\mathit{Proof}_{\mathit{publish}}$ or a reaching timeout, it exits the swap ($\sigma_1^{s_i}$). Otherwise ($\sigma_2^{s_i}$), it computes the $\omega_2^{s_i}$ operation, locks its asset a_i and assigns the new owner r_i. The source needs the *proof-of-action* to ensure that the swap graph is correct before locking its asset. Consequently, $\epsilon_1^{a_i}$ from Figure 6.2 of its asset is triggered. Then, the source sends a request message to the coordinator to give a *redeem* decision through the $\epsilon_3^{s_i}$ action. The source adds a proof ($\mathit{Proof}_{\mathit{lock}}$) for certifying that the locked asset operation has been executed properly. Hence, this step allows the coordinator to assess the validity of the lock operation executed by the source.
Depending on the coordinator's decision, either the source exits the swap if $\sigma_4^{s_i}$ is satisfied, or the source recovers its asset if $\sigma_6^{s_i}$ is satisfied. In that case, the source computes the $\omega_6^{s_i}$ operation to recover its asset. The source needs the *proof-of-action* $\mathit{Proof}_{\mathit{refund}}$ certifying that the coordinator has given a *refund* decision to satisfy $\epsilon_2^{a_i}$ from Figure 6.2 of its asset. However, if no decision has been made after the source's timeout, $\sigma_5^{s_i}$ is set to true, the source asks for a *refund* decision by sending a request message to the coordinator through $\epsilon_5^{s_i}$ action.
It is essential to clarify that Figure 6.5 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

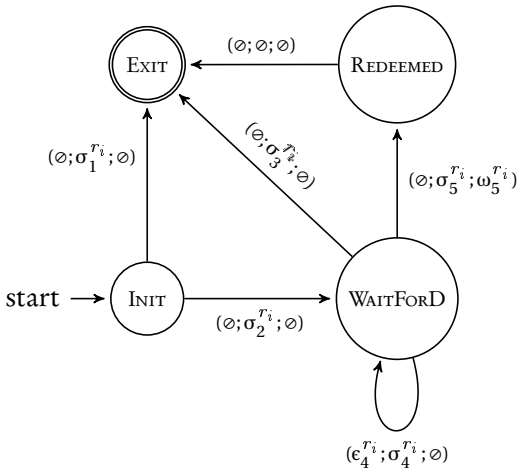


Figure 6.6 – State machine of a recipient r_i

$$
\begin{aligned}
Q_r &= \{\text{INIT}, \text{WAITFORD}, \text{REDEEMED}, \text{EXIT}\} \\
\Sigma_r &= \{(\varnothing;\sigma_1^{r_i};\varnothing),(\varnothing;\sigma_2^{r_i};\varnothing),(\epsilon_4^{r_i};\sigma_4^{r_i};\varnothing),(\varnothing;\sigma_3^{r_i};\varnothing), \\
&\qquad (\varnothing;\sigma_5^{r_i};\omega_5^{r_i}),(\varnothing;\varnothing;\varnothing)\} \\
\delta_r &= \left|\begin{array}{l}
\text{INIT} \times (\varnothing;\sigma_1^{r_i};\varnothing) \longmapsto \text{EXIT} \\
\text{INIT} \times (\varnothing;\sigma_2^{r_i};\varnothing) \longmapsto \text{WAITFORD} \\
\text{WAITFORD} \times (\epsilon_4^{r_i};\sigma_4^{r_i};\varnothing) \longmapsto \text{WAITFORD} \\
\text{WAITFORD} \times (\varnothing;\sigma_3^{r_i};\varnothing) \longmapsto \text{EXIT} \\
\text{WAITFORD} \times (\varnothing;\sigma_5^{r_i};\omega_5^{r_i}) \longmapsto \text{REDEEMED} \\
\text{REDEEMED} \times (\varnothing;\varnothing;\varnothing) \longmapsto \text{EXIT}
\end{array}\right. \\
q_{0_r} &= \{\text{INIT}\} \\
F_r &= \{\text{EXIT}\} \\
\sigma_1^{r_i} &= \neg\ \mathsf{CorrectSwap}(\mathit{Proof}_{\mathit{publish}}) \\
\sigma_2^{r_i} &= \mathsf{CorrectSwap}(\mathit{Proof}_{\mathit{publish}}) \\
\sigma_3^{r_i} &= \mathsf{AuthoRF}() \\
\sigma_4^{r_i} &= \mathsf{NoDecision}() \\
\sigma_5^{r_i} &= \mathsf{AuthoRM}() \\
\epsilon_4^{r_i} &= \mathit{askRF}! \\
\omega_5^{r_i} &= \mathit{RetrievingAsset}(a_i, \mathit{Proof}_{\mathit{redeem}})
\end{aligned}
$$

Table 6.4 – Elements of the recipient r_i

loss of generalisation. Thereby, to help understand the protocol and afterwards help the formal proof, a source transferring multiple assets will have different identification for each transfer asset. If we take the example of Figure 6.1, A as a source will have the following identification: $\{s_{A_1}, s_{A_3}\}$.

Recipients. The recipient, represented in Figure 6.6 and defined by the elements in Table 6.4, is the asset's new owner. The predicates enumerated previously, CorrectSwap, NoDecision, AuthoRM and AuthoRF, have the same definition for recipients. Like the source, the recipient must run the protocol in Figure 6.6 for each asset it receives for the same reason defined above. For example, the participant B from Figure 6.1 as a recipient will be represented by $\{r_{B_1}, r_{B_4}\}$ and the protocol for each one is the following: the recipient starts by checking the status of the graph using the *proof-of-action* $\mathit{Proof}_{\mathit{publish}}$ ($\sigma_1^{r_i}$ and $\sigma_2^{r_i}$). Depending on the coordinator's decision, either the recipient exits the swap if $\sigma_3^{r_i}$ is true, or the recipient retrieves its asset if $\sigma_5^{r_i}$ is true. To retrieve its asset, the recipient computes the $\omega_5^{r_i}$ operation. The recipient adds to the operation the proof $\mathit{Proof}_{\mathit{redeem}}$ that the coordinator has given a *redeem* decision. Consequently, this triggers $\epsilon_3^{a_i}$ from Figure 6.2 of its asset, and the recipient becomes the new owner. However, if $\sigma_4^{r_i}$ is satisfied, the recipient asks for a *refund* decision through $\epsilon_4^{r_i}$ action.

6.4 Description of the Protocol Based on *Proof-of-Actions*

In this part, we describe in detail the different phases of the protocol and how *proof-of-actions* allow countering the unacceptable behaviours of the Byzantine participants that we want to prove. The protocol $\mathscr{P}_{swap}$ runs through three phases, where the validation of a *proof-of-action* conditions each phase:

Phase 1: proof of graph publication. In phase 1, participants designate a publisher to publish the swap graph to the coordinator. Each correct participant, i.e. sources and recipients, waits for a *proof-of-action*, "$\mathit{Proof}_{\mathit{publish}}$", from the coordinator that the graph has been published. Since all information about the coordinator is public, participants can retrieve the $\mathit{Proof}_{\mathit{publish}}$ information

and verify its validity. The correct participants extract from the proof the graph published by the publisher. If their local graph and the published one are identical, the proof is valid. The coordinator being public helps prevent misbehaviour from the publisher. If "$Proof_{publish}$" is invalid or the graph has not been published after the timeout of one of the correct participants, then CorrectSwap is violated, and correct participants will abandon the swap.
Besides, imagine that a Byzantine publisher decides to publish the swap graph simultaneously with the one decided by the participants. Two swap graphs will be published to the coordinator. However, the correct participants will ignore the graph published by the Byzantine publisher because the publisher's identifier (i.e. the Byzantine one) will not match the one chosen before the swap.

Phase 2: proof of locking assets. During phase 2, sources lock their assets. The phase starts with assuming a valid $Proof_{publish}$. 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. The locking operation assigns the asset's new owner, 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 the proof $Proof_{lock}$ that the source has successfully computed $LockAsset$. All sources must send a request message accompanied by $Proof_{lock}$ for each transferred asset; otherwise, the swap cannot occur. The coordinator collects all proofs through the $askRM(Proof_{lock})$! action of all sources and checks their validity. If one proof is invalid, the coordinator aborts the swap by giving a *refund* decision.
To give a *redeem* decision, the conditions of the predicate ValidTransfer($Proof_{lock}$) 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 instance, 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 a decision. Consequently, if the coordinator gives a *redeem* decision by updating its state to "OkRM", the changing state satisfies the predicate AuthoRM(). Therefore, correct recipients retrieve the proof "$Proof_{redeem}$" from the coordinator and can redeem their assets using the *proof-of-action* $Proof_{redeem}$. Conversely, if the coordinator changes its state to "OkRF", the predicate AuthoRF() is satisfied. Therefore, correct sources retrieve from the coordinator the proof "$Proof_{refund}$" to be refunded. The two *proof-of-actions* are the only way to unlock these assets.

6.5 $\mathscr{P}_{swap}$ Implementation in TLA$^+$

The previous section formally described the protocol based on state machines. In this section, we model the *cross-chain swap* protocol, $\mathscr{P}_{swap}$, in a language dedicated to the specification of distributed systems, TLA$^+$. This modelling aims to apply the verification methods that the TLA$^+$ tool provides. This verification step is done in the next chapter. The *Two-Phase Commit* strongly inspires the $\mathscr{P}_{swap}$ protocol; therefore, the modelling method will use the same approach described in Section 4.3.3.

6.5.1 Module, Declarations and Definitions

The module of $\mathscr{P}_{swap}$ starts by its name, which is MODULE $CrossChainSwap$, and it extends the modules: $Integers$ for the participants' identifiers and arithmetic operations like {+, -, *}, and $TLAPS$ needed for the verification proof. The module's body is a sequence of statements, where

a statement is a declaration, a definition, an assumption or a theorem. Declarations in the body of MODULE $CrossChainSwap$ are: CONSTANT $NTxs, Correct, Timeout$. $NTxs$ is the number of transactions corresponding to the number of traded assets. $Correct$ is the set of correct participants involved in the swap. $Timeout$ is a boolean that models the synchrony between the participants. $Timeout$ set to TRUE means that the system is assumed to be asynchronous, and participants can timeout.

Set definitions. The modelling of the protocol is based on states; thus, we define the set of states of each component. As defined in Section 6.3.1, an asset has four possible states represented in $AStates$ (see **Definition** 19), with "OwS" the state that corresponds to the asset owned by its source and "OwR" owned by its recipient. The state "locked" reflects the locking asset, and the state "other" is any state not recognised by the swap. The coordinator and publisher states are defined in Tables 6.2 and 6.1; thereby, their set of states are formalised in TLA$^+$ as $CStates$ and $PStates$.
As described in the protocol, the state of the swap graph affects the protocol; thus, we define a set of possible states of the swap graph in $SwapStates$. At the initial state, the swap graph is in "init" state. Depending on the publisher's behaviour, the swap graph can be "correct", i.e. identical to the participants' local graph or "different", i.e. different from the participants' local graph.

Although sources and recipients were identified by their state in the state machine protocol description, in the TLA$^+$ modelling, we did not define their set of states. The approach does not define a set of states but instead uses the labels of the source's and recipient's PlusCal code. Recall that a PlusCal code defines labels in the code to define atomic actions. When the PlusCal translator generates the TLA$^+$ specification code, the program control is created, and each label corresponds to a TLA$^+$ action. Therefore, if the labels are correctly defined, one can identify the program's current state using the program control. For example, we define a label named $WaitForD$ in the source's PlusCal code. If the source's program control p is set to $WaitForD$, that means the source's state corresponds to "WAITFORD" of Figure 6.5. Thus, tracking the state of the source's (and recipient's) program control is equivalent to monitoring the source's (and recipient's) state.

Definition 19 (Set of states)**.**

$AStates \triangleq$ {"OwS", "OwR", "locked", "other"}
$CStates \triangleq$ {"init", "published", "okRM", "okRF"}
$PStates \triangleq$ {"init", "publish"}
$SwapStates \triangleq$ {"init", "correct", "different"}

Identification. We give an identifier to each participant and asset to track their state. We attribute to the coordinator and the publisher, respectively, the values 0 and -1. However, we have a parametric number of sources, recipients and assets, and we must assign a unique identifier to each. Note that assets are not processes but must have an identifier to track their state. The identifiers must be calculated according to the number of traded assets, $NTxs$. As defined in Section 6.1.2, a swap is modelled as a directed graph $S = (\Pi, E)$, where vertices are participants and edges are transferred assets. If a source transfers more than one asset in the swap, it will have as many identifiers as assets to transfer. The same applies to recipients that receive more than one asset and will have as many identifiers as assets received.
$NTxs$ is the number of transactions that correspond to the number of asset transfers. Each transaction has a source and a recipient; thus, we can define a relation between these three components to assign an identifier. The identifiers 0 and -1 have already been assigned and are no longer available; thus, the identification must start from 1. The intuitive relation is that "*1 transfers 2 to 3*" with 1 the source's identifier, 2 the asset's identifier and 3 the recipient's identifier. This relation gives the following formulas to calculate the identifier set of sources Π_s, assets Λ and recipients Π_r:

$$\Pi_s = \sum_{x=1}^{NTxs} 3x-2 \qquad \Pi_r = \sum_{x=1}^{NTxs} 3x \qquad \Lambda = \sum_{x=1}^{NTxs} 3x-1$$

$$E = \sum_{x=1}^{NTxs} (3x-2,\ 3x-1,\ 3x)$$

In the TLA$^+$ formalism, the formulas are defined as follows:

$$\begin{array}{ll} Sources & \triangleq \{3*x-2 \quad : x \in 1..NTxs\} \\ Assets & \triangleq \{3*x-1 \quad : x \in 1..NTxs\} \\ Recipients & \triangleq \{3 \quad *x : x \in 1..NTxs\} \end{array}$$

Moreover, we define a set of predicates that return the asset belonging to the argument given as a parameter. Since each asset is linked to its source and its recipient, from the identifier of a source, we can have that of its asset and the asset's recipient. Conversely, one can know the identifier of a source and the recipient from that of the asset. The predicates are $AofS$, $AofR$, $SofA$, and $RofA$. Therefore, we can have the identifier of an asset from its source with $AofS$ and its recipient with $AofR$. The set of predicates is as follows:

$$\begin{array}{l} AofS(x) \triangleq x+1 \\ AofR(x) \triangleq x-1 \\ SofA(x) \triangleq x-1 \\ RofA(x) \triangleq x+1 \end{array}$$

For example, suppose a source with the identifier 1 and a recipient with an identifier 6. Their asset are respectively:

$$\begin{array}{l} AofS(1) \triangleq 1+1=2 \\ AofR(6) \triangleq 6-1=5 \end{array}$$

Set of participants. The system consists of correct and Byzantine participants. We introduce the sets defined in **Definition** 20 to allocate actions to each participant. We define Pi as the union set of sources and recipients and Pc as the set of correct participants. Remember that $Correct$ is a constant that designates the set of correct participants. We define both sets of correct sources and recipients, respectively, in $CSources$ and $CRecipients$. Finally, we define the set of Byzantine sources and recipients in $BSources$ and $BRecipients$. The Byzantine participants' sets are the set of sources (respectively recipients) excluded from the set of correct sources (respectively recipients).

Definition 20 (Set of Participants)**.**

$$\begin{array}{l} Pi \triangleq Sources \cup Recipients \\ Pc \triangleq Pi \cap Correct \\ CSources \triangleq Pc \cap Sources \\ CRecipients \triangleq Pc \cap Recipients \\ BSources \triangleq Sources \setminus CSources \\ BRecipients \triangleq Recipients \setminus CRecipients \end{array}$$

6.5.2 The *Cross-Chain Swap* Algorithm in PlusCal

The algorithm is written in PlusCal code and then translated into TLA$^+$ language for the proof. As for the *Two-Phase Commit* example, we first define a set of variables needed for the algorithm. $assets$, $pState$, $swapGraph$ and $coordState$ are the variables that represent the state of, respectively, assets, the publisher, the swap graph and the coordinator. In the initial state, assets are owned by their source "OwS", and the publisher, the coordinator and the swap graph are set to "init". In the protocol, the participants can request a *redeem* or a *refund* decision from the coordinator. To model these actions, we define the variable qrm, a sequence of sources' identifiers that has requested a *redeem* decision. Similarly, the *refund* request is defined by the variable qrf, a sequence of participants' identifier that has requested a *refund* decision. Finally, all the *proof-of-actions* are modelled as boolean variables. When the value of the variable is TRUE, that means the proof is valid. The proof of graph publication, the proof of decision *redeem*, and the proof of decision *refund* are respectively $ProofPublish$, $ProofOkRM$ and $ProofOkRF$. The proof of locking asset variable, $ProofLock$, is a function that maps for each source a boolean value. Therefore, if the value of the $ProofLock$ function at index i is TRUE, that means the source i has provided a valid $Proof_{lock}$.

In the protocol defined in the previous section, we defined predicates that conditioned the evolution of the system, namely CorrectSwap, ValidTransfer, and AbortTransfer. In PlusCal, we define the predicate ValidTransfer, and AbortTransfer in the **define** statement. The predicate that makes it possible for the coordinator to give a *redeem* decision is $ValidTransfer$. The predicate that must be valid for the coordinator to give a *refund* decision is $AbortTransfer$. For getting a *redeem* decision, the sequence qrm must contain all the elements of the set $Sources$, and the variable $ProofLock$ must have all its elements to TRUE. A *refund* decision is given when the sequence qrf contains at least one element. Both are defined as follows:

$$ValidTransfer \triangleq qrm = Sources \land \forall\, s \in Sources : ProofLock[s] = \text{TRUE}$$
$$AbortTransfer \triangleq qrf \neq \{\}$$

The predicate CorrectSwap is valid if the swap graph's state equals $swapGraph =$ "correct", and $Timeout$ is set to FALSE.

Functions and Predicates

In Section 6.3, 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** function. Note that the parameters of the functions may vary from Section 6.3 because of the TLA$^+$ language. The following functions, written in PlusCal, are actions and operations of participants:

- The source's operation $LockAsset$:

```
macro lockAsset( self ) {
 if ( ProofPublish = TRUE ∧ self ∈ Sources ∧ assets[AofS(self)] = "OwS" )
  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 the hashtable that maps an asset with its state. A source can lock its asset only if it owns it.

- The $askRM$ and $askRF$ actions:

macro $askRM($ $self$ $)$ **{**
if ($self \in Sources \wedge ProofLock[self] = \text{TRUE} \wedge coordState =$ "published" **)**
$qrm := qrm \cup \{self\}$**; }**

macro $askRF($ $self$ $)$ **{**
if ($coordState =$ "published" **)** $qrf := qrf \cup \{self\}$**; }**

The $askRM$ function can be executed only by sources; $self \in Sources$. The $askRM$ function modelled in TLA$^+$ does not contain the lock *proof-of-action* of the source as a parameter. The modelling of *proof-of-action* is defined as global system variables since they represent the coordinator's public information. Therefore, it is not necessary to add the lock *proof-of-action* as a parameter to the function.

- Below is the $RetrievingAsset$ and $RecoveringAsset$ operations:

macro $retrievingAsset($ $self$ $)$ **{**
if ($self \in Recipients \wedge ProofOkRM = \text{TRUE} \wedge assets[AofR(self)] =$ "locked" **)**
$assets[AofR(self)] :=$ "OwR"**; }**

macro $recoveringAsset($ $self$ $)$ **{**
if ($self \in Sources \wedge ProofOkRF = \text{TRUE} \wedge assets[AofS(self)] =$ "locked" **)**
$assets[AofS(self)] :=$ "OwS"**; }**

The first function can be executed only by recipients, while the second is executed only by sources. Similarly to the action $askRM$, the functions do not require the *proof-of-action* as a parameter of the function. $ProofOkRM$ and $ProofOkRF$ are global variables.

- In the following, we describe additional actions specific to Byzantine participants:

macro $otherS($ $self$ $)$ **{**
if ($self \in Sources \wedge assets[AofS(self)] =$ "OwS" **)**
$assets[AofS(self)] :=$ "other"**; }**

macro $otherR($ $self$ $)$ **{**
if ($self \in Recipients \wedge assets[AofR(self)] =$ "OwR" **)**
$assets[AofR(self)] :=$ "other"**; }**

macro $directToR($ $self$ $)$ **{**
if ($self \in Sources \wedge assets[AofS(self)] =$ "OwS" **)**
$assets[AofS(self)] :=$ "OwR"**; }**

macro $directToS($ $self$ $)$ **{**
if ($self \in Recipients \wedge assets[AofR(self)] =$ "OwR" **)**
$assets[AofR(self)] :=$ "OwS"**; }**

$otherS$ (respectively $otherR$) is a function only executed by a Byzantine source (respectively recipient) that executes the illegal action $\epsilon_6^{a_i}$ (respectively $\epsilon_7^{a_i}$) from Figure 6.2. $directToR$ (respectively $directToS$) is a function only executed by a Byzantine recipient (respectively source) that executes the illegal action $\epsilon_4^{a_i}$ (respectively $\epsilon_5^{a_i}$).

Processes

The processes are modelled according to the state machines of the participants. Thus, a process state with several outgoing arcs will be modelled by **either -or** statement. We define the following processes: (1) the publisher, (2) the coordinator, (3) correct sources, (4) Byzantine sources, (5) correct recipients and (6) Byzantine recipients. The description of their protocol is in the following.

The publisher. The *Publisher* has -1 as an identifier ($PublisherID = -1$). The publisher has one non-deterministic possible action, "init_p" (the only label defined in the code below). The publisher either (1) publishes the swap graph by changing its state to "publish", or (2) the publisher does not publish the graph and exists the swap (**skip**), i.e. acting like a Byzantine publisher. Suppose the publisher publishes the graph, a second level of non-deterministic behaviour is defined. Hence, the graph can be either correct or different depending on the publisher's correctness. The publisher is not defined as a **fair process** (see Section 4.3.2). Therefore, even if an action is enabled, it can halt and stay in "init_p" forever and stutters. These steps of stuttering can model the crash of the publisher. Note that the process describes both a correct and a Byzantine publisher making it different from Figure 6.3.

The publisher's PlusCal code is the following:

```
process ( Publisher = PublisherID )
{
      init_p : either {
                  pState := "publish";
                  either swapGraph := "correct";
                  or swapGraph := "different"; }
               or skip;
} ;
```

The coordinator. The coordinator is identified by 0 ($CoordinatorID = 0$) and is defined by four possible actions to execute (represented by the labels of the code below). We add a fairness condition that the process cannot stop at a non-blocking action.

- *init_c*: the action is conditioned by the **await** construct. The coordinator has to wait until the *Publisher* publishes the graph. When the graph is published, the coordinator can update its state, and the proof of publication *ProofPublish* is set to TRUE.
- *decision*: the next action is a non-deterministic **either** – **or** construct, in which each branch refers to the two possible coordinator's decisions. Each branch is conditioned by the **await** construct. Consequently, the coordinator takes the **either** branch if *ValidTransfer* is valid or the **or** branch if *AbortTransfer* is valid.
- *decisionValid*: the action can be executed if the predicate *ValidTransfer* has been validated. Such a result means that all sources have a valid *ProofLock*. The coordinator updates its state to "okRM", and the *ProofOkRM* is set to TRUE. Therefore, correct recipients will be able to retrieve their assets using this information. The execution of *decisionValid* leads the coordinator to the end of its program, i.e. **goto** *Done*.
- *decisionAbort*: the action can be executed only if the predicate *AbortTransfer* is satisfied. The coordinator updates its state to "okRF", and the *ProofOkRF* is set to TRUE. Therefore, correct sources will be able to recover their assets using *ProofOkRF*. Similarly to the previous action, the execution of *decisionAbort* marks the end of the coordinator's program.

The coordinator's PlusCal code is the following:

```
fair  process ( Coordinator = CoordinatorID )
 {
 init_c:        await pState = "publish" ∧ swapGraph ≠ "init";
                coordState := "published";
                ProofPublish := TRUE;
 decision:      either {
                        await ValidTransfer;
 decisionValid:         coordState := "okRM";
                        ProofOkRM := TRUE;
                        goto Done; }
                or {
                        await AbortTransfer;
 decisionAbort:         coordState := "okRF";
                        ProofOkRF := TRUE;
                        goto Done; } ;
 } ;
```

The source. *Source* defined in the code below is a multiprocess of *CSources* processes (correct sources). As mentioned earlier, the state of a source is described according to its labels. For each state of Figure 6.5, a label describes the action to be executed corresponding to the actions of the outgoing edges of the state machine. Therefore, we have the following four actions:

- *init_src*: from Figure 6.5, the initial state of a source has two outgoing edges. Consequently, the initial action of the source's process describes a non-deterministic behaviour of two possible behaviours. The **either** branch corresponds to the violation of the predicate CorrectSwap, and the **or** branch to its validation. If the predicate is violated, the source leaves the swap; otherwise, it goes to the next action *lock*.

- *lock*: correct sources can lock their asset if the swap has been correctly published. The source executes the action *lock*; thus, it sets to TRUE its *ProofLock proof-of-action*. After locking its asset, the source can go to the next action *published*.

- *published*: the action refers to asking for a *redeem* decision from the coordinator. The function can only be executed if the *ProofLock* of *self* is valid (i.e. set to TRUE).

- *waitForD*: in Figure 6.5, the state "WAITFORD" has three outgoing edges. This state describes when the source waits for the coordinator's decision. Consequently, the action *waitForD* describes a non-deterministic action of three branches. The construct **await** conditions each branch. The first branch is when the coordinator gives its approval to redeem, in which case the correct source can leave the swap. The *Done* label corresponds to the EXIT state of the source's state machine. If the coordinator gives its authorisation to refund the assets, i.e. *ProofOkRF* = TRUE, the correct source can execute the *recoveringAsset* function, recover its asset, and leave the swap. The third branch is the case where NoDecision is validated because the correct source has reached its timeout (*Timeout* = TRUE). The correct source can request a *refund* from the coordinator, then go back to the label *waitForD* and wait once again for a decision from the coordinator.

The source's Pluscal code is the following:

```
fair process ( Source ∈ CSources )
{
  init_src : either {
               await swapGraph = "different" ∨ Timeout = TRUE ;
               goto Done ; }
             or {
               await ProofPublish = TRUE ∧ swapGraph = "correct";
  lock:      lockAsset(self) ;
  published: askRM(self) ;
  waitForD:  either {
               await ProofOkRM = TRUE ;
               goto Done ; }
             or {
               await ProofOkRF = TRUE ;
               recoveringAsset(self) ;
               goto Done ; }
             or {
               await coordState = "published" ∧ Timeout = TRUE ;
               askRF(self) ;
               goto waitForD ; } ;
              } ;
} ;
```

The recipient. *Recipient* is a multiprocess of *CRecipients* processes (correct recipients). The recipient has two possible actions, including the *Done* action, which characterises the termination of the process. The recipient starts with the *init_rcp* action. The action evaluates whether the swap is correct or different. In the case of a different swap, the recipient exists the swap. However, if the swap is published and correct, the recipient executes the action *waitForD_rcp*. The recipient waits for the coordinator's decision and exists the swap if the decision is *refund* or retrieves its asset if the decision is *redeem*. The recipient can ask for a *refund* in the case where *Timeout* = TRUE.

```
fair process ( Recipient ∈ CRecipients )
{
  init_rcp :     either {
                   await swapGraph = "different" ∨ Timeout = TRUE ;
                   goto Done ; }
                 or {
                   await ProofPublish = TRUE ∧ swapGraph = "correct";
  waitForD_rcp:  either {
                     await ProofOkRF = TRUE ;
                     goto Done ; }
                  or {
                     await ProofOkRM = TRUE ;
                     retrievingAsset(self) ;
                     goto Done ; }
                  or {
                     await coordState = "published" ∧ Timeout = TRUE ;
                     askRF(self) ;
                     goto waitForD_rcp ; } ; } ;
} ;
```

Byzantine Participants Models

In TLA$^+$, we model Byzantine participants as unpredictable participants. Hence, we use the 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 and additional actions defined in Section 6.5.2.

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

$BSource$ is the process name, and $BSources$ is the set of Byzantine sources. After each action execution, the process returns to the initial state and non-deterministically executes another action.

The following code is the process of a Byzantine recipient. Similarly to the Byzantine sources, a Byzantine recipient can execute actions of the correct recipient and actions specific to Byzantine recipients.

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

$BRecipient$ is the process name, and $BRecipients$ is the set of Byzantine recipients. Infinitely, the Byzantine performs the actions defined in its program in totally random order. After each execution, it returns to its initial state. A label represents each action of the two processes; for example, the action $otherR(self)$ of the Byzantine recipient has the label $BRother$. Consequently, $\{init\_brcp, BRretrievingAsset, BRother, BRdirectToS, BRaskRF\}$ are the Byzantine recipient's labels, and $\{BrecoveringAsset, init\_bsrc, BdirectToR, BaskRM, BlockAsset, BSaskRF, Bother\}$ are the Byzantine source's labels.

As a result, a Byzantine participant may execute any branch of its code or do nothing, acting like a crashed participant since there is no **fair** keyword. As a recall, the publisher can be Byzantine, and its possible actions are either publishing a wrong graph or doing nothing. The swap does not occur in both cases if at least one correct participant detects the behaviour of a Byzantine participant.

6.5.3 TLA$^+$ Translation

As described in the example with the *Two-Phase Commit*, after describing the PlusCal algorithm of each participant, we translate the code into the TLA$^+$ language. The result is a set of predicates

and actions that may or may not be enabled. Once we translate the PlusCal code, we obtain the following complete system specification:

$$
\begin{aligned}
&vars \triangleq \langle assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish, ProofLock,\\
&\qquad ProofOkRM, ProofOkRF, pc\rangle\\
&ProcSet \triangleq \{PublisherID\} \cup \{CoordinatorID\} \cup (CSources) \cup (BSources) \cup\\
&\qquad (CRecipients) \cup (BRecipients)\\
&Init \triangleq \land assets = [a \in Assets \mapsto \text{“OwS”}]\\
&\qquad \land pState = \text{“init”}\\
&\qquad \land coordState = \text{“init”}\\
&\qquad \land qrm = \{\}\\
&\qquad \land qrf = \{\}\\
&\qquad \land swapGraph = \text{“init”}\\
&\qquad \land ProofPublish = \text{FALSE}\\
&\qquad \land ProofLock \quad = [c \in Sources \mapsto \text{FALSE}]\\
&\qquad \land ProofOkRM = \text{FALSE}\\
&\qquad \land ProofOkRF \ = \text{FALSE}\\
&\qquad \land pc = [self \in ProcSet \mapsto \text{CASE } self = PublisherID \to \text{“init\_p”}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \square \ self = CoordinatorID \to \text{“init\_c”}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \square \ self \in CSources \to \text{“init\_src”}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \square \ self \in BSources \to \text{“init\_bsrc”}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \square \ self \in CRecipients \to \text{“init\_rcp”}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \square \ self \in BRecipients \to \text{“init\_brcp”}]\\
&Next \triangleq Publisher \lor Coordinator\\
&\qquad \lor (\exists self \in CSources : Source(self))\\
&\qquad \lor (\exists self \in BSources : BSource(self))\\
&\qquad \lor (\exists self \in CRecipients : Recipient(self))\\
&\qquad \lor (\exists self \in BRecipients : BRecipient(self))\\
&Spec \triangleq \land Init \land \square[Next]_{vars}\\
&\qquad \land \mathrm{WF}_{vars}(Next)\\
&\qquad \land \mathrm{WF}_{vars}(Coordinator)\\
&\qquad \land \forall self \in CSources : \mathrm{WF}_{vars}(Source(self))\\
&\qquad \land \forall self \in CRecipients : \mathrm{WF}_{vars}(Recipient(self))
\end{aligned}
$$

$vars$ is the variables that make up the system and update their states according to the system evolution. The translation creates a new variable, the program control variable pc. The pc initialisation gives the start of each process of the system. In addition, the translation of the algorithm creates $ProcSet$, which corresponds to the set of identifiers of the system participants.

The predicate $Init$ is the initial state of the system's variables, and the predicate $Next$ is the possible participant action from the set of all participants. When the system runs, one action is executed at a time. The predicate $Spec$ defines the algorithm specification. Because we define the coordinator, the correct sources and the correct recipients as fair processes, the specification formula adds a *weak fairness* requirement WF_{vars}. Moreover, the requirement to the $Next$ predicate implies that infinitely many $Next$ steps must occur.

The publisher action. The publisher has only one action defined by: $Publisher \triangleq init\_p$, and it can possibly execute it or not. The action, defined in **Definition** 21, corresponds to the label defined

in the PlusCal code. If the publisher is Byzantine, it can decide to do nothing, and its program control pc remains in $init\_p$ without executing it. In case it executes its action, the program control of the publisher must be in the initial state. Once the action is executed, the program control's next state (characterised by the quote) is "Done". The keyword UNCHANGED informs the variables that have not been changed during the execution of the action.

Definition 21 (The publisher "initial" action)**.**

$$
\begin{aligned}
init\_p \triangleq\ & \land pc[PublisherID] = \text{"init\_p"} \\
& \land \lor \land pState' = \text{"publish"} \\
& \quad \land \lor \land swapGraph' = \text{"correct"} \\
& \quad\quad \lor \land swapGraph' = \text{"different"} \\
& \quad \lor \land \text{TRUE} \\
& \quad\quad \land \text{UNCHANGED } \langle pState, swapGraph \rangle \\
& \land pc' = [pc \text{ EXCEPT } ![PublisherID] = \text{"Done"}] \\
& \land \text{UNCHANGED } \langle assets, coordState, qrm, qrf, ProofPublish, ProofLock, \\
& \qquad\qquad ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The coordinator actions. Formally, the coordinator is defined by the disjunction of four possible actions as follows:

$$Coordinator \triangleq init\_c \lor decision \lor decisionValid \lor decisionAbort$$

The four actions represent the four labels of the PlusCal code. Each action modifies a set of variables that evolves the system. The action $init\_c$, defined in **Definition** 22, describes the acknowledgement of the graph publication and updates the coordinator's next state and the $ProofPublish$'s next state. The next state of the program control is the next action to execute, which is "decision".

Definition 22 (the coordinator "initial" action)**.**

$$
\begin{aligned}
init\_c \triangleq\ & \land pc[CoordinatorID] = \text{"init\_c"} \\
& \land pState = \text{"publish"} \land swapGraph \neq \text{"init"} \\
& \land coordState' = \text{"published"} \\
& \land ProofPublish' = \text{TRUE} \\
& \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"decision"}] \\
& \land \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofLock, ProofOkRM, \\
& \qquad\qquad ProofOkRF \rangle
\end{aligned}
$$

The $decision$ action in **Definition** 23 describes the two possible decisions that the coordinator can make. One of the two disjunctions must be valid when the action is validated. Thereby, one of the two predicates, $ValidTransfer$ and $AbortTransfer$, must be valid.

Definition 23 (The coordinator "decision" action)**.**

$$
\begin{aligned}
decision \triangleq\ & \land pc[CoordinatorID] = \text{"decision"} \\
& \quad \land \lor \land ValidTransfer \\
& \quad\quad \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"decisionValid"}] \\
& \quad\quad \lor \land AbortTransfer \\
& \quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"decisionAbort"}] \\
& \quad \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad\qquad ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The *decisionValid* action, defined in **Definition** 24, updates the *coordState* variable to authorise the swap and updates the *ProofOkRM*'s next state to TRUE. The next state of the coordinator program control is its termination.

Definition 24 (The coordinator "valid decision" action)**.**

$$
\begin{aligned}
decisionValid \triangleq\ & \wedge pc[CoordinatorID] = \text{"decisionValid"} \\
& \wedge coordState' = \text{"okRM"} \\
& \wedge ProofOkRM' = \text{TRUE} \\
& \wedge pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"Done"}] \\
& \wedge \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad ProofLock, ProofOkRF \rangle
\end{aligned}
$$

The *decisionAbort* action, defined in **Definition** 25, authorises the refund by updating the next state of *coordState* to "okRF" and the *ProofOkRF* to TRUE. After giving its decision, the coordinator completes its program.

Definition 25 (The coordinator "abort decision" action)**.**

$$
\begin{aligned}
decisionAbort \triangleq\ & \wedge pc[CoordinatorID] = \text{"decisionAbort"} \\
& \wedge coordState' = \text{"okRF"} \\
& \wedge ProofOkRF' = \text{TRUE} \\
& \wedge pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"Done"}] \\
& \wedge \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad ProofLock, ProofOkRM \rangle
\end{aligned}
$$

The sources actions. Formally, a source is defined by the following disjunction actions:

$$Source(self) \triangleq init\_src(self) \vee lock(self) \vee published(self) \vee waitForD(self)$$

The formula *Source* is parametric and takes as input the source identifier *self*. The first action, defined in **Definition** 26, is $init\_src(self)$ and evaluates the state of the *swapGraph* variable. Depending on its state, whether the action sets the source program control to "Done"; hence no more source action is enabled, or if *ProofPublish* is valid and the variable *swapGraph* is evaluated to "correct", then the program control of the source *self* goes to the next action, *lock*.

Definition 26 (The source "initial" action)**.**

$$
\begin{aligned}
init\_src(self) \triangleq\ & \wedge pc[self] = \text{"init\_src"} \\
& \wedge \vee \wedge swapGraph = \text{"different"} \vee Timeout = \text{TRUE} \\
& \qquad \wedge pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
& \quad \vee \wedge ProofPublish = \text{TRUE} \wedge swapGraph = \text{"correct"} \\
& \qquad \wedge pc' = [pc \text{ EXCEPT } ![self] = \text{"lock"}] \\
& \wedge \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The *lock* action, defined in **Definition** 27, first evaluates if *ProofPublish* is valid and if the action caller, *self*, is a source and its corresponding asset is in "OwS" state. This condition prevents a participant other than a source from executing the action. Thus, if this condition is not met (the ELSE conjunction), no change is applied to the assets' and *ProofLock*'s state. Therfore, the conjunction UNCHANGED $\langle assets, ProofLock \rangle$ is satisfied, and the program control of the source goes to the next action. Conversely, validating the condition (the IF conjunction) changes the asset's state from "OwS" to "locked" and updates the source's *ProofLock*.

Definition 27 (The source "lock" action)**.**

$$
\begin{aligned}
lock(self) \triangleq\ & \land pc[self] = \text{"lock"} \\
& \land \text{IF } ProofPublish = \text{TRUE} \land self \in Sources \land assets[AofS(self)] = \text{"OwS"} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{"locked"}] \\
& \qquad\qquad \land ProofLock' = [ProofLock \text{ EXCEPT } ![self] = \text{TRUE}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\qquad \land \text{UNCHANGED } \langle assets, ProofLock \rangle \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{"published"}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad\qquad ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The *published* action, defined in **Definition** 28, evaluates if the asset of the action caller is correctly locked by checking the *proof-of-action* $ProofLock[self]$. If the condition is valid (the IF conjunction), the action adds the element *self* into the sequence *qrm* to confirm the *redeem* request by the source to the coordinator; otherwise (the ELSE conjunction), the *qrm* variable remains unchanged.

Definition 28 (The source "published" action)**.**

$$
\begin{aligned}
published(self) \triangleq\ & \land pc[self] = \text{"published"} \\
& \land \text{IF } self \in Sources \land ProofLock[self] = \text{TRUE} \land coordState = \text{"published"} \\
& \quad \text{THEN } \land qrm' = (qrm \cup \{self\}) \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\qquad \land qrm' = qrm \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{"waitForD"}] \\
& \land \text{UNCHANGED } \langle assets, pState, coordState, qrf, swapGraph, \\
& \qquad\qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The $waitForD(self)$ action, defined in **Definition** 29, is a disjunction of three possible outcomes: a valid *proof-of-action* for the *redeem* or the *refund* decision, or none of them, i.e. the coordinator has not given any decision. In the case of a valid *ProofOkRM*, no change is applied to the system variable, except for the source's program control that updates to "Done". A valid *ProofOkRM* modifies the source's asset state from "locked" to "OwS". In the case where the *coordState* variable is still in the *published* state, and the *Timeout* is TRUE, the action adds *self* to the variable *qrf* and updates the program control. However, it is possible that despite a request for a *refund*, the result is *redeem*. If the *refund* request were made just before the state change of the *coordState* variable, the request would be ignored, and the decision will be *redeem*.

Definition 29 (The source "wait for decision" action)**.**

$$
\begin{array}{l}
waitForD(self) \triangleq \ \land\, pc[self] = \text{"waitForD"} \\
\qquad \land \lor \land\, ProofOkRM = \text{TRUE} \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
\qquad\quad \land \text{UNCHANGED } \langle assets, qrf \rangle \\
\qquad\ \lor \land\, ProofOkRF = \text{TRUE} \\
\qquad\quad \land \text{IF } self \in Sources \land ProofOkRF = \text{TRUE} \land assets[AofS(self)] = \text{"locked"} \\
\qquad\qquad \text{THEN } \land\, assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{"OwS"}] \\
\qquad\qquad \text{ELSE } \ \land \text{TRUE} \\
\qquad\qquad\qquad\quad \land \text{UNCHANGED } assets \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
\qquad\quad \land\, qrf' = qrf \\
\qquad\ \lor \land\, coordState = \text{"published"} \land Timeout = \text{TRUE} \\
\qquad\quad \land \text{IF } coordState = \text{"published"} \\
\qquad\qquad \text{THEN } \land\, qrf' = (qrf \cup \{self\}) \\
\qquad\qquad \text{ELSE } \ \land \text{TRUE} \\
\qquad\qquad\qquad\quad \land\, qrf' = qrf \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"waitForD"}] \\
\qquad\quad \land \text{UNCHANGED } assets \\
\qquad \land \text{UNCHANGED } \langle pState, coordState, qrm, swapGraph, \\
\qquad\qquad\qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

The recipients actions. Formally, a recipient is defined by the two following disjunction actions:

$$Recipient(self) \triangleq init\_rcp(self) \lor waitForD\_rcp(self)$$

The action $init\_rcp(self)$, defined in **Definition** 30, has the same definition as $init\_src(self)$. The second level of the action is a disjunction on the $swapGraph$ state. Either the recipient program control changes to "Done" and no more action can be enabled, or the $swapGraph$ variable is correct, and the program control next state equals "waitForD_rcp". Except for the program control, no variable changes state.

Definition 30 (The recipient "initial" action)**.**

$$
\begin{array}{l}
init\_rcp(self) \triangleq \ \land\, pc[self] = \text{"init\_rcp"} \\
\qquad \land \lor \land\, swapGraph = \text{"different"} \lor Timeout = \text{TRUE} \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"Done"}] \\
\qquad\ \lor \land\, ProofPublish = \text{TRUE} \land swapGraph = \text{"correct"} \\
\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{"waitForD\_rcp"}] \\
\qquad \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, \\
\qquad\qquad\qquad swapGraph, ProofPublish, ProofLock, \\
\qquad\qquad\qquad ProofOkRM, ProofOkRF \rangle
\end{array}
$$

Like the $waitForD(self)$ source's action, the $waitForD\_rcp(self)$ action, defined in **Definition** 31, evaluates the decisions *proof-of-action*. The first disjunction that evaluates to valid $ProofOkRF$ makes no changes to the system, and the recipient has no further enable action. If the disjunction of a valid $ProofOkRM$ is satisfied, then the action changes the recipient's asset state from "locked" to "OwR" . The third disjunction changes the value of qrf if the state of the $coordState$ variable is still $published$. Consequently, the program control's next state is "waitForD_rcp(self)" action.

Definition 31 (The recipient "wait for decision" action)**.**

$$
\begin{array}{l}
waitForD\_rcp(self) \triangleq \land\, pc[self] = \text{“waitForD\_rcp”} \\
\qquad \land\, \lor\, \land\, ProofOkRF = \text{TRUE} \\
\qquad\qquad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\qquad\qquad \land\, \text{UNCHANGED } \langle assets, qrf \rangle \\
\qquad\qquad \lor\, \land\, ProofOkRM = \text{TRUE} \\
\qquad\qquad\qquad \land\, \text{IF } self \in Recipients \land ProofOkRM = \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, assets[AofR(self)] = \text{“locked”} \\
\qquad\qquad\qquad\qquad \text{THEN } \land\, assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{“OwR”}] \\
\qquad\qquad\qquad\qquad \text{ELSE } \land\, \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, \text{UNCHANGED } assets \\
\qquad\qquad\qquad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\qquad\qquad\qquad \land\, qrf' = qrf \\
\qquad\qquad \lor\, \land\, coordState = \text{“published”} \land Timeout = \text{TRUE} \\
\qquad\qquad\qquad \land\, \text{IF } coordState = \text{“published”} \\
\qquad\qquad\qquad\qquad \text{THEN } \land\, qrf' = (qrf \cup \{self\}) \\
\qquad\qquad\qquad\qquad \text{ELSE } \land\, \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, qrf' = qrf \\
\qquad\qquad\qquad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“waitForD\_rcp”}] \\
\qquad\qquad\qquad \land\, \text{UNCHANGED } assets \\
\qquad \land\, \text{UNCHANGED } \langle pState, coordState, qrm, swapGraph, \\
\qquad\qquad\qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

The Byzantine sources actions. A Byzantine participant has unpredictable behaviour, so it is impossible to define a precise protocol. However, to apply verification tools to the model, it is necessary to represent the impact of Byzantine participants in our system. To do so, we have defined possible actions of a Byzantine source while being as little restrictive as possible. Formally, the following formula defines a Byzantine source:

$$
\begin{array}{l}
BSource(self) \triangleq init\_bsrc(self) \lor BdirectToR(self) \lor Bother(self) \lor BaskRM(self) \lor \\
\qquad\qquad\qquad\qquad BlockAsset(self) \lor BSaskRF(self) \lor BrecoveringAsset(self)
\end{array}
$$

The first possible action of a Byzantine source is $init\_bsrc$, defined in **Definition** 32. This action is the disjunction of all possible actions of a Byzantine source defined in its corresponding PlusCal code. They can be executed randomly without respecting a given order.

Definition 32 (The Byzantine source "initial" action)**.**

$$
\begin{array}{l}
init\_bsrc(self) \triangleq \land\, pc[self] = \text{“init\_bsrc”} \\
\qquad \land\, \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“BdirectToR”}] \\
\qquad\qquad \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“Bother”}] \\
\qquad\qquad \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“BaskRM”}] \\
\qquad\qquad \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“BlockAsset”}] \\
\qquad\qquad \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“BSaskRF”}] \\
\qquad\qquad \lor\, \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“BrecoveringAsset”}] \\
\qquad \land\, \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, \\
\qquad\qquad\qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

One of the possible actions that a Byzantine source can execute is $BdirectToR(self)$. The action, defined in **Definition** 33, modifies the asset variable of the Byzantine $self$ from "OwS" to "OwR". The protocol does not allow this transition; however, it can be performed by a Byzantine source.

Definition 33 (the Byzantine source "direct to the recipient" action)**.**

$$
\begin{aligned}
BdirectToR(self) \triangleq\ & \wedge pc[self] = \text{"BdirectToR"} \\
& \wedge \text{IF } self \in Sources \wedge assets[AofS(self)] = \text{"OwS"} \\
& \quad \text{THEN } \wedge assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{"OwR"}] \\
& \quad \text{ELSE } \wedge \text{TRUE} \\
& \qquad\ \ \wedge \text{UNCHANGED } assets \\
& \wedge pc' = [pc \text{ EXCEPT } ![self] = \text{"init\_bsrc"}] \\
& \wedge \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The action defined in **Definition** 34 is the second specific action of Byzantine sources. $Bother$ action modifies the $self$'s asset state and sets its value to "other".

Definition 34 (The Byzantine source "other" action)**.**

$$
\begin{aligned}
Bother(self) \triangleq\ & \wedge pc[self] = \text{"Bother"} \\
& \wedge \text{IF } self \in Sources \wedge assets[AofS(self)] = \text{"OwS"} \\
& \quad \text{THEN } \wedge assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{"other"}] \\
& \quad \text{ELSE } \wedge \text{TRUE} \\
& \qquad\ \ \wedge \text{UNCHANGED } assets \\
& \wedge pc' = [pc \text{ EXCEPT } ![self] = \text{"init\_bsrc"}] \\
& \wedge \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

We notice from **Definition** 33 and **Definition** 34 that actions specific to Byzantine sources can only modify the state of their asset and their program control. They do not influence other variables from the system.
A Byzantine source can execute the same actions as correct sources (see the definitions below). Hence, the actions $BaskRM(self)$, $BlockAsset(self)$, $BSaskRF(self)$ and $BrecoveringAsset(self)$ have the same execution as correct source actions. The only difference is that when an action is executed, the Byzantine source never terminates and always returns to the $init\_bsrc$ action.

$$
\begin{aligned}
&BaskRM(self) \triangleq askRM(self) \\
&BlockAsset(self) \triangleq lockAsset(self) \\
&BSaskRF(self) \triangleq askRF(self) \\
&BrecoveringAsset(self) \triangleq recoveringAsset(self)
\end{aligned}
$$

The Byzantine recipient actions. Formally, a Byzantine recipient is defined by the following five disjunctions actions:

$$
\begin{aligned}
BRecipient(self) \triangleq\ & init\_brcp(self) \vee BRaskRF(self) \vee BRretrievingAsset(self) \vee \\
& BRdirectToS(self) \vee BRother(self)
\end{aligned}
$$

A Byzantine recipient starts with the action $init\_brcp(self)$, defined in **Definition** 35, which is the disjunction of all possible actions that a Byzantine recipient can execute. Each action can be performed as often as possible in a non-deterministic way.

Definition 35 (The Byzantine recipient "initial" action)**.**

$$
\begin{aligned}
init\_brcp(self) \triangleq\ & \land pc[self] = \text{"init\_brcp"} \\
& \land \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{"BRaskRF"}] \\
& \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{"BRretrievingAsset"}] \\
& \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{"BRdirectToS"}] \\
& \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{"BRother"}] \\
& \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

Thus, like the source, a Byzantine recipient executes specific actions and actions of a correct recipient. $BRdirectToS(self)$ and $BRother(self)$ are specific to Byzantine recipients. $BRdirectToS(self)$, defined in **Definition** 36, modifies $self$'s asset variable from "OwR" to "OwS".

Definition 36 (The Byzantine recipient "direct to the source" action)**.**

$$
\begin{aligned}
BRdirectToS(self) \triangleq\ & \land pc[self] = \text{"BRdirectToS"} \\
& \land \text{IF } self \in Recipients \land assets[AofR(self)] = \text{"OwR"} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{"OwS"}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\ \ \land \text{UNCHANGED } assets \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{"init\_brcp"}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

The action $BRother(self)$, defined in **Definition** 37, modifies the asset from "OwR" to "other" without modifying the other system's variables.

Definition 37 (The Byzantine recipient "other" action)**.**

$$
\begin{aligned}
BRother(self) \triangleq\ & \land pc[self] = \text{"BRother"} \\
& \land \text{IF } self \in Recipients \land assets[AofR(self)] = \text{"OwR"} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{"other"}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\ \ \land \text{UNCHANGED } assets \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{"init\_brcp"}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, ProofPublish, \\
& \qquad ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

Similarly to the Byzantine sources, the Byzantine recipients only update their asset's state and program control. The actions $BRaskRF(self)$ and $BRretrievingAsset(self)$ execute the same action of $askRF(self)$ and $retrievingAsset(self)$ (see the code below).

$$
\begin{aligned}
BRaskRF(self) &\triangleq askRF(self) \\
BRretrievingAsset(self) &\triangleq retrievingAsset(self)
\end{aligned}
$$

The complete TLA$^+$ code of $\mathscr{P}_{swap}$ is defined in the Appendix A.2.

6.6 Conclusion

This chapter presents a protocol that is becoming popular in the blockchain area, namely the *cross-chain swap* application. Their benefits in exchanging assets without an intermediary makes them ideal for being used in assets transfer between different blockchains. This chapter aims to establish the swap problem and its specification formally. In addition, we put forward a blockchain agnostic protocol, $\mathscr{P}_{swap}$, which aims at satisfying the swap specification. The protocol is built in a way that prevents possible Byzantine actions from violating the specification. We use the notion of *proof-of-action* to implement a mechanism of verifiable proof to ensure the correct behaviour of the participants. In a second step, we describe the protocol implemented in a specification language TLA$^+$. This step is necessary to apply verification tools to prove that our protocol satisfies the problem specification.

Chapter 7

Proof of $\mathscr{P}_{swap}$ Correctness

" What we know is a drop, what we don't know is an ocean. "

– Isaac Newton

Contents

In this chapter, we prove the protocol $\mathscr{P}_{swap}$ using formal methods provided by the TLA$^+$ tool. We focus on the theorem proving [163] for the safety property (Section 7.1) and the model-checking [58] for proving the liveness properties (Section 7.2). Regarding the safety proof approach, we apply the proof methodology described in Section 4.3.4. The methodology describes an inductive invariant and proves that this invariant is maintained in the system's initial state, then at each stage of the system's behaviour, and finally that the invariant satisfies the safety property. We prove that our system specification satisfies the safety property by assuming these three steps. Before going through the $TLAPS$ proof, we do a rough hand proof to understand the intuition of the TLA$^+$ approach. In contrast to the proof of safety, the methodology of proving liveness consists of defining the liveness properties and launch the model checker, TLC, on the system. To get conclusive results, we vary the value of constants, which are the number of transactions and the share of Byzantine participants in the system.

7.1 Proof of the Safety Property

As a recall, the *Consistency* property of a *cross-chain swap* problem, introduced in Section 6.2, is defined as follows:

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 the swap execution, either s_1 owns a_1 or r_2 owns a_2*".

As a first step, we provide a handwritten proof of safety that consists of four lemmas and the safety theorem. The proof is built around the behavioural possibilities of the coordinator. Then, in a second step, we transcribe the handwritten proof into semi-automatic proof using $TLAPS$.

7.1.1 Handwritten Proof of the *Consistency* Property

In this section, we prove manually the safety property of the swap problem defined in Section 6.2. Properties are written as an LTL formula (see Section 4.1), and we need to introduce some definitions to express the necessary lemma and theorem. Let $loc(x)$ be the location state of 'x', e.g. $loc(a)$ is the location state of the asset a. Once the coordinator decides the swap outcome, assets are described as *available* (whether to the source or the recipient). Therefore, we introduce a predicate $\mathscr{A}_r(a)$ that takes an asset identifier as input, and it describes "*available to its recipient*" with:

$$\mathscr{A}_r(a) = (loc(a) = "OwR" \vee (Proof_{redeem} \wedge loc(a) = "locked"))$$

The first predicate member represents the redeemed asset a by the recipient. The second member is when the *redeem* decision is available (a valid $Proof_{redeem}$), but the recipient has not yet redeemed its asset a. Similarly, $\mathscr{A}_s(a)$ is the predicate that defines "*available to its source*" with:

$$\mathscr{A}_s(a) = (loc(a) = "OwS" \vee (Proof_{refund} \wedge loc(a) = "locked"))$$

The first member represents the recovered asset a by the source. The second predicate member is when the *refund* decision is available (a valid $Proof_{refund}$), but the source has not yet recovered its asset a.

In the following, we recall the symbols defined in Chapter 6:

- Π: the set of participants (sources and recipients).
- Π_s: the set of sources.
- Π_r: the set of recipients.

- Π_c: the set of correct participants with $\Pi_c \subseteq \Pi$.
- Λ: the set of assets of the swap.
- *Proof* $_{redeem}$: the *redeem* decision *proof-of-action*.
- *Proof* $_{publish}$: the swap graph publication *proof-of-action*.
- *Proof* $_{refund}$: the *refund* decision *proof-of-action*.
- *Proof* $_{lock}$: the lock asset *proof-of-action*.

We introduce two additional symbols defined as follows:

- Λ_s: the set of assets initially owned by correct sources, with $\Lambda_s \subseteq \Lambda$.
- Λ_r: the set of assets intended for correct recipients, with $\Lambda_r \subseteq \Lambda$.

Let us denote 'c' the coordinator.

Lemma 7.1. *When the coordinator is in its initial state, then no correct sources are in published state and, assets initially owned by a correct source are owned by their source.*

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

Proof. From Figure 6.4, we can see that in the initial state, the coordinator has not triggered ϵ_1^c. Hence, no correct sources and correct recipients (Figures 6.5 and 6.6) will have their guard $\sigma_2^{s_i}$ and $\sigma_2^{r_i}$ satisfied. However, $\sigma_1^{s_i}$ and $\sigma_1^{r_i}$ can be satisfied if the publisher takes a long time to trigger ϵ_1^p (Figure 6.3), i.e. more time than the source's or recipient's timeout. Consequently, correct participants can exit the swap. Correct sources would not lock their assets in both scenarios, and these remain owned by their source.

□

Lemma 7.2. *When the coordinator is in "PUBLISHED" state, then no assets initially owned by a correct source are available to their recipient.*

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

Proof. When the coordinator is in the "PUBLISHED" state, then ϵ_1^p has been triggered by the publisher in Figure 6.3, allowing the coordinator to change its state. Consequently, correct participants will verify "*Proof* $_{publish}$"; if the proof is valid, correct sources could lock their assets (executing $\omega_2^{s_i}$) and trigger $\epsilon_1^{a_i}$ from Figure 6.2. Since the coordinator is in the "PUBLISHED" state, neither σ_2^c nor σ_3^c is satisfied. Thereby, no decision has been taken by the coordinator. Therefore, an asset cannot be available to the recipient as long as the coordinator is in the "PUBLISHED" state. □

Lemma 7.3. *When the coordinator gives a redeem decision, then all assets are available to their recipient.*

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

Proof. For the coordinator to make a *redeem* decision, σ_3^c from Figure 6.4 must be satisfied. ValidTransfer is satisfied when all sources have executed the action $\epsilon_3^{s_i}$ from Figure 6.5, and the "*Proof* $_{lock}$" provided by the sources to the coordinator is correct and valid. Consequently, satisfying σ_3^c makes all assets accessible to their recipients. Depending on the recipient's behaviour, assets can stay in "LOCKED" or move to the "OWR" state by using "*Proof* $_{redeem}$" to satisfy AuthoRM. In both cases, the assets are available to their recipient. If the recipient is correct, its asset will eventually be retrieved by executing $\omega_5^{r_i}$. □

Lemma 7.4. *When the coordinator gives a refund decision, then assets initially owned by a correct source are available to their source.*

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

Proof. For the coordinator to make a *refund* decision, σ_4^c from Figure 6.4 must be satisfied. Hence, the conditions for the AbortTransfer predicate are fulfilled. Namely, either a "*Proof* $_{lock}$" provided by a source has been proven invalid, or there exists a participant who asked for a *refund* decision (triggering $\epsilon_5^{s_i}$ if the participant is a source or triggering $\epsilon_4^{r_i}$ 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 the source's behaviour, assets can stay in "LOCKED" or move to the "OwS" state by using "*Proof* $_{refund}$" to satisfy AuthoRF. Both cases set the assets available to their source. If the source is correct, its asset will eventually be recovered by executing $\omega_6^{s_i}$. □

Theorem 7.5. *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 the swap execution, either s_1 owns a_1 or r_2 owns a_2.*

Proof. We have proven from **Lemma** 7.1 that a *correct* source s_1 can timeout and finish its execution before locking its asset a_1. Consequently, a_1 remains in the "OwS" state. **Lemma** 7.3 proves that a *correct* recipient r_2 can finish its execution by retrieving its asset a_2 if a *redeem* decision is given. In that case, the asset's state changes to "OwR". However, though r_2 can timeout at the beginning of the swap (before the swap graph publication), a_2 is accessible by the recipients when the *redeem* decision is given. Indeed, they can retrieve a_2 asynchronously since the decision will always be available. From **Lemma** 7.4, s_1 finishes its execution by recovering its asset if a *refund* decision is given. 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** 7.1, **Lemma** 7.2 and **Lemma** 7.4, 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's decision. From **Lemma** 7.3, 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; then the outcome is that s1 owns its asset or r2 owns its asset. Hence, the *Consistency* property of the swap is proven. □

7.1.2 Proof of the *Consistency* Property Using *TLAPS*

In the following, we demonstrate the proof strategy described in Section 4.3.4 applied to the *cross-chain swap* system. The methodology is divided into three steps. The first step is to define the safety property – the *Consistency*. The second step is to define an inductive invariant according to the coordinator's behaviour. Finally, the third step is the proof of the resulting invariant.

Step 1. The definition of the *Consistency* property. In the previous Section 7.1.1, we introduced two predicates, "*available to its source*" and "*available to its recipient*", describing assets' availability. We define these two predicates into a TLA$^+$ formalism in **Definition** 38.

Definition 38 (Asset's availability predicates)**.**

$$AvailableS(a) \triangleq assets[a] = \text{"OwS"} \lor (ProofOkRF = \text{TRUE} \land assets[a] = \text{"locked"})$$
$$AvailableR(a) \triangleq assets[a] = \text{"OwR"} \lor (ProofOkRM = \text{TRUE} \land assets[a] = \text{"locked"})$$

$AvailableS(a)$ (resp. $AvailableR(a)$) is a predicate that evaluates the asset ownership whether is owned by its source, $assets[a] =$ "OwS" (resp. by its recipient, $assets[a] =$ "OwR"), or accessible by the source, $ProofOkRF = \text{TRUE} \wedge assets[a] =$ "locked" (resp. by the recipient, $ProofOkRM = \text{TRUE} \wedge assets[a] =$ "locked").
Accessible by source or recipient describes 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. Consequently, the TLA$^+$ *Consistency* property is defined in **Invariant** 1.

Invariant 1 (Consistency Property)**.**

$$Consistency \triangleq \forall s \in CSources, r \in CRecipients : Finish(s,r) \implies AvailableS(AofS(s)) \vee AvailableR(AofR(r))$$

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 finished their protocol:

$$Finish(s,r) \triangleq pc[s] = \text{"Done"} \wedge pc[r] = \text{"Done"}$$

$AofS$ and $AofR$ are defined in Section 6.5.1.

Step 2. The definition of the inductive invariant. The strategy described in Section 4.3.4 defines 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:

$$\frac{Init \implies Inv \quad Inv \wedge Next \implies Inv' \quad Inv \implies Consistency}{Spec \implies \Box Consistency} \tag{7.1}$$

We construct the inductive invariant Inv of the $\mathscr{P}_{swap}$ system based on the proof methodology described in Section 4.3.4 for proving the *Two-Phase Commit* algorithm and the proven lemmas in the previous Section 7.1.1. The following predicate captures the inductive invariant:

$$Inv = TypeOk \wedge CoordInv$$

with $TypeOk$, the type correctness invariant, defined in **Invariant** 2, and $CoordInv$, defined in **Invariant** 3, the predicate that specifies the system's state of each variable at each coordinator's step.

Type correctness invariant. The TLA$^+$ language is untyped; thus, state variables should conform to their expected data structure to ensure successful access to data. This requirement represents the variables' type correctness for the proof of safety, and it is proved to be an invariant for the system. Therefore, $TypeOk$, defined in **Invariant** 2, asserts that all relevant variables have values of the expected sets.

Invariant 2 (Type Correctness Invariant)**.**

$$
\begin{aligned}
TypeOk \triangleq\ & \wedge assets \in [Assets \rightarrow AStates] \\
& \wedge pState \in PStates \\
& \wedge coordState \in CStates \\
& \wedge ProofLock \in [Sources \rightarrow \text{BOOLEAN }] \\
& \wedge ProofPublish \in \text{BOOLEAN} \\
& \wedge ProofOkRM \in \text{BOOLEAN} \\
& \wedge ProofOkRF \in \text{BOOLEAN} \\
& \wedge qrm \subseteq Sources \\
& \wedge qrf \subseteq Pi \\
& \wedge swapGraph \in SwapStates \\
& \wedge pc[CoordinatorID] \in \{\text{“init\_c”}, \text{“decision”}, \text{“decisionValid”}, \text{“decisionAbort”}, \text{“Done”}\} \\
& \wedge pc[CoordinatorID] = \text{“Done”} \implies coordState \in \{\text{“okRM”}, \text{“okRF”}\} \\
& \wedge \forall\, s \in CSources : pc[s] \in \{\text{“published”}, \text{“waitForD”}, \text{“init\_src”}, \text{“lock”}, \text{“Done”}\} \\
& \wedge \forall\, r \in CRecipients : pc[r] \in \{\text{“init\_rcp”}, \text{“waitForD\_rcp”}, \text{“Done”}\} \\
& \wedge pc \in [ProcSet \rightarrow Labels]
\end{aligned}
$$

With $Labels$ (**Definition** 39) the set of all defined labels in the module:

Definition 39 (Labels Definition)**.**

$$
\begin{aligned}
Labels \triangleq\ & \{\text{“init\_c”}, \text{“decision”}, \text{“decisionValid”}, \text{“decisionAbort”}, \text{“Done”}, \text{“init\_p”}, \text{“init\_src”}, \text{“lock”}, \\
& \text{“published”}, \text{“waitForD”}, \text{“refunded”}, \text{“Done”}, \text{“init\_bsrc”}, \text{“BdirectToR”}, \text{“Bother”}, \\
& \text{“BaskRM”}, \text{“BlockAsset”}, \text{“BSaskRF”}, \text{“BrecoveringAsset”}, \text{“init\_rcp”}, \text{“waitForD\_rcp”}, \\
& \text{“redeemed”}, \text{“exit\_rcp”}, \text{“Done”}, \text{“init\_brcp”}, \text{“BRaskRF”}, \text{“BRretrievingAsset”}, \\
& \text{“BRdirectToS”}, \text{“BRother”}\}
\end{aligned}
$$

For example, the invariant ensures that the domain set values of the $assets$ variable is $Assets$ that maps to $AStates$. With $Assets$, the set of asset identifiers and $AStates$ the possible states of an asset. The set of values of the *proof-of-actions* ($ProofPublish$, $ProofOkRM$, $ProofOkRF$) is the set of booleans. Note that the variables qrm and qrf are not defined in the same domain. Indeed, qrm is a variable that translates the function calls of $askRM$ defined in Section 6.5.2. In the TLA$^+$ model, if a participant calls the $askRM$ function, then the caller's identifier is added to qrm. The protocol only allows sources to execute this function. Therefore, the elements of qrm are included in the $Sources$ set. Conversely, qrf translates $askRF$ function calls defined in Section 6.5.2. Both sources and recipients can execute this function. Therefore, the elements of qrf are included in Pi, which corresponds to the set of the system's participants. Moreover, the program control type is also defined, depending on whether it is of the coordinator, a source or a recipient. That makes it possible to prove that a source cannot execute an action defined for the coordinator.

Coordinator correctness invariant. The **Invariant** 3 consists of six conjunctions, where three of them have an equivalent proven lemma. The first conjunction describes **Lemma** 7.1, the fifth conjunction describes **Lemma** 7.3, and the sixth conjunction describes **Lemma** 7.4. The **Lemma** 7.2 is represented by the second, third, and fourth conjunctions. Intermediate states do not appear explicitly in the handwritten approach, thanks to abstraction and infer steps. However, using a formal tool need to describe all the intermediate steps. It is necessary to make everything explicit to constitute complete and sufficient proof. Recall that $CoordinatorID$ is the identifier of the coordinator.

Invariant 3 (The Coordinator Invariant)**.**

$$
\begin{aligned}
CoordInv \triangleq\ & \wedge pc[CoordinatorID] = \text{“init\_c”} \implies init\_cInv \\
& \wedge pc[CoordinatorID] = \text{“decision”} \implies decisionInv \\
& \wedge pc[CoordinatorID] = \text{“decisionValid”} \implies decisionValidInv \\
& \wedge pc[CoordinatorID] = \text{“decisionAbort”} \implies decisionAbortInv \\
& \wedge (pc[CoordinatorID] = \text{“Done”} \wedge coordState = \text{“okRM”}) \implies okRMInv \\
& \wedge (pc[CoordinatorID] = \text{“Done”} \wedge coordState = \text{“okRF”}) \implies okRFInv
\end{aligned}
$$

The $CoordInv$ elements are defined as follows: {“init_c”, “decision”, “decisionValid”, “decisionAbort”, “Done”} are labels of the coordinator and $pc[]$ the program control variable that tracks which label the coordinator is currently on. The elements $okRMInv, init\_cInv, decisionInv, decisionValidInv, decisionAbortInv, okRFInv$ are sub-invariants defined later, and $coordState$ represents the state of the coordinator. The defined invariant $CoordInv$ represents the overall correctness of the swap. Therefore, it must be sufficiently complete to permit the proof of *Consistency*. The effort in this part of the proof is the construction of the invariant. It requires a lot of work done in several iterations before having the inductive invariant for the proof. In the following, we describe each coordinator's action:

1. The coordinator is in “init_c”. The first conjunction is constructed according to **Lemma** 7.1. The **Invariant** 4 describes the initial state of the coordinator that does not allow the majority of the variables to evolve or change. For example, *proof-of-actions* remain in their initial state, which is FALSE and cannot evolve while the coordinator is in its initial state. A correct source may exit the swap, described by its program control equal to “Done”. The same is true for recipients. On the other hand, the publisher can remain in its initial state, just like the $swapGraph$ variable. If the publisher takes action and publishes the graph, whether the publisher is correct, i.e. the $swapGraph$ changes its state to “correct”, or the publisher is Byzantine, i.e. the $swapGraph$ changes its state to “ different”.

Invariant 4 (The $init\_cInv$ predicate)**.**

$$
\begin{aligned}
init\_cInv \triangleq\ & \wedge coordState = \text{“init”} \\
& \wedge (ProofOkRM = \text{FALSE} \wedge ProofOkRF = \text{FALSE} \wedge ProofPublish = \text{FALSE}) \\
& \wedge qrf = \{\} \\
& \wedge qrm = \{\} \\
& \wedge \forall s \in Sources : \wedge ProofLock[s] = \text{FALSE} \\
& \wedge \forall s \in CSources : \wedge pc[s] \in \{\text{“init\_src”}, \text{“Done”}\} \\
& \qquad\qquad \wedge ProofLock[s] = \text{FALSE} \\
& \qquad\qquad \wedge assets[AofS(s)] = \text{“OwS”} \\
& \wedge \forall r \in CRecipients : pc[r] \in \{\text{“init\_rcp”}, \text{“Done”}\} \\
& \wedge swapGraph = \text{“init”} \implies pState = \text{“init”} \\
& \wedge swapGraph = \text{“correct”} \implies pState = \text{“publish”} \\
& \wedge swapGraph = \text{“different”} \implies pState = \text{“publish”} \\
& \wedge \forall a \in AssetsFromCS : assets[a] = \text{“OwS”} \\
& \wedge pState = \text{“publish”} \implies pc[PublisherID] = \text{“Done”}
\end{aligned}
$$

Note that $AssetsfromCS$ is the set of assets of correct sources defined as:

$$AssetsFromCS \triangleq \{AofS(x) : x \in CSources\}$$

2. The coordinator is in "decision". The second conjunction is the one that describes the state of the variables when the coordinator is in the published state, and the "decision" action is enabled. The **Invariant** 5 is constructed according to **Lemma** 7.2. The invariant asserts that the publisher has finished its protocol (its program control is at "Done"), and it has published the swap graph (its state is at "publish"). Therefore, the *proof-of-action* $ProofPublish$ must be set to TRUE. Given the distributed and asynchronous aspect of the system, the sources are not at the same level in the protocol. The sources that execute the "published" or "waitForD" actions have locked their asset, and their *proof-of-action* must be valid. Conversely, the correct sources that execute $init\_src$ or $lock$ action still own their asset and have not yet validated their *proof-of-action* $ProofLock[]$. At this stage of the coordinator protocol, if a source finishes its protocol, i.e. its program control is at "Done", the source has decided to leave the swap before locking its asset. That describes that the correct source has reached its timeout while waiting for the graph publication. The only possible states of an asset are "OwS" if the source has not yet locked the asset and still owns it or "locked" if the asset is locked.

Invariant 5 (The $decisionInv$ predicate)**.**

$$
\begin{aligned}
decisionInv \triangleq\ & \wedge coordState = \text{"published"} \\
& \wedge (pState = \text{"publish"} \wedge pc[PublisherID] = \text{"Done"}) \\
& \wedge (ProofPublish = \text{TRUE} \wedge ProofOkRM = \text{FALSE} \wedge ProofOkRF = \text{FALSE}) \\
& \wedge \forall s \in Sources : \wedge s \in qrm \implies ProofLock[s] = \text{TRUE} \\
& \qquad\qquad \wedge ProofLock[s] = \text{TRUE} \implies assets[AofS(s)] = \text{"locked"} \\
& \wedge \forall s \in CSources : \wedge pc[s] \in \{\text{"published"}, \text{"waitForD"}\} \\
& \qquad\qquad \implies \wedge ProofLock[s] = \text{TRUE} \wedge assets[AofS(s)] = \text{"locked"} \\
& \qquad\qquad \wedge pc[s] \in \{\text{"init\_src"}, \text{"lock"}, \text{"Done"}\} \\
& \qquad\qquad \implies \wedge ProofLock[s] = \text{FALSE} \wedge assets[AofS(s)] = \text{"OwS"} \\
& \qquad\qquad \wedge pc[s] \in \{\text{"init\_src"}, \text{"lock"}, \text{"Done"}, \text{"published"}\} \implies s \notin qrm \\
& \qquad\qquad \wedge s \in qrm \implies pc[s] = \text{"waitForD"} \\
& \wedge \forall a \in AssetsFromCS : assets[a] \in \{\text{"locked"}, \text{"OwS"}\}
\end{aligned}
$$

3. The coordinator is in "decisionValid". The third conjunction of the invariant is an intermediate step imbedded in **Lemma** 7.2. The invariant $decisionValidInv$ defined in **Invariant** 6 describes the system's state that satisfies the conditions for the coordinator to authorise the redeem. All sources must have asked for a *redeem* decision. Therefore, the set qrm must contain all elements of the set $Sources$. As a result, the *proof-of-action* $ProofLock[]$ of all sources must be valid, and all assets are in "locked" state.

Invariant 6 (The $decisionValidInv$ predicate)**.**

$$
\begin{aligned}
decisionValidInv \triangleq\ & \wedge coordState = \text{"published"} \\
& \wedge (pState = \text{"publish"} \wedge pc[PublisherID] = \text{"Done"}) \\
& \wedge (ProofPublish = \text{TRUE} \wedge ProofOkRM = \text{FALSE} \wedge ProofOkRF = \text{FALSE}) \\
& \wedge qrm = Sources \\
& \wedge \forall s \in Sources : \wedge ProofLock[s] = \text{TRUE} \\
& \qquad\qquad \wedge assets[AofS(s)] = \text{"locked"} \\
& \wedge \forall s \in CSources : \wedge pc[s] \in \{\text{"waitForD"}\} \\
& \wedge \forall r \in CRecipients : \wedge assets[AofR(r)] = \text{"locked"} \\
& \qquad\qquad \wedge pc[r] = \text{"init\_src"} \implies assets[AofR(r)] = \text{"locked"} \\
& \wedge qrm = Sources \implies \forall a \in Assets : assets[a] = \text{"locked"}
\end{aligned}
$$

4. The coordinator is in "decisionAbort". As the $decisionValidInv$ invariant, the $decisionAbortInv$ is an additional step needed for the $TLAPS$ proof. The invariant defined in **Invariant** 7 is the system's state that satisfies the condition for the coordinator to authorise the assets refund. The state of qrf must not be empty, which implies that at least one participant has made the *refund* request. At this protocol stage, the assets of correct sources are either in the "locked" state or "OwS" state.

Invariant 7 (The $decisionAbortInv$ predicate)**.**

$$
\begin{aligned}
decisionAbortInv \triangleq\ & \wedge coordState = \text{"published"} \\
& \wedge (pState = \text{"publish"} \wedge pc[PublisherID] = \text{"Done"}) \\
& \wedge (ProofOkRM = \text{FALSE} \wedge ProofOkRF = \text{FALSE} \wedge ProofPublish = \text{TRUE} \\
& \wedge qrf \neq \{\} \\
& \wedge \forall s \in CSources : \wedge assets[AofS(s)] \in \{\text{"locked"}, \text{"OwS"}\} \\
& \qquad\qquad \wedge assets[AofS(s)] = \text{"locked"} \implies ProofLock[s] = \text{TRUE} \\
& \wedge pc[s] \in \{\text{"Done"}, \text{"init\_src"}\} \implies assets[AofS(s)] = \text{"OwS"} \\
& \wedge \forall a \in AssetsFromCS : assets[a] \in \{\text{"locked"}, \text{"OwS"}\}
\end{aligned}
$$

5. The coordinator is in "Done" with a redeem decision. The fifth conjunction describes the system's state that allows the recipients to redeem their assets. The invariant $okRMInv$ defined in **Invariant** 8 is constructed according to **Lemma** 7.3. The state variable of $coordState$ is "okRM", and the *proof-of-action* $ProofOkRM$ must be valid. Consequently, recipients can asynchronously change the state of their assets from "locked" to "OwR". A *redeem* decision implies that all sources are correct participants. However, it is possible for recipients to be Byzantine so that the assets intended for them remain in the "locked" or in the "other" state (see Figure 6.2).

Invariant 8 (The $okRMInv$ predicate)**.**

$$
\begin{aligned}
okRMInv \triangleq\ & \wedge (ProofOkRM = \text{TRUE} \wedge ProofOkRF = \text{FALSE} \wedge ProofPublish = \text{TRUE}) \\
& \wedge qrm = Sources \\
& \wedge \forall s \in CSources : \wedge pc[s] \in \{\text{"waitForD"}, \text{"Done"}\} \\
& \wedge \forall r \in CRecipients : \wedge assets[AofR(r)] \in \{\text{"locked"}, \text{"OwR"}\} \\
& \qquad\qquad \wedge pc[r] = \text{"Done"} \\
& \qquad\qquad\quad \implies assets[AofR(r)] \in \{\text{"OwR"}, \text{"locked"}\} \\
& \qquad\qquad \wedge pc[r] \in \{\text{"init\_rcp"}, \text{"waitForD\_rcp"}\} \\
& \qquad\qquad\quad \implies assets[AofR(r)] = \text{"locked"} \\
& \qquad\qquad \wedge \quad pc[r] = \text{"init\_src"} \implies assets[AofR(r)] = \text{"locked"} \\
& \qquad\qquad \wedge \quad qrm = Sources \implies \forall a \in AssetsForCR : \\
& \qquad\qquad\qquad\qquad\qquad\qquad assets[a] \in \{\text{"locked"}, \text{"OwR"}\}
\end{aligned}
$$

Note that $AssetsforCR$ is the set of assets of correct recipients defined as:

$$AssetsForCR \triangleq \{AofR(x) : x \in CRecipients\}$$

6. The coordinator is in "Done" with a refund decision. The sixth and last conjunction corresponds to the system's state that authorises the refund of assets. The invariant $okRFInv$ defined in **Invariant** 9 is constructed according to **Lemma** 7.4. The *proof-of-action* $ProofOkRF$ must be valid, and the state of the $coordState$ variable is at "okRF". Consequently, the correct sources can asynchronously change the state of the assets from "locked" to "OwS" if the asset has been locked before the *refund* decision. However, a Byzantine source may leave the asset in the "locked" state or perform any other action unknown to the protocol (see Section 6.5.2 for the possible Byzantine sources' behaviour).

Invariant 9 (The $okRFInv$ predicate)**.**

$$
\begin{aligned}
okRFInv \triangleq\ & \wedge (ProofOkRM = \text{FALSE} \wedge ProofOkRF = \text{TRUE} \wedge ProofPublish = \text{TRUE}) \\
& \wedge qrf \neq \{\} \\
& \wedge \forall\, s \in CSources : \wedge\, assets[AofS(s)] \in \{\text{“locked”}, \text{“OwS”}\} \\
& \qquad\qquad \wedge\, pc[s] \in \{\text{“init\_src”}, \text{“Done”}\} \implies assets[AofS(s)] = \text{“OwS”} \\
& \wedge \forall\, a \in AssetsFromCS : assets[a] \in \{\text{“locked”}, \text{“OwS”}\}
\end{aligned}
$$

Remark. In the proof approach of the *Two-Phase Commit* algorithm in Section 4.3.4, it was necessary to add a conjunction that dealt with the case where the coordinator's program control is at "Done", and the coordinator state is at "pre-commit", and "init". However, the approach defined in this chapter did not require the addition of equivalent cases because additional conjunction was added in the $TypeOk$ invariant. The twelfth conjunction in $TypeOk$: "$pc[CoordinatorID]$ = "Done" $\implies$ coordState $\in$ {"okRM", "okRF"}", ensures that if the coordinator's program control is at "Done", then the only possible states of $coordState$ are either "okRM" or "okRF". This example shows the interest in defining the $TypeOk$ invariant. It can lighten the construction of a proof.

Step 3. The proof of the resulting invariant. We have fulfilled two steps of the proof methodology. It remains the last step, which is the proof of the resulting invariant. As for the proof of the *Two-Phase Commit*, we decompose the equation of the inductive invariant in such a way that each component will be a theorem to prove. The inductive invariant is defined in **Invariant** 10.

Invariant 10 (Inductive Invariant)**.**

$$Inv \triangleq TypeOk \wedge CoordInv$$

The set theorem. We introduce a theorem, $SetTheorem$ defined in **Theorem** 9, that sets the links between sets, similarly to **Theorem** 3. For the theorem citing $SetTheorem$, all definitions will be opaque. The theorem expresses, for example, the proof that an element in the set Pc cannot be found in the set $BSources$ or $BRecipients$, given that the latter contains the identifiers of the Byzantine participants and that Pc contains the identifiers of the correct participants. The theorem also proves the uniqueness of the identifiers of the different participants. Thus, ensuring that the identifier of the coordinator, publisher, sources and recipients cannot be equal. Moreover, the theorem proves the different equivalences between the following functions: $AofS()$, $AofR()$, $SofA()$, $RofA()$. To summarise, the theorem proves the following elements:

- A unique identifier for each participant, and abstracting arithmetic calculations for provers.
- Sets of assets, $AssetsFromCS$ and $AssetsForCR$ are subsets of $Assets$.
- Sets $Sources$ and $Recipients$ are the subsets of Pi. In addition, set $CSources$ and $CRecipients$ are subset of Pc (thereby, a subset of Pi). The set of correct participants and Byzantine participants are mutually exclusive.
- The equivalence between the helping function used to calculate the asset identifier from a source or a recipient (and vice versa).
- Set of participants states.

Theorem 9. Set Theorem

THEOREM $SetTheorem \triangleq$

$\land\ CoordinatorID \neq PublisherID$
$\land\ \forall\, a \in AssetsFromCS, b \in AssetsForCR : a \in Assets \land b \in Assets$
$\land\ \forall\, s \in Sources, r \in Recipients\ : s \in Pi \land r \in Pi$
$\land\ \forall\, p \in Pc : \land\, p \in Pi$
$\land\ \lor\ \land\, (p \in CSources \land p \in Sources)$
$\lor\ \land\, (p \in CRecipients \land p \in Recipients)$
$\land\ (p \notin BSources \land p \notin BRecipients)$
$\land\ \forall\, s \in CSources : \land\, (s \in Sources \land s \in Pi \land s \in Pc)$
$\land\ (s \notin BSources \land s \notin CRecipients \land s \notin Recipients$
$\land\ s \notin BRecipients)$
$\land\ (s \neq PublisherID \land s \neq CoordinatorID)$
$\land\ AofS(s) \in AssetsFromCS$
$\land\ \forall\, r \in CRecipients : \land\, (r \in Recipients \land r \in Pi \land r \in Pc)$
$\land\ r \notin BRecipients$
$\land\ (r \neq PublisherID \land r \neq CoordinatorID)$
$\land\ AofR(r) \in AssetsForCR$
$\land\ SofA(AofR(r)) \in Sources$
$\land\ \forall\, bs \in BSources : \land\, (bs \in Pi \land bs \in Sources)$
$\land\ (bs \notin CSources \land bs \notin Pc)$
$\land\ (bs \neq PublisherID \land bs \neq CoordinatorID)$
$\land\ AofS(bs) \notin AssetsFromCS$
$\land\ \forall\, br \in BRecipients : \land\, (br \in Recipients \land br \in Pi)$
$\land\ (br \notin Pc \land br \notin CRecipients)$
$\land\ (br \neq PublisherID \land br \neq CoordinatorID)$
$\land\ AofR(br) \notin AssetsForCR$
$\land\ ProcSet = \{PublisherID\} \cup \{CoordinatorID\} \cup (CSources)$
$\cup\ (BSources) \cup (CRecipients) \cup (BRecipients)$
$\land\ Pi = Sources \cup Recipients$
$\land\ Pc = Pi \cap Correct$
$\land\ CSources = Pc \cap Sources$
$\land\ CRecipients = Pc \cap Recipients$
$\land\ BSources = Sources \setminus CSources$
$\land\ BRecipients = Recipients \setminus CRecipients$
$\land\ BSources \cap CSources = \{\}$
$\land\ BRecipients \cap CRecipients = \{\}$
$\land\ AStates = \{$"OwS", "OwR", "locked", "other"$\}$
$\land\ CStates = \{$"init", "published", "okRM", "okRF"$\}$
$\land\ PStates = \{$"init", "publish"$\}$
$\land\ SwapStates = \{$"init", "correct", "different"$\}$
$\land\ \forall\, s \in Sources : SofA(AofS(s)) = s$
$\land\ \forall\, r \in Recipients : RofA(AofR(r)) = r$
$\land\ \forall\, a \in Assets : (AofS(SofA(a)) = a \land AofR(RofA(a)) = a)$
$\land\ \forall\, s \in Sources : AofS(s) \in Assets$
$\land\ \forall\, a \in Assets : SofA(a) \in Sources$

BY DEF $ProcSet, CSources, CRecipients, Sources, Recipients, AssetsFromCS, Assets, AssetsForCR, AofS, AofR, SofA, RofA, Pi, Pc, BSources, BRecipients, PublisherID, CoordinatorID, AStates, CStates, PStates, SwapStates$

Prove that Inv is true assuming $Init$ true. According to formula 7.1, the first component to prove is the one that states the initial conditions and is defined in **Theorem** 10.

Theorem 10. The theorem $Init \implies Inv$

```
THEOREM InitImpliesInv ≜
ASSUME Init
PROVE Inv
  ⟨1⟩ USE DEF Init, Inv, TypeOk, CoordInv
  ⟨1⟩1. TypeOk
    BY SetTheorem
  ⟨1⟩2. CoordInv
    BY SetTheorem DEF init_cInv
  ⟨1⟩3. QED
    BY ⟨1⟩1, ⟨1⟩2 DEF Inv
```

The theorem $InitImpliesInv$ has one level ⟨1⟩ since the proof is simple and does not need to be decomposed. The ⟨1⟩ USE DEF line expend the definitions to all the proof since definitions and facts must be cited explicitly for $TLAPS$ to use them. The way for that is by using the keyword BY and DEF . We prove the two conjunctions, $TypeOk$ and $CoordInv$, separately. They both use the theorem $SetTheorem$, to prove the membership of the elements in the different sets. For the $CoordInv$ proof, we need to cite the action $init\_cInv$, which corresponds to the initialisation action. The theorem must end with a QED step that asserts all the needed sub-proofs for the theorem proof.

Prove that Inv holds for arbitrary state transitions permitted by the predicate $Next$. The second component is the proof that if the invariant is true in any state of the behaviour, it is true in the next state of the behaviour, and the formula is $Inv \wedge Next \implies Inv'$. The formula needs to be decomposed into two distinct theorems—one for the conjunction of type correctness invariant and a second for the invariant coordinator conjunction.

The type correctness invariant theorem. Once the Inv invariant is decomposed according to its invariant member, we obtain for the type correctness part the following formula to prove:

$$TypeOk \wedge Next \implies TypeOk'$$

The theorem $TypeOkInvariant$, defined in **Theorem** 11, proves that at each iteration or behaviour of the system, the type of the variables remains unchanged. The first level, steps numbered ⟨1⟩, is a CASE step on participants (like a pattern-matching). The level ⟨1⟩1 states for the publisher, the level ⟨1⟩2 for the coordinator, the level ⟨1⟩3 for the correct sources, the level ⟨1⟩4 for the Byzantine sources, the level ⟨1⟩5 for the correct recipients, the level ⟨1⟩6 for the Byzantine recipients and the level ⟨1⟩7 for the terminating proof. All theorems constructed in cases must finish with a last QED step, e.g. the level ⟨1⟩8. When the proof is too complicated, it is necessary to decompose it until the provers are capable of proving it. For instance, the sub-proofs of the level 1 (except for ⟨1⟩1 and ⟨1⟩7) are decomposed into level ⟨2⟩. The sub-proofs of the second level iterate on the possible actions of the participants. Indeed, if we take the level ⟨1⟩2, which is the coordinator's case, we have four sub-proofs, and each corresponds to the four possible actions of the coordinator: $\{init\_c, decision, decisionValid, decisionAbort\}$. For the sake of clarity, we have deliberately omitted parts of the theorem that are repetitive. For example, the sub-proof ⟨2⟩1 has the same structure as ⟨2⟩2 with the exception that the $init\_c$ action must be replaced by the $decision$ action and the action is cited after the DEF . The same applies to the remaining two actions.

Theorem 11. The theorem $TypeOk \wedge Next \implies TypeOk'$

```
THEOREM TypeOkInvariant ≜
ASSUME TypeOk, Next
PROVE TypeOk'
  ⟨1⟩USE DEF TypeOk
  ⟨1⟩1.CASE Publisher
    BY ⟨1⟩1, SetTheorem DEF Publisher, init_p
  ⟨1⟩2.CASE Coordinator
    ⟨2⟩1.CASE init_c
      BY ⟨2⟩1, ⟨1⟩2, SetTheorem DEF init_c
    ...
    ⟨2⟩5. QED BY ⟨1⟩2, ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4 DEF Coordinator
  ⟨1⟩3.CASE ∃ self ∈ CSources : Source(self)
    ⟨2⟩ SUFFICES ASSUME NEW self ∈ CSources, Source(self)
                 PROVE  TypeOk'
      BY ⟨1⟩3
    ⟨2⟩1.CASE init_src(self)
    ⟨2⟩2.CASE lock(self)
    ...
    ⟨2⟩5. QED BY ⟨1⟩3, ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4 DEF Source

  ⟨1⟩4.CASE ∃ self ∈ BSources : BSource(self)
    ⟨2⟩ SUFFICES ASSUME NEW self ∈ BSources, BSource(self)
                 PROVE  TypeOk'
      BY ⟨1⟩4
    ⟨2⟩1.CASE init_bsrc(self)
    ...
    ⟨2⟩8. QED BY ⟨1⟩4, ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7 DEF BSource

  ⟨1⟩5.CASE ∃ self ∈ CRecipients : Recipient(self)
      ⟨2⟩ SUFFICES ASSUME NEW self ∈ CRecipients, Recipient(self)
                   PROVE  TypeOk'
      BY ⟨1⟩5
    ⟨2⟩1.CASE init_rcp(self)
    ...
    ⟨2⟩3. QED BY ⟨1⟩5, ⟨2⟩1, ⟨2⟩2 DEF Recipient
  ⟨1⟩6.CASE ∃ self ∈ BRecipients : BRecipient(self)
      ⟨2⟩ SUFFICES ASSUME NEW self ∈ BRecipients, BRecipient(self)
                   PROVE  TypeOk'
      BY ⟨1⟩6
    ⟨2⟩1.CASE init_brcp(self)
    ...
    ⟨2⟩6. QED BY ⟨1⟩6, ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5 DEF BRecipient
  ⟨1⟩7. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6 DEF Next
```

The coordinator invariant theorem. The second part of the **Invariant** 10 is the coordinator invariant. The theorem we want to prove is that:

$$CoordInv \wedge TypeOk \wedge Next \implies CoordInv'$$

We add the type correctness to facilitate the proof of the coordinator invariant. The resulting theorem is defined in **Theorem** 12.

Theorem 12. The Coordinator Invariant

THEOREM $CoordInvariant \triangleq$
  ASSUME $CoordInv$, $TypeOk$, $TypeOk'$, $Next$
  PROVE $CoordInv'$
⟨1⟩USE DEF $TypeOk$, $CoordInv$
⟨1⟩1.CASE $pc[CoordinatorID] =$ "init_c"
  …
⟨1⟩2.CASE $pc[CoordinatorID] =$ "decision"
  …
⟨1⟩3.CASE $pc[CoordinatorID] =$ "decisionValid"
  …
⟨1⟩4.CASE $pc[CoordinatorID] =$ "decisionAbort"
  …
⟨1⟩5.CASE $pc[CoordinatorID] =$ "Done" ∧ $coordState =$ "okRM"
  …
⟨1⟩6.CASE $pc[CoordinatorID] =$ "Done" ∧ $coordState =$ "okRF"
  …
⟨1⟩7. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6

As with the type correctness theorem, the structure of the $CoordInv$ proof is decomposed into several levels. The decomposition is performed according to the conjunctions of the **Invariant** 3, i.e. $CoordInv$. The latter comprises six conjunctions; thus, the first level of the theorem will be divided into six steps to prove. The proof begins with the extension of the $TypeOk$ and $CoordInv$ definitions via the USE DEF to lighten the structure of the theorem.
For each case of level ⟨1⟩1, there will be sub-proofs of a higher level than ⟨1⟩1. A proof is read from the lowest to the highest level. Each theorem step is independent and must be provable with only the information provided by the step. We describe the proof pattern for the case ⟨1⟩1, which is $pc[CoordinatorID] =$ "init_c", to illustrate the construction of the proof.

Description of the level ⟨1⟩1. The first step of the first level assumes the first conjunction of the $CoordInv$ invariant to be true, which is $pc[CoordinatorID] =$ "init_c". The second step of the first level assumes the second conjunction of the $CoordInv$ invariant to be true, and so on.

The first level needs to be decomposed because the step is too complicated to prove by the provers. With the TLA$^+$ toolbox, we can choose which definition we want to decompose. The definitions that can be decomposed are the predicates that we assume to be true, which in our case can be $TypeOk$, $CoordInv$ and $Next$. The first level has been decomposed according to the $CoordInv$ predicate; hence, for the second, we choose to decompose the $Next$ predicate. The split gives us seven cases of level ⟨2⟩, which correspond to the seven possible disjunctions of the $Next$ predicate: $Publisher$, $Coordinator$, $Source(self)$, $BSource(self)$, $Recipient$, $BRecipient(self)$, $Terminating$. The first case to prove is that of the $Publisher$. Step ⟨2⟩1 assumes that $Publisher$ is true and attempts to prove $init\_cInv$. The information needed for the proof of step ⟨2⟩1 is the context of step ⟨1⟩1 and its own context. Then, after the DEF keyword, the step cites the definition it tries to prove and the publisher's information, i.e. its behaviour definition and its initial action.

```
⟨1⟩1.CASE pc[CoordinatorID] = "init_c"
  ⟨2⟩1.CASE Publisher
    BY ⟨1⟩1, ⟨2⟩1 DEF Publisher, init_p, init_cInv
  ⟨2⟩2.CASE Coordinator
    ⟨3⟩1.CASE init_c
     BY ⟨3⟩1 DEF init_c, init_cInv, decisionInv
    ⟨3⟩2.CASE decision
     BY ⟨1⟩1, ⟨3⟩2 DEF decision
          ...
    ⟨3⟩5. QED BY ⟨2⟩2, ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4 DEF Coordinator
  ⟨2⟩3.CASE ∃self ∈ CSources : Source(self)
    ⟨3⟩USE DEF init_cInv
    ⟨3⟩SUFFICES ASSUME NEW self ∈ CSources, Source(self)
                PROVE CoordInv'
     BY ⟨2⟩3
    ⟨3⟩1.CASE init_src(self)
     BY ⟨3⟩1, ⟨1⟩1, SetTheorem DEF init_src
          ...
    ⟨3⟩5.QED BY ⟨2⟩3, ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4 DEF Source
  ⟨2⟩4. CASE ∃self ∈ BSources : BSource(self)
    ⟨3⟩USE DEF init_cInv
    ⟨3⟩SUFFICES ASSUME NEW self ∈ BSources, BSource(self)
                PROVE CoordInv'
     BY ⟨2⟩4
    ⟨3⟩1.CASE init_bsrc(self)
     BY ⟨3⟩1, ⟨1⟩1, SetTheorem DEF init_bsrc
          ...
    ⟨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 ∃self ∈ CRecipients : Recipient(self)
    ⟨3⟩USE DEF init_cInv
    ⟨3⟩SUFFICES ASSUME NEW self ∈ CRecipients, Recipient(self)
                PROVE CoordInv'
     BY ⟨2⟩5
    ⟨3⟩1.CASE init_rcp(self)
     BY ⟨3⟩1, ⟨1⟩1, SetTheorem DEF init_rcp
          ...
    ⟨3⟩3.QED BY ⟨2⟩5, ⟨3⟩1, ⟨3⟩2 DEF Recipient
  ⟨2⟩6. CASE ∃self ∈ BRecipients : BRecipient(self)
    ⟨3⟩USE DEF init_cInv
    ⟨3⟩SUFFICES ASSUME NEW self ∈ BRecipients, BRecipient(self)
                PROVE CoordInv'
     BY ⟨2⟩6
    ⟨3⟩1.CASE init_brcp(self)
     BY ⟨3⟩1, ⟨1⟩1, SetTheorem DEF init_brcp
          ...
    ⟨3⟩6.QED BY ⟨2⟩6, ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4, ⟨3⟩5 DEF BRecipient
  ⟨2⟩7. QED BY ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6 DEF Next
```

The second case of the level ⟨2⟩ is for the coordinator. Step ⟨2⟩2 assumes that one of the coordinator's actions is executed. The coordinator has four possible actions to execute; therefore, the step ⟨2⟩2 is split into four sub-steps at a higher level, the level ⟨3⟩. Step ⟨3⟩1 assumes the *init_c* action is

true. This step is proved by sending the context of step ⟨3⟩1 to the provers and citing the definitions needed for the proof.
Since proofs are constructed hierarchically and in levels, a higher level step usually cites the step from which it originates. However, it is possible not to do this if calling a step does not provide additional information to the proof. For example, the case ⟨3⟩1 is self-sufficient and does not cite the steps that precede it, namely steps ⟨2⟩2 and ⟨1⟩1. The reason is that step ⟨1⟩1 assumes that the coordinator's program control is at "init_c", yet step ⟨3⟩1 assumes that $init\_c$ is at true, which amounts to the same thing. Therefore, we can omit citing step ⟨1⟩1 and thus lighten the proof obligation. Moreover, citing step ⟨2⟩2 is unnecessary since assuming that the action $init\_c$ is true implies that it is the coordinator who performs an action, so it is no longer necessary to cite step ⟨2⟩2. Following this proof logic, we have constructed the rest of the theorem. For instance, the following steps, which are ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4, need in their context the step ⟨1⟩1.

The third step of the second level ⟨2⟩3 concerns the correct sources. We decompose the step according to the disjunctions of the $Source(self)$ predicate, which gives us four sub-proofs of a higher level. These steps correspond to the actions of a correct source. We start with step ⟨3⟩ by extending the definition of the predicate $init\_cInv$ to the entire level ⟨3⟩. The next statement is a SUFFICES step asserting that to prove the goal of ⟨2⟩3 it suffices to assume a new variable $self$ from the set $CSources$ that satisfies $Source(self)$ and prove $CoordInv'$. The two facts are unnamed; thus, the backend provers use them without being mentioned in a BY step.
What follows is a step proof for each source's actions. Step ⟨3⟩1 assumes that the action $init\_src(self)$ is true. This step is the only one that requires citing the $SetTheorem$. The reason is that, at this level of the proof, i.e. assuming that the coordinator's program control is on "init_c" (without having executed the action $init\_c$), the action $init\_src(self)$ is the only one that can be true. The remaining cases, ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4 are by definition false. Therefore, it is not necessary to cite $SetTheorem$ because provers do not need to prove a step that is false.

The proof of the following cases, i.e. ⟨2⟩4,⟨2⟩5, ⟨2⟩6, which corresponds to Byzantine sources, correct recipients and Byzantine recipients, is similar to case ⟨2⟩3. Therefore, it is not necessary to explain these steps in more detail. The same applies to the steps ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, and ⟨1⟩6, which use the same proof approach as the step ⟨1⟩1 The complete proof of the invariant is in the following GitHub link [1].

We define a **Theorem** 13 that gathers the **Theorem** 11 and **Theorem** 12 to have a single theorem that expresses the second component of the formula 7.1.

Theorem 13. The Inductive Invariant

THEOREM $InvInvariant \triangleq$
  ASSUME $Inv, Next$
  PROVE Inv'
BY $TypeOkInvariant, CoordInvariant$ DEF Inv

The theorem $InvInvariant$ assumes Inv and $Next$ to be true and proves the next possible states of Inv'. We provide to the provers the theorems $TypeOkInvariant$ and $CoordInvariant$ by citing them after the BY step.

Prove that the $Consistency$ is true in all reachable states. The third component of formula 7.1 is $Inv \implies Consistency$. Although $Consistency$ is an invariant of the algorithm, it is not an inductive invariant. For proving the invariance of $Consistency$, it suffices to assume Inv. The inductive

[1] https://github.com/ZeinabYeong/ICDCN22/blob/main/CrossChain.tla

invariant *Inv* is true in all reachable states, which implies that *Consistency* is true in all reachable states, so it is an invariant. The proof of the *Consistency* invariant is defined in **Theorem** 14.

Theorem 14. Invariant Implies Consistency

```
THEOREM InvImpliesConsistency ≜
    ASSUME Inv
    PROVE Consistency
  ⟨1⟩ USE DEF CoordInv, Inv, Consistency
  ⟨1⟩1.CASE pc[CoordinatorID] = "init_c"
    BY ⟨1⟩1 DEF init_cInv, AvailableS
  ⟨1⟩2.CASE pc[CoordinatorID] = "decision"
    BY ⟨1⟩2 DEF decisionInv, Finish, AvailableS
  ⟨1⟩3.CASE pc[CoordinatorID] = "decisionValid"
    BY ⟨1⟩3 DEF decisionValidInv, Finish
  ⟨1⟩4.CASE  pc[CoordinatorID] = "decisionAbort"
    BY ⟨1⟩4 DEF decisionAbortInv, Finish, AvailableS
  ⟨1⟩5.CASE  (pc[CoordinatorID] = "Done" ∧ coordState = "okRM")
    BY ⟨1⟩5 DEF okRMInv, AvailableR
  ⟨1⟩6.CASE  (pc[CoordinatorID] = "Done" ∧ coordState = "okRF")
    BY ⟨1⟩6 DEF okRFInv, AvailableS
  ⟨1⟩7. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6 DEF TypeOk
```

We decompose the theorem according to the *CoordInv* invariant, and we get six steps of level ⟨1⟩. Each one corresponds to a conjunction of the invariant *CoordInv*. Each step must cite its corresponding *CoordInv*'s sub-invariant. Thus, step ⟨1⟩1 cites the definition *init_cInv* and also *AvailableS*. We need to expand the definition *AvailableS* because the provers must evaluate if this predicate is true since a source can reach the end of its program when the coordinator is at "init_c". Thereby, *AvailableS* can be evaluated as true. This is not the case for step ⟨1⟩3, which does not require extending the *AvailableS* predicate. Indeed, if we look at the invariant *decisionValidInv*, all the system assets are in "locked" state. This implies that the *AvailableS* (but also *AvailableR*) predicate is set to false. Therefore, the provers do not need to evaluate this predicate. On the other hand, *Finish* needs to be extended because a recipient may have its program control set to "Done"; thus, the provers need to evaluate the predicate.

Breaking down the proof into sub-proofs provides a better understanding of how the proof system in *TLAPS* works. For example, what definition a step should cite or what context provers need to prove a step. For this reason, we have decomposed the *InvImpliesConsistency* theorem to explain the proof logic. Indeed, the theorem can be written in a much more simplified way, defined in **Theorem** 15.

Theorem 15. Simplified Theorem

```
THEOREM InvImpliesConsistency ≜
    ASSUME Inv
    PROVE Consistency
  BY DEF TypeOk, CoordInv, Consistency, Inv, okRFInv, AvailableS, okRMInv, AvailableR,
  decisionAbortInv, Finish, decisionValidInv, decisionInv, init_cInv
```

Prove the *Consistency* **property.** The last step is to prove the safety property. Proving that *Consistency* is an invariant of our system requires proving the three components of formula 7.1. The theorems 10, 13, and 15 prove, respectively, the formulas: $init \implies Inv$, $Inv \land [Next]_{vars} \implies Inv'$, and $Inv \implies Consistency$. **Theorem** 16 is the theorem that proves the invariance of the *cross-chain swap* algorithm. Assuming the three steps, which are ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3, to be true, we can ensure that the safety property of *Consistency* is always true for all reachable states of the system.

Theorem 16. Invariance Proof of *Algorithm cross-chain swap*

THEOREM $Safety \triangleq Spec \implies \Box Consistency$
⟨1⟩1. $Init \implies Inv$
  BY $InitImpliesInv$
⟨1⟩2. $Inv \land [Next]_{vars} \implies Inv'$
  ⟨2⟩1.CASE $Next$
    BY ⟨2⟩1, $InvInvariant$
  ⟨2⟩2.CASE UNCHANGED $vars$
    BY ⟨2⟩2 DEF $vars$, Inv, $TypeOk$, $CoordInv$, $okRFInv$, $okRMInv$, $decisionAbortInv$, $decisionValidInv$, $init\_cInv$, $decisionInv$
  ⟨2⟩3. QED BY ⟨2⟩1, ⟨2⟩2
⟨1⟩3. $Inv \implies Consistency$
  BY $InvImpliesConsistency$ DEF Inv
⟨1⟩4. QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, PTL DEF $Spec$

We cite the PTL backend (Propositional Temporal Logic) in the QED step since the theorem contains temporal logic symbols (□ describing the always symbol). We have launched the prover on the module, and it takes less than 8 minutes to prove the model.

7.2 Proof of the Liveness Properties

In this section, we verify the liveness properties of *Ownership* and *Retrieving*. As for the *Consistency* property, we apply two verification approaches. The first is a handwritten proof, and the second is a verification by model-checking using the toolbox of TLA$^+$. As a recall, the liveness properties of the *cross-chain swap* problem, introduced in Section 6.2, are defined as follows:

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*".

7.2.1 Handwritten Proof of the *Ownership* and *Retrieving* Properties

In this section, we prove manually the liveness properties defined in Section 6.2. The properties are expressed over time in LTL formulas. From **Lemma** 7.1, **Lemma** 7.2, **Lemma** 7.3, and **Lemma** 7.4, we prove that the coordinator's decisions are mutually exclusive and a correct participant is never worse off. Please refer to the previous Section 7.1.1 for the symbols' definition.

Lemma 7.6. *If "Proof$_{publish}$" is valid and at least one participant is correct, then the coordinator eventually makes a decision.*

Formally: $Proof_{publish} \wedge \Pi_c \neq \{\} \implies \Diamond(loc(c) = \text{“OKRF"} \vee loc(c) = \text{“OKRM"})$

Proof. From **Lemma** 7.2, we have proven that the coordinator in the state "PUBLISHED" satisfies the *proof-of-action Proof$_{publish}$*. From Figure 6.4, we can see that after being published, 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. Suppose the coordinator is in the "PUBLISHED" state for a while without evolving. It suffices to have only one correct participant to ensure the system's evolution and exit from the blocking state. Assuming this scenario, the correct participant, whether the source (Figure 6.5) or the recipient (Figure 6.6), must be in the "WAITFORD" state. After reaching the participant's timeout, the predicate NoDecision() will be satisfied ($\sigma_5^{s_i}$ if the participant is a source and $\sigma_4^{r_i}$ if the participant is a recipient). The validated predicate allows the participant to request a *refund* decision from the coordinator ($\epsilon_5^{s_i}$ or $\epsilon_4^{r_i}$). The operation of asking *refund* satisfies the predicate AbortTransfer() and leads to the coordinator's decision for a *refund* authorisation. Moreover, if all participants are correct, then all sources will lock their assets and give a valid "*Proof$_{lock}$*" to the coordinator. Hence, conditions of the ValidTransfer predicate will be satisfied and lead to a *redeem* authorisation from the coordinator. □

Lemma 7.7. *If the coordinator authorises the refund, then no asset initially owned by a correct source is ownerless forever.*

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

Proof. If the coordinator authorises the *refund*, the predicate AbortTransfer() has been satisfied (**Lemma** 7.4). As a result, assets ownership return to their sources ($\epsilon_2^{a_i}$ is satisfied); hence all assets initially owned by a correct source become available to their source. A correct source will retrieve the *proof-of-action* "*Proof$_{refund}$*" from the coordinator. A valid proof satisfies $\sigma_6^{s_i}$, and a correct source will be able to recover its assets by executing $\omega_6^{s_i}$ and become the owner again. If the source is Byzantine, it might never recover its asset, thus leaving the asset ownerless. In addition, the Byzantine source could lock its asset out of the swap with no way to recover it. These two situations are acceptable and satisfy the property. □

Lemma 7.8. *If the coordinator authorises the redeem, then no asset intended for a correct recipient is ownerless forever.*

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

Proof. If the coordinator authorises the *redeem*, then the predicate ValidTransfer has been satisfied (see **Lemma** 7.3). In the *redeem* case, all assets become available to the recipient. A correct recipient will retrieve the *proof-of-action* "*Proof$_{redeem}$*" from the coordinator. A valid proof satisfies $\sigma_5^{r_i}$, and a correct recipient only has to retrieve the asset by executing $\omega_5^{r_i}$ and updating the state of the asset to "OWS" ($\epsilon_3^{a_i}$). However, if a Byzantine recipient decides not to get its asset back, then that asset will be ownerless. It is an acceptable situation and satisfies the property. □

Theorem 7.9. *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** 7.6, we have proven that if "*Proof$_{publish}$*" is valid, it only takes one correct participant in our system for the coordinator to issue a decision. Moreover, if all participants are Byzantine, the theorem is still satisfied. From **Lemma** 7.7, we have proven that correct sources will not lose their asset. However, no conclusions are possible for assets owned by Byzantine sources. From **Lemma** 7.8, the same assumption has been proven for assets intended for correct recipients. Likewise, no conclusions are possible for assets intended for Byzantine recipients. As a result, we have proven the *Ownership* property of the swap. □

Theorem 7.10. *If all participants are correct, then all recipients will retrieve their intended assets.*

Proof. If all participants are correct, 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 "$Proof_{redeem}$". □

7.2.2 Proof of the *Ownership* and *Retrieving* Properties Using TLC

In this part, the model checker TLC is applied to the swap model, $\mathscr{P}_{swap}$, to verify the liveness properties. In the case of model-checking, it is necessary to set a value to the constants defined in our model. As a reminder, the constants are $NTxs$; the number of transactions in our swap, $Correct$ the set of correct participants involved in the swap, and $Timeout$; the constant that describes whether the participants can timeout.

The *Ownership* property. The property is defined in TLA$^+$ formalism in **Definition** 40 [2].

Definition 40 (The *Ownership* Propertiy)**.**

$$Ownership \triangleq AtLeastOneCorrect \rightsquigarrow (OwnershipS \vee OwnershipR)$$

This property ensures that it is sufficient to have at least one correct participant so that the assets of the correct ones are not locked forever, i.e. in a locked state and without any possibility of using them. The expression of at least one correct participant is defined in **Definition** 41, and it expresses that the set P_c, which is the correct sets, must not be equal to the empty set and thus must contain at least one element. The predicate is true if the condition is verified.

Definition 41 (At least one participant is correct)**.**

$$AtLeastOneCorrect \triangleq Pc \neq \{\}$$

Not being locked means that the assets will eventually be either in possession of a source or in possession of a recipient. These expressions are represented by the definitions $OwnershipS$ and $OwnershipR$, respectively defined in **Definition** 42 and **Definition** 43.

Definition 42 (The ownership of a source asset)**.**

$$OwnershipS \triangleq \forall\, a \in AssetsFromCS : AvailableS(a)$$

The definition says that all assets from the set $AssetsFromCS$, which correspond to the set of assets initially owned by correct sources, satisfy the predicate $AvailableS$ defined in **Definition** 38. Saying that the asset belongs to the set $AssetsFromCS$ excludes assets belonging to Byzantine sources. It is not possible to predict the state of these assets since it is impossible to predict the behaviour of a Byzantine source. As a result, the predicate only targets assets from correct sources.

[2] Note that $(A \rightsquigarrow B)$ is "syntactic sugar" for $\Box(A \implies \Diamond B)$, with the temporal operators $\Box$; *always*, $\Diamond$; *eventually* and $\implies$; *implies*.

Definition 43 (Expressing the ownership of a recipient asset)**.**

$$OwnershipR \triangleq \forall\, a \in AssetsForCR : AvailableR(a)$$

The second definition says that all assets from the set $AssetsForCR$, which correspond to the set of assets intended for correct recipients, satisfy the predicate $AvailableR$ defined in **Definition** 38. As with **Definition** 42, one can assert the state of assets that are intended for the correct recipients, hence the use of the $AssetsForCR$ set. Byzantine recipients may never get back an asset due to them or may give it to others immediately after receiving it.

The *Retrieving* property. The property is defined in TLA$^+$ formalism in **Definition** 44.

Definition 44 (The *Retrieving* Properties)**.**

$$Retrieving \triangleq AllParticipantsAreCorrect \rightsquigarrow (\forall\, r \in Recipients : assets[AofR(r)] = \text{“OwR”})$$

This property ensures that if all participants are correct, then the expected result is to have made the swap and therefore, the asset transfer must occur. If the transfer takes place, all assets must be in the “OwR" state, reflecting the state owned by recipient. In this case, we have the set $Recipients$ and $CRecipients$ equal, and the set $BRecipients$ is empty. The expression, which represents that all participants are correct, is defined in **Definition** 45.
$AllParticipantsAreCorrect$ is a true predicate if all participants (sources, recipients and publisher) are correct. The statement $swapGraph =$ “correct" describes that the publisher is correct.

Definition 45 (Expressing all participants are correct)**.**

$$AllParticipantsAreCorrect \triangleq (Pi = Pc) \wedge swapGraph = \text{“correct”}$$

Model-checking of $\mathscr{P}_{swap}$. Once the liveness properties and definitions are well established, we can launch the model checker to the model. TLC is multithreaded and can take advantage of multiprocessors. Traditional model-checking works on finite-state specifications—specifications with an a priori upper bound on the number of reachable states. TLC explores reachable states, looking for the violation of an invariant or for deadlock occurrence—meaning that there is no possible next state. TLC stops when it has examined all states reachable by traces that contain only states satisfying the constraint. The power of model-checking allows exploring even infinite executions over finite state spaces. From the invariants' point of view, the model checker's invariant is that all reachable states of the program are either in the visited set *or* queued to be visited later *or* reachable from some queued state. When the queue is empty, all reachable states have been visited. Suppose the model-checking process exceeds the threshold of running time or memory, usually due to state-space explosion. In that case, the model checker is typically interrupted, and the model is supposed to be reduced.

Running the model checker from the TLA$^+$ toolbox. Running the model checker can be done via a command line or the TLA$^+$ toolbox. Figure 7.1 is a screenshot of the toolbox user interface. To launch the model checker, the user must click on the green button at the top left. A *General* section in the interface gives us the date and time we started the checking. Just below, we have a set of

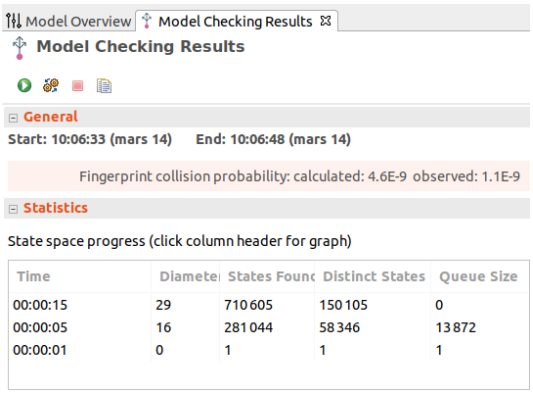


Figure 7.1 – Screenshot of the TLA$^+$ toolbox

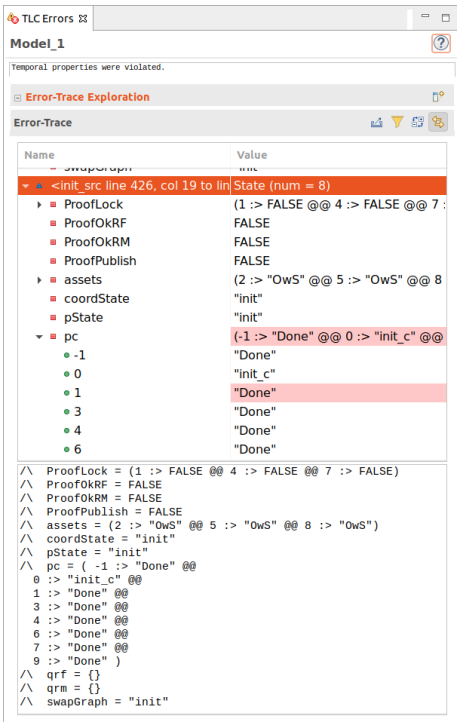


Figure 7.2 – Example of a counter-example

statistical data about the result of the checking. From left to right, we have the *Time* taken to check the property; the *Diameter* gives us the graph's diameter or depth. Remember, TLC transforms the model into a graph. In graph theory, the diameter of a graph is the largest possible distance between two of its vertices, the distance between two vertices being defined by the length of the shortest path between them. We also find the number of *States found* and the *Distinct states* among the statistics. The number of states could depend on the unspecified parameter; $NTxs$ and $Correct$. Finally, *Queue Size* gives the number of states not yet explored. If TLC has successfully checked the property, the toolbox returns the time taken by the check. Conversely, if a property is violated, then TLC returns a counter-example, as shown in Figure 7.2. When a variable is modified, the variable is highlighted in red. The counter-example simulates the trace from the initial state to the state violating the property. When the model takes a long time to be checked, there are tricks to reduce the time. For example, in the TLC options, it is possible to disable the profiling option. Profiling helps to identify specification errors such as permanently disabled actions. Similarly, it helps identify the source of state-space explosion by reporting the states found and distinct states on a per action level. Profiling negatively impacts model checking performance and should be disabled when checking large models.

Results interpretations. We run the model checker on a model that ensures the transfer of 3 assets, $NTxs = 3$, involving 6 participants (3 sources and 3 recipients). While the number of transfers remains unchanged throughout our calculations, the share of correct participants varies. $Correct$ is a set, and its elements are the correct participant identifiers. If the set is empty, $Correct = \{\}$, then it means that all sources and recipients turn out to be Byzantine participants. The *Ownership* property does not require synchrony to be satisfied; thus, the constant $Timeout$ is set to TRUE. Conversely, the *Retrieving* property requires synchrony; thus, the constant $Timeout$ is set to FALSE.

Table 7.1 gives some results concerning the model-checking carried out on $\mathscr{P}_{swap}$ for the *Ownership* and *Retrieving* property. The results of the liveness verification are according to the proportion of Byzantine participants in the model. Accordingly, we have four tables corresponding to the number of Byzantine sources in the system. Table 7.2 gives the results with no Byzantine

sources, Table 7.3 with one Byzantine source, Table 7.4 with two Byzantine sources, and Table 7.5 with three Byzantine sources, i.e. all the sources. Each table gives the *Ownership* and *Retrieving* verification results according to the number of Byzantine recipients in the model (second column). The tables give the time of checking, the depth of the generated graph, the generated distinct states, and the last column gives information about the result of the checking.

Table 7.2 gives the checking results without considering Byzantine sources in the model. The computation time for both the *Ownership* and the *Retrieving* does not exceed $6min$, which is relatively fast. The number of generated distinct states is also low. The model checker manages to verify the model without any optimisation. However, we can see from the different tables that the verification time and the generated distinct states increase as the share of Byzantine participants in the model increases. In addition, this observation is even more true as the share of Byzantine sources increases in the model. We also notice that, although the number of Byzantine recipients increases the verification time, it seems negligible compared to the influences of Byzantine sources. If we compare the row with 1 Byzantine recipient in Table 7.2 and the row with 0 Byzantine recipient in Table 7.3. In both cases, the system has an equal number of Byzantine participants, only one participant. Nevertheless, the verification time is almost four times longer considering the Byzantine participant as a source than if it were a recipient. The diameter is also larger, and there are 1766184 more distinct states. We conclude that recipients do not have as much impact on the system as sources. Indeed, the behavioural model of a Byzantine recipient in TLA$^+$ has less actions than a Byzantine source (see Section 6.5.3). The more actions a process defines, the more possible states are generated, increasing the verification time.

Moreover, the verification time of *Retrieving* is faster than *Ownership* since the former requires synchrony, and the constant $Timeout$ is set to FALSE. Indeed, considering a model where the participants do not timeout disables the possibility of satisfying the predicate ¬CorrectSwap and NoDecision from the Figures 6.5 and 6.6. Consequently, the model is reduced, which decreases the number of generated distinct states.

The increase in the number of states in the system increases the memory allocation, and if we face a run out of memory, the model-checking stops. The calculations of the first three tables (7.2, 7.3 and 7.4) were possible via the TLA$^+$ toolbox on a computer Intel® Core™ i7-8850H CPU @ 2.60GHz × 12 (except for the case "Byzantine sources = 2 and Byzantine recipients = 3" from Table 7.4, for both *Ownership* and *Retrieving* property). The computer had enough memory to handle the generated number of distinct states. However, the cases in Table 7.5 and the case "Byzantine sources = 2 and Byzantine recipients = 3" from 7.4 have generated a large number of states requiring a larger quantity of memory. As a result, those cases have been checked using the TLC command-line `java -cp tla2tools.jar tlc2.TLC CrossChain.tla` on a server having more memory allocation. The last Table 7.5, gives the model-checking results considering all the sources are Byzantine. Although the used server had more memory than the computer, the case where all sources are Byzantine could not be verified (except for the two first cases of *Retrieving*). TLC has produced an enormous file to store the state exploration queue and exceeds the memory storage.

Table 7.1 – Model-checking results

Table 7.2 – Model-checking results with 6 participants, including 0 Byzantine sources

Properties	Number of Byzantine recipients	Time	Depth	Distinct States	Result
Ownership	0	$22s$	29	150105	validated
	1	$38s$	33	385175	validated
	2	$1min15$	37	1248225	validated
	3	$05min10$	41	5482375	validated
Retrieving	0	$02s$	23	867	validated
	1	$12s$	28	23010	validated
	2	$26s$	33	142650	validated
	3	$05min54s$	38	956500	validated

Table 7.3 – Model-checking results with 6 participants, including 1 Byzantine source

Properties	Number of Byzantine recipients	Time	Depth	Distinct States	Result
Ownership	0	$02min22s$	35	2151359	validated
	1	$05min45$	35	4771585	validated
	2	$15min38$	39	11683175	validated
	3	$43min39$	43	35268625	validated
Retrieving	0	$30s$	30	123494	validated
	1	$01min04s$	31	648550	validated
	2	$03min03s$	31	648550	validated
	3	$01h12min$	41	13980750	validated

Table 7.4 – Model-checking results with 6 participants, including 2 Byzantine sources

Properties	Number of Byzantine recipients	Time	Depth	Distinct States	Result
Ownership	0	$28min32$	41	32606609	validated
	1	$06h39min39$	41	69945295	validated
	2	$10h07min37$	41	158633825	validated
	3	$2\ days$	45	405517875	validated
Retrieving	0	$23min27$	37	4078466	validated
	1	$01h52min28s$	38	17111780	validated
	2	$06h23min$	39	68974850	validated
	3	$2days$	40	611282250	validated

Table 7.5 – Model-checking results with 6 participants, including 3 Byzantine sources

Properties	Number of Byzantine recipients	Time	Depth	Distinct States	Result
Ownership	0	–	–	–	run out of memory
	1	–	–	–	run out of memory
	2	–	–	–	run out of memory
	3	–	–	–	run out of memory
Retrieving	0	$10h12min56$	44	122382400	validated
	1	$21h51min36$	45	500112074	validated
	2	–	–	–	run out of memory
	3	–	–	–	run out of memory

7.3 Conclusion

In this chapter, we apply formal verification tools to the swap model $\mathscr{P}_{swap}$ to verify if the model satisfies the specification of a *cross-chain swap* problem. The first Section 7.1, deals with verifying the safety property – the *Consistency*. The verification approach is the one described in Chapter 4 using $TLAPS$, and we have shown that the methodology was perfectly adapted to the $\mathscr{P}_{swap}$ algorithm. Using the deductive approach allows proving the system without limiting the number of participants, which is a great advantage. *Consistency* has been proven in a system including Byzantine participants. The two remaining properties are liveness, verified using the model-checking approach. TLC was able to verify that the models with less than three Byzantine sources satisfy the liveness properties, but not the models with three Byzantine sources. Those cases generate an enormous number of states. In addition, the verification time can be very long for some models (2 days for the model with 3 Byzantine recipients and 2 Byzantine sources) if we compare it with the time taken by $TLAPS$, which is less than 8 minutes. Considering using $TLAPS$ on liveness properties would be interesting. Unfortunately, $TLAPS$ is not suited for proving eventually temporal logic.

Chapter 8

Analysis of $\mathscr{P}_{swap}$ Instantiation in a Blockchain Environment

" However bad life may seem, there is always something you can do and succeed at. "

— Stephen Hawking

Contents

The last step of our study is to show that the protocol introduced in Chapter 6 and proved in Chapter 7 is quite adapted to blockchain systems. This chapter describes how we instantiate the participants, the actions and other primitives described in earlier chapters in a blockchain environment. We discuss how the specifics of blockchains can be exploited to meet the requirements of the $\mathscr{P}_{swap}$ protocol (Section 8.1). Furthermore, in the second part of the chapter, we analyse how the protocol can be applied or adapted depending on the types of blockchain introduced in Chapter 2 (Section 8.2). As a recall, the different types of blockchains are public permissionless, public permissioned, private permissionless and private permissioned.

8.1 $\mathscr{P}_{swap}$ in a Blockchain Environment

We have described a $\mathscr{P}_{swap}$ protocol that is voluntarily agnostic of all blockchain systems references not to limit possible use cases. However, this thesis focuses on studying blockchain systems; hence, in this section, we show how the protocol could be instantiated within such an environment. Therefore, we present the protocol $\mathscr{P}_{inst}$, an instantiation of $\mathscr{P}_{swap}$. The goal of $\mathscr{P}_{inst}$ is to transfer assets between different participants of different blockchains. Every action and primitive described in the abstract protocol applies to the $\mathscr{P}_{inst}$ protocol with the addition of blockchain-specific implementations. After recalling the requirements of $\mathscr{P}_{swap}$, we describe which blockchain specificities can fulfil the requirements. We set four requirements for the instantiated protocol to satisfy the abstract protocol as follows:

1. The swap graph must be public and available to all the participants.
2. The coordinator must be public and its correctness verifiable.
3. The asset must be locked.
4. The mechanism of *proof-of-action* must be trustworthy.

8.1.1 $\mathscr{P}_{inst}$ Outline and Instantiation of $\mathscr{P}_{swap}$ Requirements

The participants of $\mathscr{P}_{inst}$ consist of sources, recipients, a publisher, and a coordinator. They are assigned a public and a private key to each (except the coordinator). As for the abstract protocol, their behaviour is defined by their state machine depicted, respectively, in Figures 6.5, 6.6, 6.3 and 6.4 of Chapter 6.
Before beginning the swap, the participants agree on the transfers and create the swap graph that will serve as an input of the protocol. Sources are participants who own the asset to be transferred, and recipients are those who will receive the assets. The publisher is the one who makes the swap graph public, and the coordinator is the one who provides the decision to *redeem* or *refund* the assets. The source's private key allows it to access its wallet, where its assets are stored. A source may own several assets stored on different blockchains, e.g. an asset a_A is stored in wallet W_A on blockchain A, and an asset a_B is stored in wallet W_B on blockchain B, with both wallets W_A and W_B belonging to the source.

8.1.2 How Can the Swap Graph and the Coordinator be Public?

For the graph to be accessible and visible to all, storing it in a blockchain is the most fitting choice. The best way to store data in the blockchain is by creating a smart contract and storing the data on it. As defined in Section 1.1.1, a unique address identifies a smart contract, and once it is published, its location is known to all the blockchain's participants thanks to its address. Therefore, a smart contract can be created to contain the swap graph and then published on the blockchain. The participants can use the smart contract's address containing the graph to access it. As a result, the coordinator will be represented by the smart contract that contains the swap graph noted SC_c. The smart contract SC_c implements the state machine logic introduced in Figure 6.4. This contract is

used to coordinate the protocol preventing the occurrence of both *redeem* and *refund* decisions. The participants choose the blockchain that will host SC_c before the beginning of the swap. The algorithm and the criteria for choosing the blockchain where SC_c is published are out of scope. However, specific requirements must be met to claim to be the "coordinating" blockchain (the one containing the coordinator contract). These characteristics will be discussed in the second section of this chapter.
We assume that correct participants can evaluate the coordinator's correctness in the $\mathscr{P}_{swap}$ protocol. Delegating the coordinator's responsibility to a contract makes it possible to satisfy this assumption. Once published on a blockchain, everyone should analyse SC_c smart contract. All participants can obtain information from SC_c using its address and evaluate its correctness. However, some blockchains do not yet offer the possibility of writing smart contracts, which would pose a problem for instantiating the swap protocol. We analyse blockchains' compatibility with the protocol $\mathscr{P}_{inst}$ in the next Section 8.2.

8.1.3 How Can Assets be Locked During the Swap?

Smart contracts also ensure the locking of assets. Each edge of the swap (i.e. each transfer) will result in a smart contract creation with information such as the asset's identifier and the recipient's identifier (its public key). The smart contract functions are implemented in such a way as to establish rules based on these two pieces of information. According to $\mathscr{P}_{swap}$, each source has to lock each asset it wants to transfer. In $\mathscr{P}_{inst}$, each asset a_i is locked in a unique smart contract SC_{a_i}, with $i \in \{1,...,m\}$ and m the total number of assets. As a result, a source publishes a smart contract SC_{a_i} on its corresponding blockchain for each asset it wants to transfer within the swap. The publication of SC_{a_i} sets the new asset's owner r_i. By publishing a contract, sources express their agreement to transfer assets and prevent the asset from double-spending. This operation of publishing asset a_i corresponds to the operation $LockAsset(a_i,\ r_i)$ defined in Section 6.3.3. By publishing the contract, the asset is locked and can only be unlocked if pre-defined conditions in the contract are met. The only ones who can unlock the asset are the recipient of the asset defined in the contract and the contract creator, i.e. the source. The smart contract identifies the source and the recipient thanks to their public key.

Each participant's action in $\mathscr{P}_{swap}$ has an equivalent function in the smart contracts. For instance, a source has actions $LockingAsset(a_i,\ r_i)$ and $RecoveringAsset(a_i, Proof_{refund})$ in $\mathscr{P}_{swap}$, where the second action allows for the recovery of the asset in case of a swap *refund*. Both actions will be coded as a function in the SC_{a_i} smart contract. Similarly, the recipient can retrieve the asset that the source has transferred to it using the action $RetrievingAsset(a_i, Proof_{redeem})$, which will be coded in the SC_{a_i} contract as well. As a reminder, when a source wants to transfer a physical asset to a recipient, the source must tokenise the asset (see Section 2.2.3). Since the transfer is done within the blockchain, the asset needs to have a token representation to perform actions on it. A template of SC_c and SC_{a_i} smart contracts are depicted in Algorithms 1 and 2.

8.1.4 How to Instantiate Trustworthy *proof-of-actions*?

As described in Section 6.3.2, the *proof-of-action* is a mechanism to guard against Byzantine participants. This mechanism of verifiable proof can be instantiated in various ways. $\mathscr{P}_{inst}$ relies on smart contracts to ensure locking assets and coordinate the swap. Participants invoke a series of function calls of involved contracts to execute the swap throughout the protocol. Each function call or contract modification generates a transaction stored in a block of its corresponding blockchain. Blocks that contain the transactions can neither be removed nor modified and therefore represent a guarantee of confidence that a transaction has taken place. Therefore, a block that records transactions could constitute a reliable *proof-of-action* to certify the execution of a particular action or operation, as in [102]. However, as discussed in Sections 2.2.4 and 2.2.5, there are different types of blockchain, each with various characteristics. For example, the blockchain consensus mechanism

may be different, and depending on the consensus, the blockchain behaves differently. Bitcoin [144] and Ethereum [50], are based on *Proof-of-Work* (*PoW*) [144] and *Proof-of-Stake* (*PoS*) [178] consensus. Their mechanism for adding blocks to the chain can generate *forks* (see Section 2.2.5). The result is that some participants might not have the same chain view locally. The rule of the longest chain allows reconciling the blockchain state. However, after reconciliation, the blocks of alternatives chains parallel to the longest chain are revoked, i.e. the transactions inside those blocks are cancelled. In that case, we say that confirmation is probabilistic. As a result, what we have introduced about blocks used for *proof-of-action* is likely to be complicated.
Conversely, a *committee-based blockchain* [1] is a category of blockchain that relies on the Byzantine Fault Tolerant (*BFT*) consensus mechanism [125]. As a recall, the block creators are known and clearly defined as the *validators*. Each produced block is signed by a subset of validators called a *committee*. Using deterministic *BFT* consensus offers consistency guarantees that forks will never occur as long as no more than $\frac{1}{3}$ of the committee members are Byzantine participants; hence the blockchain will always have a unique chain. These blockchains guarantee immediate block finality, 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 maximum number of participants that can deviate from the protocol, i.e. that can be Byzantine. A block signed by a quorum is called a certified block. Examples of such blockchains are *Zilliqa* [167] and *Tendermint* [115].
As a result, we rely on committee-based blockchains in the instantiated protocol presented in this section, and all the blockchains involved in the swap are of this category. Thus, an added block to the chain is an immediately confirmed block and can no longer be undone. Consequently, the instantiated protocol uses proofs based on certified blocks to implement *proof-of-action* that conditioned the protocol.

8.1.5 Description of the $\mathscr{P}_{inst}$ Phases

In addition to the swap graph, the list of validator addresses of the blockchains participating in the swap is given as input to the protocol. The participants define a coordinating blockchain where SC_c is published. In the following, we describe the phases of the protocol $\mathscr{P}_{inst}$ referring to Figures 8.1 and 8.2 and Algorithms 1 and 2. Figure 8.1 illustrates the protocol execution flow in the case of a *redeem* scenario, while Figure 8.2 illustrates the case of the *refund* scenario. Note that the case of the *refund* can be achieved by other possible scenarios, unlike the *redeem*, which can only be reached by the one shown in Figure 8.1.
Figures 8.1 and 8.2 are sequence diagrams that give high-level interactions between participants of $\mathscr{P}_{inst}$ and the involved blockchains. The figures visually show the interaction's order by using the vertical axis of the diagram to symbolise time, what messages are sent and when.

The *redeem* scenario. We start by describing Figure 8.1, which represents the scenario where the swap occurs. We distinguish three phases (identified by the three different colours). The assumption for achieving the *redeem* scenario is that the publisher and the sources must be correct.

Phase 1: proof of SC_c publication. First of all, the $\mathscr{P}_{swap}$ protocol starts with the publisher's action. As mentioned earlier, the swap graph is stored in the SC_c smart contract. Therefore, the publisher's role is to publish the smart contract SC_c on the coordinating blockchain containing the swap graph $swap$ and the list of validators addresses $validators$. The function that provides this action is PUBLISH($Swap\_swap, list\_validators$), defined in Algorithm 1 line 9. Once the function is executed, it initialises the SC_c global variables $swap$ and $validators$ with the value given as inputs ($swap$ and $validators$). Moreover, the contract state (represented by the $state$ variable in line 6) is set to $Published$. Line 5 of the SC_c represents the set of states in which the variable $state$ can be

[1] Note that committee-based does not refer to the types of blockchain we introduced in Section 2.2.6. It refers to a category of blockchain that relies on a committee (or set of validators) to validate blocks, not a type of blockchain.

(i.e. the state of the contract). The execution of the PUBLISH function from Algorithm 1 creates a transaction inside the coordinating blockchain. After being signed and validated by the validators of the blockchain, the transaction is permanently added to a block called B_{SC_c}. That block represents the proof of SC_c publication that can be noted as $proof\_of\_publish$ (referring to $Proof_{publish}$ in Section 6.4). At this point, the role of the publisher is over. However, it is not excluded that the publisher is also a source or recipient of the swap. In this case, it has to execute the protocol corresponding to the source or recipient.

At the beginning of the protocol, the source assets are in their associated wallet, not yet locked. The $proof\_of\_publish$ is needed for each source to publish its contract and lock its asset. The sources must wait for SC_c publication before publishing SC_{a_i} to avoid a forever locking asset. Consequently, sources retrieve block B_{SC_c} from the coordinating blockchain and verify its validity. If the block satisfies the validity conditions, it can be considered a valid $proof\_of\_publish$. A valid proof of publication means that the transaction of SC_c publication exists in block B_{SC_c}. Plus, the information contained in it must be consistent with that held locally by the sources and recipients. For example, if the swap graph $swap$ stored in the SC_c contract does not match the swap graph constructed upstream, the sources (and the recipients) may abandon the swap. The same applies if the list of validators $validators$ is not consistent with the actual set of validators. Note that this proof-checking step is done at the protocol level (via CorrectSwap predicate defined in Figure 6.5), not at the smart contract level.

Phase 2: proof of locking assets in SC_{a_i}. The validity of B_{SC_c} enables the lock of each asset a_i in a smart contract SC_{a_i} by its corresponding source and publishing it. In this instantiation example of $\mathscr{P}_{swap}$, the PUBLISH function defined in Algorithm 2 has no precondition to satisfy, unlike its equivalent $LockAsset(a_i, r_i)$ action implemented in TLA$^+$. The way the function is instantiated is perfectly adaptable according to the needs.
The locking asset action, executed by the sources, represents the publication of the contract that contains the asset a_i to be transferred. Once the asset is locked in its contract, it is no longer in the source's wallet. The function PUBLISH($Asset\ \_asset$, $Address\ \_recipient$, $Block\ proof\_of\_publish$) is defined in the Algorithm 2 line 9. When the source publishes SC_{a_i}, it must provide the address of the recipient r_i and the $proof\_of\_publish$; that is B_{SC_c}. Similarly to SC_c, the publication of SC_{a_i} initialises the variables $asset$ (line 3) and $recipient$ (line 2) with the values given as inputs ($\_asset$ and $\_recipients$). Plus, the state of the contract updates by assigning the $state$ variable (line 6) to $Published$. The function extracts the SC_c's information from B_{SC_c}, such as its address, and assigns it to scc (line 12). Once the sources' contracts are published, a transaction is created for each, validated and added to a block in their corresponding blockchain. The block where the transaction representing SC_{a_i} publication is located is named $B_{Sc_{a_i}}$. That block will correspond to the proof of locking asset noted $proof\_of\_lock$, proving the good behaviour of sources. This proof refers to $Proof_{lock}$ in Section 6.4.
According to the source's protocol, each correct source must request a *redeem* decision to SC_c along with a $proof\_of\_lock$. Therefore, the sources retrieve from their blockchain the block $B_{Sc_{a_i}}$, representing the $proof\_of\_lock$, and perform the request represented by the call of the function VALIDTRANSFER($B_{Sc_{a_i}}$) from the contract SC_c (line 13 in Algorithm 1). Notice that if a source transfers multiple assets, it must make for each asset a request. The VALIDTRANSFER function can only be executed if its precondition is satisfied. The precondition is that the contract SC_c must already be published, and the provided $proof\_of\_lock$ must be valid (line 14). The verification of $proof\_of\_lock$ is done at the SC_c level using the helping function VALIDPOA($Block\ proof\_of\_lock$) (line 21).

As the contract SC_c has the list of validators of each blockchain of the swap ($validators$, line 3 in Algorithm 1), the function in line 21 will check if the block $B_{Sc_{a_i}}$, that represents the $proof\_of\_lock$, has at least $2f+1$ signatures of the $validators$. The function extracts information about the sources'

asset and the recipient meant to receive the asset from the $swap$ variable. The verification succeeds if the source has correctly published its contract (giving the correct recipient address and locked the correct asset) and if the block belongs to a real blockchain (it has the necessary validators' signature). The contract defines a kind of a hashtable variable $ProofLock[]$ with the sources' address as the key and the value a boolean that sets $True$ if the $proof\_of\_lock$ provided by the source is valid.

Phase 3: proof of redeem decision. Participants, sources and recipients wait for SC_c to change state from $Published$ to $OkRM$. As mentioned previously, the function responsible for changing the state of SC_c to $OkRM$ is ValidTransfer defined in Algorithm 1. If all sources provide a valid $proof\_of\_lock$ (i.e. all values of $ProofLock$ are set to $True$), the *redeem* decision is given by changing the state of SC_c to $OkRM$. The changing state of SC_c to $OkRM$ creates a transaction. After being validated by the coordinating blockchain validators, the transaction is added to a block. The block where the transaction is located is named B_{OkRM}. Therefore, correct recipients can retrieve the block B_{OkRM} from the coordinating blockchain representing the $proof\_of\_redeem$. They execute the function RetrievingAsset(B_{OkRM}) from SC_{a_i} (line 14 in Algorithm 2) to retrieve their assets from their corresponding contract. The function is conditioned by the fact that the contract must be published and the $proof\_of\_redeem$ must be valid. The contract SC_{a_i} defines a helping function, ValidRedeemPoA($Block\ proof\_of\_redeem$), to check whether the proof is valid. The verification confirms that the smart contract contained in $proof\_of\_redeem$, i.e. B_{OkRM}, is the same as scc. It is sufficient to check that the addresses of the two contracts are the same to confirm the correspondence. Moreover, the helping function verifies if the function caller is the rightful recipient. If the conditions of the function are not met like the function caller does not have the same public key as defined in the contract, the function is not executed, and the asset remains locked in Sc_{a_i}. Thus, if all the conditions are met, the correct recipient can retrieve its expected asset by using its public key to attest its legitimacy to the received asset. Thereby, the contract state changes from $Published$ to $Redeemed$.

The *refund* scenario. The second described scenario is the *refund* case (Figure 8.2), which is divided into four phases (identified by the four colours). The scenario assumes that the publisher is correct and that there exists a subset of Byzantine sources that do not lock their assets correctly. Therefore, the *Phase 1* is identic to the *redeem* scenario, meaning that the swap is correctly published in the coordinating blockchain and the $proof\_of\_publish$ is valid.

Phase 2: proof of locking assets in SC_{a_i}. The second phase of the *refund* scenario is almost identical to the *redeem* scenario with one exception. The assumptions said that a subset of sources are Byzantine and do not perform the asset locking step correctly. Conversely, the correct sources perform the *Phase 2* described in the *redeem* scenario.
Suppose that a $proof\_of\_lock$ provided by a Byzantine source is invalid (e.g. a wrong recipient address). In that case, the precondition of the ValidTransfer function in SC_c will not be satisfied because the helping function ValidPoA will return $False$. This result prevents the update of the $ProofLock[]$ variable at the Byzantine source's index. In doing so, the contract will not be able to change state to $OkRM$, and the only possible end for the protocol is a *refund* decision.

Phase 2': asking for refund decision. As the state of the contract SC_c does not change, due to Byzantine sources' actions, at any time, a participant [2] can request a *refund* decision. The function responsible for changing the state of SC_c to $OkRF$ is AbortTransfer (line 18 in Algorithm 1). The only precondition to satisfy the function is that the contract is still in the $Published$ state. If so, the contract SC_c changes its state to $OkRF$.

[2] The dashed arrows in Figure 8.2 express that the function call of AbortTransfer can be made by a source (correct or Byzantine) or a recipient (correct or Byzantine), or both.

Phase 3: proof of refund decision. If a participant calls the function AbortTransfer, the contract SC_c updates its state to $OkRF$. The changing state creates a transaction and it is added in a block B_{OkRF}. The correct sources retrieve the block B_{OkRF}, referring to the proof for a *refund* decision noted $proof\_of\_refund$, to recover their assets. The recover asset operation is possible due to the function RecoveringAsset($Block\ proof\_of\_refund$) in SC_{a_i} (line 19 in Algorithm 2) that need proof of *refund* decision. The contract SC_{a_i} defines a helping function to verify the validity of $proof\_of\_refund$, i.e. the function ValidRefundPoA($Block\ proof\_of\_refund$) line 28 in Algorithm 2. The verification consists of checking if the public key calling the function matches the public key of the contract creator. Moreover, the function verify if the address of the contract extracted from $proof\_of\_refund$ correspond to scc. If the conditions are not met, the function cannot be executed, and the asset remains locked in SC_{a_i}. Conversely, once the conditions are satisfied, the RecoveringAsset function in line 19 unlocks the asset, and correct sources can retrieve the assets they have locked using their public key, changing the contract state to $Refunded$.

Smart Contract 1 SC_c contract

1: **variables**
2: $swap : Swap$ ▷ The swap graph of $\mathscr{P}_{inst}$ that the participants must construct
3: $validators : list\ address$ ▷ The list of block validators of the involved blockchains
4: $ProofLock[address] : boolean$ ▷ The size of the hashtable is the number of sources
5: $Enum\ State = \{Published, OkRM, OkRF\}$ ▷ Possible states of the contract
6: $state : State$ ▷ The variable that characterises the state of the contract
7: **variables**
8:
9: **function** PUBLISH($Swap\ \_swap$, $list\ \_validators$) ▷ The function executed by the publisher in *Phase 1*
10: Assign the variables $swap$ and $validators$ to $\_swap$ and $\_validators$ parameters;
11: Assign the keys of $ProofLock[]$ to the sources' addresses and initialise the values to $False$;
12: Upate the state of the contract $state$ to "$Published$";
13:
14: **function** VALIDTRANSFER($Block\ proof\_of\_lock$) ▷ The function executed by the sources in *Phase 2*
15: **if** $state = Published$ and VALIDPOA($proof\_of\_lock$) = $True$ **then** ▷ VALIDPOA is the helping function defined in line 22
16: $ProofLock[caller] = True$; ▷ $caller$ is the VALIDTRANSFER function caller
17: **if** $\forall i \in$ {the set of sources from $swap$}, $ProofLock[i] = True$ **then** ▷ the set of sources is extracted from the $swap$ variable by VALIDPOA
18: $state \leftarrow OkRM$;
19:
20: **function** ABORTTRANSFER ▷ The function can be executed by a source or a recipient in *Phase 2*'
21: **if** $state = Published$ **then** $state \leftarrow OkRF$;
22:
23: **function** VALIDPOA($Block\ proof\_of\_lock$) ▷ The function that verifies the $proof\_of\_lock$
24: Extract from $swap$ the set of sources;
25: Check if $proof\_of\_lock$ is valid according to the swap and the set of sources;

Smart Contract 2 SC_{a_i} contract

```
1:  variables
2:      recipient : Address                                   ▷ The new owner of the asset asset
3:      asset : Asset                                         ▷ The locked asset
4:      scc : SmartContract                                   ▷ The variable that instantiates the contract SC_c
5:      Enum State = {Published, Redeemed, Refunded}          ▷ The possible states of the contract
6:      state : State                                         ▷ The variable that characterises the contract state
7:  variables
8:
9:  function Publish(Asset _asset, Address _recipient, Block proof_of_publish)   ▷ The function is executed by the sources in Phase 2
10:     Assign the variables asset and recipient to _asset and _recipients parameter;
11:     Upate the state of the contract state to "Published";
12:     Extract from proof_of_publish the contract SC_c and assign it to scc variable;
13:
14: function RetrievingAsset(Block proof_of_redeem)   ▷ The function is executed by the recipients in Phase 3 of the redeem scenario
15:     if state = Published and ValidRedeemPoA(proof_of_redeem) = True then   ▷ ValidRedeemPoA is a helping function defined in line 24
16:         Transfer the asset's ownership to the recipient recipient;
17:         Upate the state of the contract state to "Redeemed";
18:
19: function RecoveringAsset(Block proof_of_refund)   ▷ The function is executed by the sources in Phase 3 of the refund scenario
20:     if state = Published and ValidRefundPoA(proof_of_refund) then   ▷ ValidRefundPoA is a helping function defined in line 28
21:         Transfer the asset's ownership to the contract creator;   ▷ The creator is the source of the asset
22:         Upate the state of the contract state to "Refunded";
23:
24: function ValidRedeemPoA( Block proof_of_redeem)   ▷ The function that verifies the proof_of_redeem
25:     Check if the proof_of_redeem is valid according to the variable scc;
26:     Verify if the function caller is the expected recipient, i.e. recipient;
27:
28: function ValidRefundPoA(Block proof_of_refund)   ▷ The function that verifies the proof_of_refund
29:     Check if the proof_of_refund is valid according to the variable scc;
30:     Verify if the function caller is the expected source, i.e. the contract creator;
```

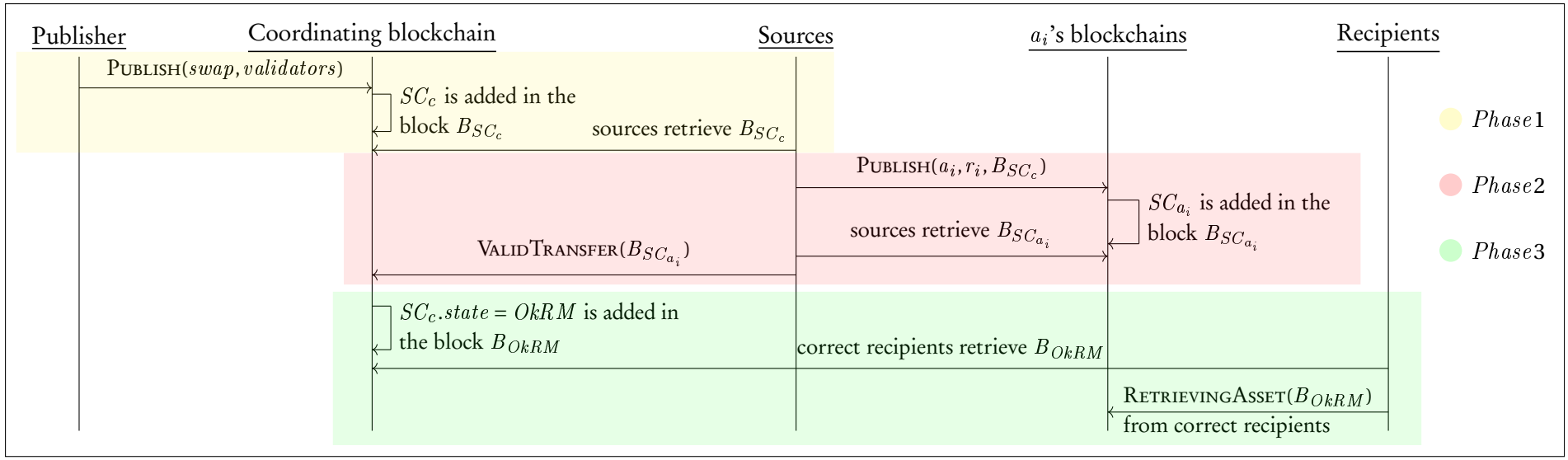


Figure 8.1 – The execution flow of a *redeem* scenario

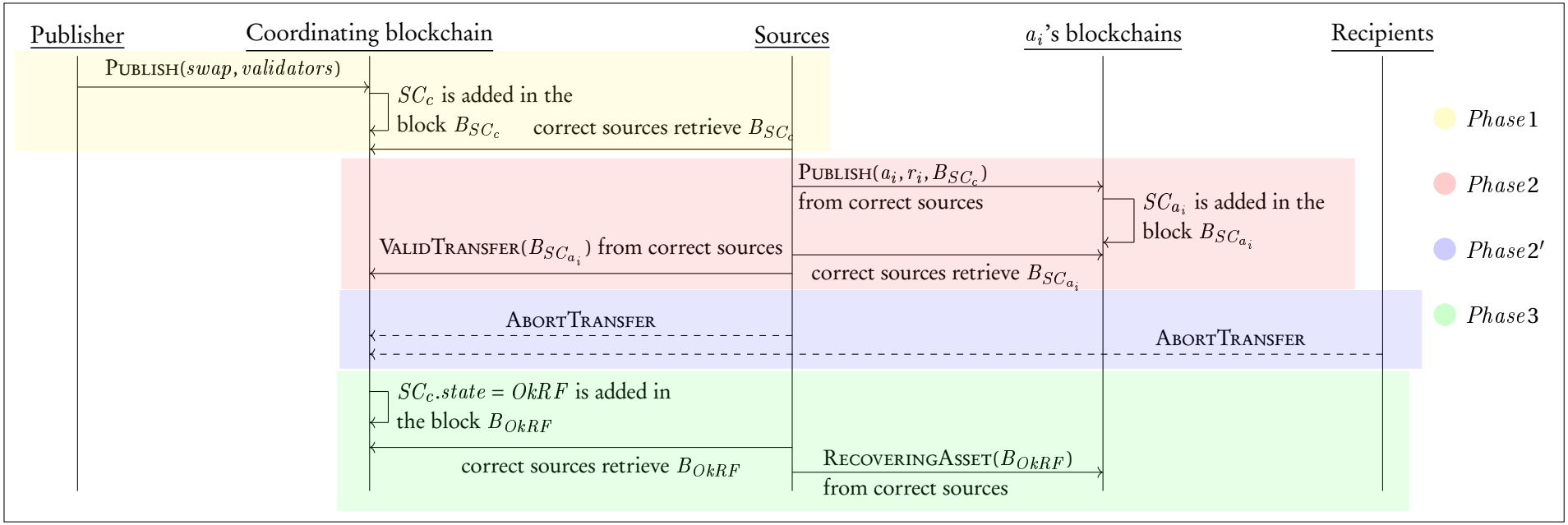


Figure 8.2 – The execution flow of a *refund* scenario

8.2 Protocol Compatibility with Different Known Blockchains

In the previous section, we introduced the requirements of the blockchain involved in the swap, which is, among others, the possibility to write smart contracts and to have immediate finality of blocks possible with non-forking blockchains. To contextualise these requirements, we analyse the adaptability of the protocol to a set of blockchains. This section considers some blockchains already defined in Section 2.2.6, namely public/private and permissionless/permissioned. The previous section explored how the protocol could exploit the specificities of blockchain technology to instantiate the swap protocol $\mathscr{P}_{swap}$. That said, not all blockchains provide the same specificities. Therefore, it is necessary to identify the essential characteristics to be compatible with the protocol. Four requirements must be satisfied for a blockchain to take part in $\mathscr{P}_{inst}$:

1. The block data must be public (at least the coordinating blockchain).
2. The possibility of writing smart contracts with a high level of expressiveness.
3. A consensus mechanism based on a committee and validators producing certified blocks.
4. A blockchain with immediate finality without the appearance of forks.

We study 13 blockchains, some of them are introduced in Section 2.2.6: Bitcoin [144], Ethereum[50], Ripple [22], Hyperledger Fabric [21], Monet [23], MultiChain [93], Quorum [161], EOS [188], Cosmos Hub [116], Tendermint [9, 48], Cardano [51], Tezos [14], and Zilliqa [167].
Table 8.1 summarises the analysis made below and provides the necessary information to set the compatibility of the analysed blockchains with the protocol $\mathscr{P}_{inst}$. The left-hand column of the table lists the requirements that a blockchain must meet to participate in the $\mathscr{P}_{inst}$ swap. The blockchain satisfies the requirement if a checkmark (✓) is drawn. Conversely, a cross mark (✗) means that the blockchain does not fulfil the requirement.

8.2.1 Public Data

First of all, one of the four requirements for a blockchain to instantiate the protocol is to have complete data read access. Remember, the role of the coordinator is endorsed by a smart contract in the blockchain environment. In order to track the state of the smart contract, the participants must have constant access to that contract; hence, the contract SC_c must be readable by all participants. Suppose the state of the contract SC_c changes to authorise the assets to be redeemed, i.e. becomes "$OkRM$". In that case, the recipients must read the information on the coordinating blockchain that hosts the contract SC_c. This feature of public readable data is found in public blockchains such as Bitcoin and Ethereum, representing the first generation of blockchain. Other blockchains allow open data reading, such as Ripple [22], Monet [23], Tezos [14], MultiChain [93], Cardano [51], and Zilliqa [167]. Moreover, each user of the cited blockchains can participate in the consensus mechanism by creating and validating transactions. EOS, part of the public permissioned blockchain type, allow authorised users to access the blockchain data; however, EOS can create custom permission on specific features of a smart contract imperceptible to users.

Not all blockchains have their transactions and blocks data open to users by design. The Hyperledger Fabric [21] and Quorum [161] blockchains are private and permissioned with customisation and modularity services in their privacy rules (the consensus mechanism depends on the organisation). Both offer developers the ability to make their transaction data private or public. For example, Hyperledger has no unique blockchain network. Instead, businesses, consortia, and other organisations deploy Hyperledger technologies to build networks that support their needs. Thus, whoever deploys a blockchain based on Hyperledger technology will have the choice to make the data of the blockchain public or not. Therefore, if either blockchain becomes involved in $\mathscr{P}_{inst}$, the transaction data must be configured to be public; otherwise, the blockchain will not apply to the protocol. At least, the transactions concerning the protocol must remain public, i.e. the SC_c

contract and the transactions that reflect the change of the smart contract state. From a protocol point of view, all other data that does not concern the swap can remain private.

Similarly, the Cosmos Hub [116] and Tendermint [9] are frameworks for building public and private blockchain applications. Tendermint offer, like an engine, the network and consensus layers so that the developer only has to focus on the application layer. Tendermint is a blockchain protocol used to replicate and launch blockchain applications across machines securely and consistently. Therefore, if a blockchain based on the Tendermint foundation wishes to participate in the swap, it must to publicise the block data.
The Cosmos Hub is a blockchain based on Tendermint. It presents functionalities allowing independent blockchains to rely on pre-elaborated methods of consensus and governance and communicate easily by sending tokens or messages. Using the Tendermint protocols, Cosmos presents itself as the internet of blockchains. It relies on a set of validators responsible for committing new blocks in the blockchain. These validators participate in the consensus protocol by broadcasting votes that contain cryptographic signatures signed by each validator's private key. Access to the blockchain data is decided at the time of deployment of the blockchain by the developer. Thus, like Tendermint, a Cosmos blockchain can be protocol compliant if access to transaction data is open.

8.2.2 Smart Contracts

Blockchains must handle the implementation of smart contracts to execute the protocol or equivalent programs with the same level of expressiveness. Programs establishing transaction rules appeared with Nick Szabo [172] and later with Ethereum smart contracts. As a result, most Ethereum-derived blockchains maintain the specificity of writing smart contracts, such as the Quorum blockchain – a private permissioned Ethereum blockchain. It has all the features of Ethereum, including the ability to write smart contracts. Quorum is adapted to the protocol where the network allows publicly publishing contracts. Ethereum smart contracts can share or transact unique physical or digital assets and tokens across many of the world's leading blockchain platforms and networks that use the Ethereum virtual machine (EVM).

Solidity [78] is the Ethereum smart contract's programming language and allows complex user-defined types. *Solidity* supports mapping data structures, which act as hash tables and consist of key types and key-value pairs. That makes it perfect for the instantiation of the protocol from a smart contract language point of view. This feature has allowed blockchain to emancipate and reach a broader range of users. As a result, many new blockchains have adopted this feature as ideal for developing decentralised applications like the Monet Hub platform [23], where anyone can publish smart contracts.
The Hyperledger Foundation supports different programming languages to write smart contracts, such as Javascript, C++, and *Solidity*. Hyperledger Fabric (one of the six graduated Hyperledger projects) allows the building of smart contracts called *chaincode*. A chaincode is programmatic code published on the network, where it is executed and validated by chain validators together during the consensus process. Similarly, in Tendermint and Cosmos Hub, the application layer can be developed in any programming language. The same goes for EOS, which implements smart contracts with a high expressiveness written in C++. A virtual machine executes smart contracts, and the generated files are the smart contracts that can be published on EOS blockchains.
Zilliqa and Tezos blockchain implement so-called formal smart contracts. The smart contract language in Zilliqa called Scilla [168] follows a dataflow programming paradigm. Scilla is motivated by functional programming languages such as OCaml, making it suitable for formal verification. Likewise, Tezos blockchain offers a platform to create smart contracts in the *Michelson* language [173].
The Cardano blockchain is a developing platform and supports the development and publication of smart contracts using various programming languages, including formal language like Plutus [52].

The possibility or not of writing smart contracts will depend on the blockchain. Some blockchains implement the concept of smart contracts differently but with the same goal of establishing rules for transactions execution. For example, the very first blockchain, Bitcoin, has its equivalent scripts that establish rules for execution. These scripts can only be applied to a limited number of scenarios. For instance, the Bitcoin smart contract excludes loops to avoid potential ongoing operations resulting in the workflow bottleneck. As for Ripple, it does not allow writing smart contracts as defined above. One can use *Ripple Ledger escrows*[3] as smart contracts that release the escrowed asset after a particular time or after a fulfilled cryptographic condition. Those limits of Bitcoin and Ripple smart contracts make them unsuitable for the protocol.
The last example of a smart contract equivalent is the *smart filters* of the MultiChain blockchain [143]. A Smart Filter is a Turing-complete piece of code embedded in the blockchain and allows custom rules to be defined regarding the validity of transactions. They are written in JavaScript and handle the definition of functions. Although they are different from smart contracts, smart filters seem adapted to the $\mathscr{P}_{inst}$.

8.2.3 Certified Blocks and Absence of Forks

Another requirement of the protocol is the implementation of the *proof-of-action*. In order to implement the concept of *proof-of-action*, what is crucial from blockchain is to ensure the reliability of the proof provided on the one hand to the smart contract coordinator and the other hand to the smart contract of the sources. Therefore, it is essential to have blockchains based on a consensus mechanism that does not generate forks and ensures blocks with immediate finality. These requirements exclude all blockchains based on *PoW*-type and *PoS*-type consensus, including the Bitcoin and Ethereum blockchains. This type of blockchain uses probabilistic consensus mechanisms, and we have shown that it is impossible to implement *proof-of-action* assuming this type of consensus. Indeed, probabilistic consensus can generate forks in case of simultaneous block validation, which leads to the possibility of a different reading of the chain by two users. In case of a probabilistic consensus, for the *proof-of-action* validation based on blocks, it would be necessary to provide the function verifying the *proof-of-action*, the whole blockchain chain, i.e. from block genesis to the current block. This scenario seems impossible to achieve, so this type of blockchain is not suitable for the instantiated protocol because of the consensus mechanism that cannot provide certified blocks.

In addition, built on a probabilistic consensus, Ripple and Cardano (based on *PoS*) can not ensure certified blocks. These blockchains apply the longest chain rule to solve the fork problem. As a result, blocks are not immediately finite, which compromises the reliability of the *proof-of-action*. Tezos also implements *PoS* consensus called Emmy+ [118]. The protocol is a *PoW*-style consensus; hence, it offers only probabilistic finality.
MultiChain can generate blockchains that might fork. MultiChain has a very high level of customisation of the blockchain. Each user who wants to create a blockchain can configure it as it wishes. It has a list of parameters, and depending on this configuration, the blockchain can be forkable or not. For example, by varying parameters such as `target-block-time`[4], the average time between each block, the forks are minimised or maximised. Moreover, a lower value of the `mining-turnover`[4] parameter reduces the number of forks, making the blockchain more efficient, but increasing the level of validator concentration. Depending on its implementation, a MultiChain blockchain can be a Bitcoin-based blockchain or a committee-based blockchain.

Blockchains that rely on a committee to perform the consensus mechanism more efficiently solve the fork problem, as with Hyperledger Fabric, Quorum and Monet. It ensures that the blocks generated are immediately final and that each block added to the blockchain is certified.

[3] Ripple Ledger escrows: https://xrpl.org/use-an-escrow-as-a-smart-contract.html
[4] List of API commands: https://www.multichain.com/developers/json-rpc-api/

The EOS blockchain consensus mechanism is divided into two levels. The first level is the "producer voting/scheduling", which uses the consensus mechanism of *Delegated Proof-of-Stake* (*DPoS*) to elect the active producers authorised to sign valid blocks. The second level is the "block production/validation", which performs an *asynchronous* Byzantine Fault Tolerant (aBFT) consensus [140] to confirm each produced block until it becomes final (irreversible). The EOS consensus model achieves algorithmic finality through signatures from the chosen set of participants (active producers) arranged in a schedule to determine which party is authorised to sign the block at a particular time slot. When a valid block meets the consensus requirements, the block becomes final and is considered irreversible. EOS consensus mechanism does not need to wait for all the nodes to finish a transaction. Consequently, EOS achieves high transaction throughputs to achieve finality. This behaviour results in faster confirmations and lower latency. We can assume that a fork will have a low probability of appearing.
Tendermint, Cosmos Hub and Zilliqa do not employ *PoW*-based protocol to achieve consensus. These blockchains make use of the BFT protocol [125] to create and add a block to the chain. As a result, once a transaction is included in a block, it cannot be cancelled. More precisely, Zilliqa leverages *PoW* to establish identities but employs the Practical Byzantine Fault Tolerance protocol (*PBFT*) [54] for consensus.

$\mathscr{P}_{inst}$ requirements	Ripple	Bitcoin	Ethereum	Hyperledger Fabric	EOS	MultiChain	Monet	Quorum	Cosmos Hub	Tendermint	Cardano	Tezos	Zilliqa
Public data	✓	✓	✓	*	✓	✓	✓	*	*	*	✓	✓	✓
Smart contracts	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No forks occurence	✗	✗	✗	✓	⋆	⋆	✓	✓	✓	✓	✗	✗	✓
Certified Blocks	✗	✗	✗	✓	✓	•	✓	✓	✓	✓	✗	✗	✓

*: The public aspect of these blockchains depends on their configuration.

⋆: These blockchains can be configured to have a very low probability of fork.

•: The MultiChain can be configured to provide certified blocks.

Table 8.1 – Compatibility of $\mathscr{P}_{inst}$ requirements of some known blockchains

Through these examples of blockchains, we show how to identify the applicability of a blockchain to the $\mathscr{P}_{inst}$ protocol. The private blockchains (permissionless or permissioned) are favoured because they guarantee deterministic consensus. A set of validators performs the validation of blocks. Table 8.1 gives a comparison between the analysed blockchain regarding the protocol requirements. We can conclude that the adapted blockchains are Monet [23] and Zilliqa [167] by design. However, several blockchains are customisable and can be suitable for implementing the $\mathscr{P}_{inst}$ protocol. The blockchain group that brings together Hyperledger Fabric [21], Quorum [161], Cosmos Hub [116], and Tendermint [9] has the same characteristic of customising openness and access to blockchain data. Therefore, a participant on one of these blockchains can participate in the swap if its underlying blockchain has been configured to have open access to transactions.
The adaptability of EOS [188] and MultiChain [93] to the protocol needs more than an analysis of the documentation. Both ensure a very low probability of forks (or even none) if the blockchain's configuration parameters are correctly tuned. In order to conclusively assess the applicability of these blockchains, it would be necessary to implement a running example of the protocol that involves EOS and MultiChain blockchains.

8.3 Conclusion

This chapter shows how the protocol defined in Chapter 6 is suitable in a blockchain environment. We have shown that some blockchain-specificities, such as the ability to write smart contracts, fit perfectly with the protocol $\mathscr{P}_{inst}$. However, we have seen in Chapter 2 that there are several types of blockchains with different characteristics, and we cannot ensure the compatibility of the protocol with all types of blockchains. Table 8.1 shows the diversity of existing blockchains, making it impossible to implement the protocol for some of them. It can be seen that blockchains based on the *BFT* consensus are better able to fulfil all the protocol requirements. It should be noted that the analysis in this chapter is not intended to provide a precise implementation method. Instead, it provides an example of instantiation that can be different from one instantiation to another.

Part V

Conclusion

Chapter 9

Conclusion

“ Those who can imagine anything, can create the impossible. ”

– Alan Turing

Contents

This section gives general conclusion about the work that has been done in this thesis. In addition, we provide some perspectives and future work.

9.1 General Conclusion of the Thesis

A distributed system is a computing environment in which different components are spread across multiple computers on a network. They are complex to configure and difficult to manage. Interesting features characterise distributed systems: scalability, fault tolerance, concurrency, replication, and transparency, making these systems appealing to use. However, a program or a system shared and used by many processes or participants can quickly become a source of issues. Blockchains are examples of complex distributed systems. The study of a blockchain system is multi-level and brings together a wide range of skills like cryptography, algorithmics and economics. This thesis studies two aspects of blockchain systems, *smart contracts* and *cross-chain swap* applications. The study consists of the system's design and its formal verification. In the case of smart contracts, an additional step is their compilation to a virtual machine.
The first question introduced at the beginning of the manuscript is " *how to ensure that a contract is correct and respects its specification?*". One of the answers to this question is to manage to write precise contracts using languages with formal semantics. In Chapter 5, it was shown that the *Solidity* language does not fulfil this requirement making their smart contracts prone to bugs and flaws. In response to this observation, we used a formal language with well-defined semantics to write smart contracts. The *WhyML* language is well adapted for this kind of use since it allows for writing both logical and imperative code; Chapter 4 provides sufficient information on the language to support this. In addition to providing a precise and concise description of the proof methodology, a case study with contracts written entirely in *WhyML* is provided. This approach made it possible to ensure the correctness of *WhyML* smart contracts and to prove that these contracts respect their specification.

The second question of the thesis is that "*assuming the smart contract correct, how to ensure the transfer of assets assuming the implication of Byzantine participants?*". Indeed, a smart contract is considered an account; hence, it has a balance and can send transactions over the network. As a result, transferring digitised assets can be empowered by smart contracts. The answer to this question is the same as the first question, which is the application of formal modelling and verification tools. In this thesis, we take the example of an application designed for transferring assets across different blockchains based on smart contracts, namely the *cross-chain swap* algorithms. First of all, we had to design an algorithm that allows the transfer of assets in the presence of Byzantine participants. We defined the algorithm $\mathscr{P}_{swap}$ that abstracts blockchain implementations so as not to limit the algorithm to a blockchain environment.
Nevertheless, before that, we defined a specification and then designed the algorithm, both formally, to avoid any ambiguity. In a second step, we expressed the specification and modelled the algorithm into a formal language, TLA$^+$. Chapter 6 gives the modelling details, including how we have represented the Byzantine participants in the system. This study step partly answers our question, as it provides a precise and formal system model. Moreover, it is necessary to be able to apply verification tools and prove that our model satisfies the problem specification. We have proven in Chapter 7 that the model satisfies the safety property of the *cross-chain swap* problem by applying a well-defined proof methodology described in Chapter 4. The methodology applies concepts of deductive verification, while liveness properties have been verified using model-checking. Moreover, we give in Chapter 8 the instantiation of $\mathscr{P}_{swap}$ into a blockchain environment. In this chapter, we define how the abstractions of the $\mathscr{P}_{swap}$ protocol can be represented in a blockchain instantiation, and we obtain the $\mathscr{P}_{inst}$ protocol. The approach defined therein shows that the protocol is well suited to blockchains. So far, Chapters 6, 7 and 8 provide a proper answer to the question, "*assuming the smart contract correct, how to ensure the transfer of assets assuming the implication of Byzantine participants?*". However, we wanted to go a step further in our analysis of *cross-chain swaps*. We

provide in Chapter 8 an analysis of a set of well-known blockchains and their eligibility, based on specific characteristics, to implement the defined $\mathscr{P}_{inst}$ protocol.

9.2 Future Work

This section presents open research questions from the work presented in this thesis.

9.2.1 Improvement of *WhyML* Smart Contracts

Simplifying the proofs. This thesis proposes a formal language, *WhyML*, as a writing language for smart contracts. We express complex contract properties requiring a non-negligible amount of proof. The writing of the logic part (preconditions, postconditions, invariants) is the user's responsibility (the developer of the contracts). However, this step is often a difficult task because of the invariants definition. Therefore, one approach to improve contracts in *WhyML* would be to simplify the proofs, as we have seen that proofs require some expertise and can sometimes be complicated to write. Alleviating the burden of proof would be possible by enriching the *Why3* library to automate as much proof as possible.

Compilation to other virtual machines. Once the contracts have been written and proven, the last step is to compile the contracts into a virtual machine. In this thesis, contracts are compiled to the Ethereum Virtual Machine (EVM). This step allows calculating the amount of gas consumed by each executed function. We focused on the Ethereum VM as a first approach because it is currently the most important blockchain in terms of use cases. However, one can consider compiling *WhyML* contracts towards other virtual machines. For instance, NeoVM [149] or Algorand VM [13]. The approach would be similar to what was done with the EVM. It will be necessary to create a library for each virtual machine. These libraries will contain the contract parser, which takes the contracts as input and translates them into opcodes.

9.2.2 Improvement of the $\mathscr{P}_{swap}$ Algorithm

Detailed the $\mathscr{P}_{swap}$ assumptions. In the algorithm described in Chapter 6, we made some modelling assumptions. For example, we assumed that the participants chose the coordinator and the publisher before the swap. However, it would be interesting to more detail this working hypothesis and define the algorithm that allows for making those choices. The same comment applies to the swap graph construction, which is assumed to be built upstream of the swap. In the study, the swap graph is an input. However, we could imagine a smart contract that allows the construction of the swap according to the wishes and desires of the users, for instance, like the *Trading* contract of the BEMP application defined in Chapter 5, which allows matching an energy consumer with a producer. In this way, the construction of the swap would be decentralised via a smart contract, thus ensuring its well-construction.

9.2.3 Going Further into the Proof of the $\mathscr{P}_{swap}$ Algorithm

Proof of liveness using deductive verification approach. In this thesis, we have applied two methods of verifying the $\mathscr{P}_{swap}$ algorithm. A deductive approach, using the $TLAPS$ tool to prove the safety, and a model-checking approach with TLC to prove the liveness properties. Although the deductive approach requires user interaction, it was able to prove the *Consistency* of a parametric model in a relatively short time. In contrast, the results obtained for model-checking were limited by the combinatorial explosion of the number of generated states, making verification of the *Ownership* (and partially *Retrieving*) property impossible with a large number of Byzantines participants.
The intuition to correct this problem is to apply deductive verification methods to liveness. However, verifying liveness properties with an unbounded number of processes is difficult. A solution to

this could be to reduce the liveness properties to a safety property. For example, the methodology in [81] is based on the generation of an inductive invariant and a "liveness monitor" that observes the system's behaviour. The methodology consists of a parametrised system S and a liveness property of the form: $\phi : q \implies \Diamond r$, with q and s states of the system. The property ϕ is bounded if there is a chosen bound, K, independent of the number of processes, such that once q is reached after at most K rounds in which each process takes at least one step, a goal r is reached.
The liveness monitor, M_ϕ, increases a round counter when each process takes a step and resets it once q is reached. The round counter never exceeds K if K bounds ϕ. Therefore, proving the liveness property ϕ will be equivalent to proving the formula: $S|||M_\phi \models \Box(rnd < K)$, with rnd the round counter.

Proof of the protocol $\mathscr{P}_{inst}$. The model proven in Chapter 7 is the abstract *cross-chain swap* model, the $\mathscr{P}_{swap}$ algorithm. The modelling in TLA$^+$ is also abstract and models, for example, *proof-of-actions* as a boolean, which is true or false depending on its validity. One can imagine the proof of the instantiated model in a blockchain environment $\mathscr{P}_{inst}$ in future work. According to that, we would implement and model the specificities of blockchains, such as blocks, transactions, and the set of validators that compose the blockchains committee. Thus, the *proof-of-action* (i.e. certified blocks) could be modelled in such a way as to consider the number of signatures. With the resulting $\mathscr{P}_{inst}$ model, we could verify whether the properties of the swap specification are still satisfied with this new model.

9.2.4 Analysis of the Implementation Feasibility of $\mathscr{P}_{inst}$

A running implementation of the $\mathscr{P}_{inst}$ protocol. In this thesis, a theoretical approach to a *cross-chain swap* protocol was undertaken. Although the protocol has been modelled and proven using technical tools, our approach does not involve a practical implementation of $\mathscr{P}_{swap}$. Chapter 8 is a first step in the practical implementation of the protocol, in the sense that we analyse the blockchains that can implement $\mathscr{P}_{inst}$. The continuation of these results would be developing a practical application, with smart contracts written in *WhyML*, that enable the transfer of tokens between the blockchains, cited in Chapter 8, that are considered suitable to implement $\mathscr{P}_{inst}$.

Appendix A

Appendix

A.1 *Two-Phase Commit* TLA$^+$ Code

```
MODULE TwoPhaseCommit
EXTENDS Integers, TLAPS
CONSTANT N
Participants ≜ 1..N
CoordinatorID ≜ 0
CStates ≜ {"init", "pre-commit", "commit", "abort"}
PStates ≜ {"working", "committed", "aborted", "prepared"}

--algorithm TwoPhaseCommit {
  variables cState = "init", pState = [p ∈ Participants ↦ "working"], abortFlag = FALSE;

  define {
  allPCommit ≜ ∀ p ∈ Participants : pState[p] = "prepared"
  atLeastOneAbort ≜ abortFlag = TRUE
   }

  fair process ( Coordinator ∈ CoordinatorID )
    {
c0:   await cState = "init";
      either {
c1:           cState := "abort"; }
      or {
              cState := "pre-commit";
c2:           either {
                      await allPCommit;
                      cState := "commit"; }
              or {
                      await atLeastOneAbort;
                      goto c1; } ; } ;
    } ;

  fair process ( Participant ∈ Participants )
    {
p0:   await pState[self] = "working";
      either {
p1:           await cState ∈ {"pre-commit", "abort"};
              pState[self] := "aborted";
```

```
                abortFlag := TRUE ; }
        or {
                await cState = "pre-commit";
                pState[self] := "prepared";
p2:             either {
                        await cState = "commit";
                        pState[self] := "committed"; }
                or {
                        await cState = "abort";
                        goto p1; } ; } ;
    } ;
```

BEGIN TRANSLATION
VARIABLES $cState, pState, abortFlag, pc$

$allPCommit \triangleq \forall p \in Participants : pState[p] = \text{"prepared"}$
$atLeastOneAbort \triangleq abortFlag = \text{TRUE}$

$vars \triangleq \langle cState, pState, abortFlag, pc \rangle$

$ProcSet \triangleq \{CoordinatorID\} \cup (Participants)$

$Init \triangleq$
$\land\ cState = \text{"init"}$
$\land\ pState = [p \in Participants \mapsto \text{"working"}]$
$\land\ abortFlag = \text{FALSE}$
$\land\ pc = [self \in ProcSet \mapsto \text{CASE } self = CoordinatorID \to \text{"c0"}$
$\qquad\qquad\qquad\qquad\qquad\qquad \square\ self \in Participants \to \text{"p0"}]$

$c0 \triangleq \land\ pc[CoordinatorID] = \text{"c0"}$
$\land\ cState = \text{"init"}$
$\land\ \lor \land\ pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"c1"}]$
$\qquad \land \text{ UNCHANGED } cState$
$\qquad \lor \land\ cState' = \text{"pre-commit"}$
$\qquad \land\ pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"c2"}]$
$\land \text{ UNCHANGED } \langle pState, abortFlag \rangle$

$c1 \triangleq \land\ pc[CoordinatorID] = \text{"c1"}$
$\land\ cState' = \text{"abort"}$
$\land\ pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"Done"}]$
$\land \text{ UNCHANGED } \langle pState, abortFlag \rangle$

$c2 \triangleq \land\ pc[CoordinatorID] = \text{"c2"}$
$\land\ \lor \land\ allPCommit$
$\qquad \land\ cState' = \text{"commit"}$
$\qquad \land\ pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"Done"}]$
$\qquad \lor \land\ atLeastOneAbort$
$\qquad \land\ pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{"c1"}]$
$\qquad \land \text{ UNCHANGED } cState$
$\land \text{ UNCHANGED } \langle pState, abortFlag \rangle$

$Coordinator \triangleq c0 \lor c1 \lor c2$

$p0(self) \triangleq \land\ pc[self] = \text{"p0"}$
$\land\ pState[self] = \text{"working"}$
$\land\ \lor \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{"p1"}]$

$$
\begin{array}{l}
\quad \land \text{UNCHANGED } pState \\
\quad \lor \land cState = \text{“pre-commit”} \\
\quad\quad \land pState' = [pState \text{ EXCEPT } ![self] = \text{“prepared”}] \\
\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“p2”}] \\
\land \text{UNCHANGED } \langle cState, abortFlag \rangle
\end{array}
$$

$$
\begin{array}{l}
p1(self) \triangleq \land pc[self] = \text{“p1”} \\
\quad \land cState \in \{\text{“pre-commit”}, \text{“abort”}\} \\
\quad \land pState' = [pState \text{ EXCEPT } ![self] = \text{“aborted”}] \\
\quad \land abortFlag' = \text{TRUE} \\
\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\quad \land \text{UNCHANGED } cState
\end{array}
$$

$$
\begin{array}{l}
p2(self) \triangleq \land pc[self] = \text{“p2”} \\
\quad \land \lor \land cState = \text{“commit”} \\
\quad\quad\quad \land pState' = [pState \text{ EXCEPT } ![self] = \text{“committed”}] \\
\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\quad\quad \lor \land cState = \text{“abort”} \\
\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“p1”}] \\
\quad\quad\quad \land \text{UNCHANGED } pState \\
\quad \land \text{UNCHANGED } \langle cState, abortFlag \rangle
\end{array}
$$

$$Participant(self) \triangleq p0(self) \lor p1(self) \lor p2(self)$$

$$
\begin{array}{l}
Terminating \triangleq \land \forall self \in ProcSet : pc[self] = \text{“Done”} \\
\quad\quad\quad\quad\quad \land \text{UNCHANGED } vars
\end{array}
$$

$$Next \triangleq Coordinator \lor (\exists self \in Participants : Participant(self)) \lor Terminating$$

$$Spec \triangleq \land Init \land \Box[Next]_{vars}$$

$$Termination \triangleq \Diamond(\forall self \in ProcSet : pc[self] = \text{“Done”})$$

END TRANSLATION

A.2 $\mathscr{P}_{swap}$ TLA$^+$ Code

MODULE *CrossChainSwap*

EXTENDS *Integers*, *TLAPS*
CONSTANT *NTxs*, *Correct*, *Timeout*

$$
\begin{array}{l}
CStates \triangleq \{\text{“init”}, \text{“pre-commit”}, \text{“commit”}, \text{“abort”}\} \\
PStates \triangleq \{\text{“working”}, \text{“committed”}, \text{“aborted”}, \text{“prepared”}\} \\
AStates \triangleq \{\text{“OwS”}, \text{“OwR”}, \text{“locked”}, \text{“other”}\} \\
SwapStates \triangleq \{\text{“init”}, \text{“correct”}, \text{“different”}\} \\
PublisherID \triangleq -1 \\
CoordinatorID \triangleq 0
\end{array}
$$

$$
\begin{array}{lll}
Sources & \triangleq & \{3 * x - 2 \quad : x \in 1 \,..\, NTxs\} \\
Assets & \triangleq & \{3 * x - 1 \quad : x \in 1 \,..\, NTxs\} \\
Recipients & \triangleq & \{3 \quad * x : x \in 1 \,..\, NTxs\}
\end{array}
$$

$$Pi \triangleq Sources \cup Recipients$$

$Pc \triangleq Pi \cap Correct$
$CSources \triangleq Pc \cap Sources$
$CRecipients \triangleq Pc \cap Recipients$
$BSources \triangleq Sources \setminus CSources$
$BRecipients \triangleq Recipients \setminus CRecipients$

$AofS(x) \triangleq x + 1$
$AofR(x) \triangleq x - 1$
$SofA(x) \triangleq x - 1$
$RofA(x) \triangleq x + 1$

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

--fair algorithm *CrossChainSwap* **{**
variable $coordState = \text{“init”},$
$assets = [a \in Assets \mapsto \text{“OwS”}],$
$pState = \text{“init”},$
$qrm = \{\},$
$qrf = \{\},$
$swapGraph = \text{“init”},$
$ProofPublish = \text{FALSE},$
$ProofOkRM = \text{FALSE},$
$ProofOkRF = \text{FALSE},$
$ProofLock = [s \in Sources \mapsto \text{FALSE}]$ **;**

define {
$ValidTransfer \triangleq qrm = Sources \land \forall s \in Sources : ProofLock[s] = \text{TRUE}$
$AbortTransfer \triangleq qrf \neq \{\}$
}

macro *lockAsset*(*self*) **{**
if ($ProofPublish = \text{TRUE} \land self \in Sources \land assets[AofS(self)] = \text{“OwS”}$ **)**
$assets[AofS(self)] := \text{“locked”}$**;** $ProofLock[self] := \text{TRUE}$**; }**

macro *askRM*(*self*) **{**
if ($self \in Sources \land ProofLock[self] = \text{TRUE} \land coordState = \text{“published”}$ **)**
$qrm := qrm \cup \{self\}$**; }**

macro *askRF*(*self*) **{**
if ($coordState = \text{“published”}$ **)** $qrf := qrf \cup \{self\}$**; }**

macro *retrievingAsset*(*self*) **{**
if ($self \in Recipients \land ProofOkRM = \text{TRUE} \land assets[AofR(self)] = \text{“locked”}$ **)**
$assets[AofR(self)] := \text{“OwR”}$**; }**

macro *recoveringAsset*(*self*) **{**
if ($self \in Sources \land ProofOkRF = \text{TRUE} \land assets[AofS(self)] = \text{“locked”}$ **)**
$assets[AofS(self)] := \text{“OwS”}$**; }**

macro *otherS*(*self*) **{**
if ($self \in Sources \land assets[AofS(self)] = \text{“OwS”}$ **)**

```
 assets[AofS(self)] := "other"; }

macro otherR( self ) {
 if ( self ∈ Recipients ∧ assets[AofR(self)] = "OwR" )
 assets[AofR(self)] := "other"; }

macro directToR( self ) {
 if ( self ∈ Sources ∧ assets[AofS(self)] = "OwS" )
 assets[AofS(self)] := "OwR"; }

macro directToS( self ) {
 if ( self ∈ Recipients ∧ assets[AofR(self)] = "OwR" )
 assets[AofR(self)] := "OwS"; }

process ( Publisher = PublisherID )
{
      init_p : either {
                  pState := "publish";
                  either swapGraph := "correct";
                  or swapGraph := "different"; }
               or skip;
} ;

 fair  process ( Coordinator = CoordinatorID )
   {
   init_c:          await pState = "publish" ∧ swapGraph ≠ "init";
                    coordState := "published";
                    ProofPublish := TRUE;
   decision:        either {
                           await ValidTransfer;
   decisionValid:          coordState := "okRM";
                           ProofOkRM := TRUE;
                           goto Done; }
                    or {
                           await AbortTransfer;
   decisionAbort:          coordState := "okRF";
                           ProofOkRF := TRUE;
                           goto Done; } ;
    } ;

 fair  process ( Source ∈ CSources )

{
   init_src : either {
                  await swapGraph = "different" ∨ Timeout = TRUE;
                  goto Done; }
              or {
                  await ProofPublish = TRUE ∧ swapGraph = "correct";
   lock:          lockAsset(self);
   published:     askRM(self);
   waitForD:      either {
                      await ProofOkRM = TRUE;
                      goto Done; }
                  or {
```

```
                    await ProofOkRF = TRUE;
                    recoveringAsset(self);
                    goto Done; }
                or {
                    await coordState = "published" ∧ Timeout = TRUE;
                    askRF(self);
                    goto waitForD; } ;
                } ;
} ;

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

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

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

 BEGIN TRANSLATION
VARIABLES assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish,
```

$ProofLock, ProofOkRM, ProofOkRF, pc$

$ValidTransfer \triangleq qrm = Sources \land \forall s \in Sources : ProofLock[s] = \text{TRUE}$
$AbortTransfer \triangleq qrf \neq \{\}$

$vars \triangleq \langle assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish, ProofLock,$
$ProofOkRM, ProofOkRF, pc\rangle$

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

$$
\begin{aligned}
Init \triangleq\ & \land assets = [a \in Assets \mapsto \text{“OwS”}] \\
& \land pState = \text{“init”} \\
& \land coordState = \text{“init”} \\
& \land qrm = \{\} \\
& \land qrf = \{\} \\
& \land swapGraph = \text{“init”} \\
& \land ProofPublish = \text{FALSE} \\
& \land ProofLock \quad = [c \in Sources \mapsto \text{FALSE}] \\
& \land ProofOkRM = \text{FALSE} \\
& \land ProofOkRF \ = \text{FALSE} \\
& \land pc = [self \in ProcSet \mapsto \text{CASE } self = PublisherID \to \text{“init\_p”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \square\ self = CoordinatorID \to \text{“init\_c”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \square\ self \in CSources \to \text{“init\_src”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \square\ self \in BSources \to \text{“init\_bsrc”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \square\ self \in CRecipients \to \text{“init\_rcp”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \square\ self \in BRecipients \to \text{“init\_brcp”}]
\end{aligned}
$$

$$
\begin{aligned}
init\_p \triangleq\ & \land pc[PublisherID] = \text{“init\_p”} \\
& \land \lor \land pState' = \text{“publish”} \\
& \qquad \land \lor \land swapGraph' = \text{“correct”} \\
& \qquad\quad \lor \land swapGraph' = \text{“different”} \\
& \quad \lor \land \text{TRUE} \\
& \qquad \land \text{UNCHANGED } \langle pState, swapGraph\rangle \\
& \land pc' = [pc \text{ EXCEPT } ![PublisherID] = \text{“Done”}] \\
& \land \text{UNCHANGED } \langle assets, coordState, qrm, qrf, ProofPublish, \\
& \qquad\qquad ProofLock, ProofOkRM, ProofOkRF\rangle
\end{aligned}
$$

$Publisher \triangleq init\_p$

$$
\begin{aligned}
init\_c \triangleq\ & \land pc[CoordinatorID] = \text{“init\_c”} \\
& \land pState = \text{“publish”} \land swapGraph \neq \text{“init”} \\
& \land coordState' = \text{“published”} \\
& \land ProofPublish' = \text{TRUE} \\
& \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“decision”}] \\
& \land \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofLock, \\
& \qquad ProofOkRM, ProofOkRF\rangle
\end{aligned}
$$

$$
\begin{aligned}
decision \triangleq\ & \land pc[CoordinatorID] = \text{“decision”} \\
& \land \lor \land ValidTransfer \\
& \qquad \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“decisionValid”}] \\
& \quad \lor \land AbortTransfer \\
& \qquad \land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“decisionAbort”}] \\
& \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF\rangle
\end{aligned}
$$

$decisionValid \triangleq \land pc[CoordinatorID] = \text{“decisionValid”}$
$\land coordState' = \text{“okRM”}$
$\land ProofOkRM' = \text{TRUE}$
$\land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“Done”}]$
$\land \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofPublish, ProofLock, ProofOkRF \rangle$

$decisionAbort \triangleq \land pc[CoordinatorID] = \text{“decisionAbort”}$
$\land coordState' = \text{“okRF”}$
$\land ProofOkRF' = \text{TRUE}$
$\land pc' = [pc \text{ EXCEPT } ![CoordinatorID] = \text{“Done”}]$
$\land \text{UNCHANGED } \langle assets, pState, qrm, qrf, swapGraph, ProofPublish, ProofLock, ProofOkRM \rangle$

$Coordinator \triangleq init\_c \lor decision \lor decisionValid \lor decisionAbort$

$init\_src(self) \triangleq \land pc[self] = \text{“init\_src”}$
$\land \lor \land swapGraph = \text{“different”} \lor Timeout = \text{TRUE}$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}]$
$\lor \land ProofPublish = \text{TRUE} \land swapGraph = \text{“correct”}$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“lock”}]$
$\land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle$

$lock(self) \triangleq \land pc[self] = \text{“lock”}$
$\land \text{IF } ProofPublish = \text{TRUE} \land self \in Sources \land assets[AofS(self)] = \text{“OwS”}$
$\text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{“locked”}]$
$\land ProofLock' = [ProofLock \text{ EXCEPT } ![self] = \text{TRUE}]$
$\text{ELSE } \land \text{TRUE}$
$\land \text{UNCHANGED } \langle assets, ProofLock \rangle$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“published”}]$
$\land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, ProofPublish, ProofOkRM, ProofOkRF \rangle$

$published(self) \triangleq \land pc[self] = \text{“published”}$
$\land \text{IF } self \in Sources \land ProofLock[self] = \text{TRUE} \land coordState = \text{“published”}$
$\text{THEN } \land qrm' = (qrm \cup \{self\})$
$\text{ELSE } \land \text{TRUE}$
$\land qrm' = qrm$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“waitForD”}]$
$\land \text{UNCHANGED } \langle assets, pState, coordState, qrf, swapGraph, ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle$

$waitForD(self) \triangleq \land pc[self] = \text{“waitForD”}$
$\land \lor \land ProofOkRM = \text{TRUE}$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}]$
$\land \text{UNCHANGED } \langle assets, qrf \rangle$
$\lor \land ProofOkRF = \text{TRUE}$
$\land \text{IF } self \in Sources \land ProofOkRF = \text{TRUE} \land assets[AofS(self)] = \text{“locked”}$
$\text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{“OwS”}]$
$\text{ELSE } \land \text{TRUE}$
$\land \text{UNCHANGED } assets$
$\land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}]$
$\land qrf' = qrf$

$$
\begin{array}{l}
\quad \lor \land coordState = \text{“published”} \land Timeout = \text{TRUE} \\
\quad\quad \land \text{IF } coordState = \text{“published”} \\
\quad\quad\quad \text{THEN } \land qrf' = (qrf \cup \{self\}) \\
\quad\quad\quad \text{ELSE } \land \text{TRUE} \\
\quad\quad\quad\quad\quad \land qrf' = qrf \\
\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“waitForD”}] \\
\quad\quad \land \text{UNCHANGED } assets \\
\quad \land \text{UNCHANGED } \langle pState, coordState, qrm, swapGraph, \\
\quad\quad\quad\quad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

$$Source(self) \triangleq init\_src(self) \lor lock(self) \lor published(self) \lor waitForD(self)$$

$$
\begin{array}{l}
init\_rcp(self) \triangleq \land pc[self] = \text{“init\_rcp”} \\
\quad \land \lor \land swapGraph = \text{“different”} \lor Timeout = \text{TRUE} \\
\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\quad\quad \lor \land ProofPublish = \text{TRUE} \land swapGraph = \text{“correct”} \\
\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“waitForD\_rcp”}] \\
\quad \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, \\
\quad\quad\quad\quad swapGraph, ProofPublish, ProofLock, \\
\quad\quad\quad\quad ProofOkRM, ProofOkRF \rangle
\end{array}
$$

$$
\begin{array}{l}
waitForD\_rcp(self) \triangleq \land pc[self] = \text{“waitForD\_rcp”} \\
\quad \land \lor \land ProofOkRF = \text{TRUE} \\
\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\quad\quad\quad \land \text{UNCHANGED } \langle assets, qrf \rangle \\
\quad\quad\quad \lor \land ProofOkRM = \text{TRUE} \\
\quad\quad\quad\quad \land \text{IF } self \in Recipients \land ProofOkRM = \text{TRUE} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land assets[AofR(self)] = \text{“locked”} \\
\quad\quad\quad\quad\quad\quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{“OwR”}] \\
\quad\quad\quad\quad\quad\quad \text{ELSE } \land \text{TRUE} \\
\quad\quad\quad\quad\quad\quad\quad\quad \land \text{UNCHANGED } assets \\
\quad\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Done”}] \\
\quad\quad\quad\quad \land qrf' = qrf \\
\quad\quad\quad \lor \land coordState = \text{“published”} \land Timeout = \text{TRUE} \\
\quad\quad\quad\quad \land \text{IF } coordState = \text{“published”} \\
\quad\quad\quad\quad\quad\quad \text{THEN } \land qrf' = (qrf \cup \{self\}) \\
\quad\quad\quad\quad\quad\quad \text{ELSE } \land \text{TRUE} \\
\quad\quad\quad\quad\quad\quad\quad\quad \land qrf' = qrf \\
\quad\quad\quad\quad \land pc' = [pc \text{ EXCEPT } ![self] = \text{“waitForD\_rcp”}] \\
\quad\quad\quad\quad \land \text{UNCHANGED } assets \\
\quad \land \text{UNCHANGED } \langle pState, coordState, qrm, swapGraph, \\
\quad\quad\quad\quad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

$$Recipient(self) \triangleq init\_rcp(self) \lor waitForD\_rcp(self)$$

$$
\begin{array}{l}
init\_bsrc(self) \triangleq \land pc[self] = \text{“init\_bsrc”} \\
\quad \land \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BdirectToR”}] \\
\quad\quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“Bother”}] \\
\quad\quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BaskRM”}] \\
\quad\quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BlockAsset”}] \\
\quad\quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BSaskRF”}] \\
\quad\quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BrecoveringAsset”}] \\
\quad \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, \\
\quad\quad\quad\quad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{array}
$$

$$
\begin{aligned}
BdirectToR(self) \triangleq\ & \land pc[self] = \text{“BdirectToR”} \\
& \land \text{IF } self \in Sources \land assets[AofS(self)] = \text{“OwS”} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{“OwR”}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land \text{UNCHANGED } assets \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$
\begin{aligned}
Bother(self) \triangleq\ & \land pc[self] = \text{“Bother”} \\
& \land \text{IF } self \in Sources \land assets[AofS(self)] = \text{“OwS”} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{“other”}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land \text{UNCHANGED } assets \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$
\begin{aligned}
BaskRM(self) \triangleq\ & \land pc[self] = \text{“BaskRM”} \\
& \land \text{IF } self \in Sources \land ProofLock[self] = \text{TRUE} \land coordState = \text{“published”} \\
& \quad \text{THEN } \land qrm' = (qrm \cup \{self\}) \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land qrm' = qrm \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle assets, pState, coordState, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$
\begin{aligned}
BlockAsset(self) \triangleq\ & \land pc[self] = \text{“BlockAsset”} \\
& \land \text{IF } ProofPublish = \text{TRUE} \land self \in Sources \land assets[AofS(self)] = \text{“OwS”} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] \quad = \text{“locked”}] \\
& \qquad\quad \land ProofLock' = [ProofLock \text{ EXCEPT } ![self] = \text{TRUE}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land \text{UNCHANGED } \langle assets, ProofLock \rangle \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$
\begin{aligned}
BrecoveringAsset(self) \triangleq\ & \land pc[self] = \text{“BrecoveringAsset”} \\
& \land \text{IF } self \in Sources \land ProofOkRF = \text{TRUE} \land assets[AofS(self)] = \text{“locked”} \\
& \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofS(self)] = \text{“OwS”}] \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land \text{UNCHANGED } assets \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$
\begin{aligned}
BSaskRF(self) \triangleq\ & \land pc[self] = \text{“BSaskRF”} \\
& \land \text{IF } coordState = \text{“published”} \\
& \quad \text{THEN } \land qrf' = (qrf \cup \{self\}) \\
& \quad \text{ELSE } \land \text{TRUE} \\
& \qquad\quad \land qrf' = qrf \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_bsrc”}] \\
& \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, swapGraph, \\
& \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle
\end{aligned}
$$

$$BSource(self) \triangleq init\_bsrc(self) \lor BdirectToR(self) \lor Bother(self) \lor BaskRM(self) \lor BlockAsset(self) \lor BSaskRF(self) \lor BrecoveringAsset(self)$$

$$\begin{aligned} init\_brcp(self) \triangleq\ & \land pc[self] = \text{“init\_brcp”} \\ & \land \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BRaskRF”}] \\ & \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BRretrievingAsset”}] \\ & \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BRdirectToS”}] \\ & \quad \lor \land pc' = [pc \text{ EXCEPT } ![self] = \text{“BRother”}] \\ & \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, qrf, swapGraph, \\ & \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle \end{aligned}$$

$$\begin{aligned} BRdirectToS(self) \triangleq\ & \land pc[self] = \text{“BRdirectToS”} \\ & \land \text{IF } self \in Recipients \land assets[AofR(self)] = \text{“OwR”} \\ & \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{“OwS”}] \\ & \quad \text{ELSE } \land \text{TRUE} \\ & \qquad \land \text{UNCHANGED } assets \\ & \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_brcp”}] \\ & \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\ & \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle \end{aligned}$$

$$\begin{aligned} BRother(self) \triangleq\ & \land pc[self] = \text{“BRother”} \\ & \land \text{IF } self \in Recipients \land assets[AofR(self)] = \text{“OwR”} \\ & \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{“other”}] \\ & \quad \text{ELSE } \land \text{TRUE} \\ & \qquad \land \text{UNCHANGED } assets \\ & \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_brcp”}] \\ & \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\ & \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle \end{aligned}$$

$$\begin{aligned} BRretrievingAsset(self) \triangleq\ & \land pc[self] = \text{“BRretrievingAsset”} \\ & \land \text{IF } self \in Recipients \land ProofOkRM = \text{TRUE} \\ & \qquad \land assets[AofR(self)] = \text{“locked”} \\ & \quad \text{THEN } \land assets' = [assets \text{ EXCEPT } ![AofR(self)] = \text{“OwR”}] \\ & \quad \text{ELSE } \land \text{TRUE} \\ & \qquad \land \text{UNCHANGED } assets \\ & \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_brcp”}] \\ & \land \text{UNCHANGED } \langle pState, coordState, qrm, qrf, swapGraph, \\ & \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle \end{aligned}$$

$$\begin{aligned} BRaskRF(self) \triangleq\ & \land pc[self] = \text{“BRaskRF”} \\ & \land \text{IF } coordState = \text{“published”} \\ & \quad \text{THEN } \land qrf' = (qrf \cup \{self\}) \\ & \quad \text{ELSE } \land \text{TRUE} \\ & \qquad \land qrf' = qrf \\ & \land pc' = [pc \text{ EXCEPT } ![self] = \text{“init\_brcp”}] \\ & \land \text{UNCHANGED } \langle assets, pState, coordState, qrm, swapGraph, \\ & \qquad ProofPublish, ProofLock, ProofOkRM, ProofOkRF \rangle \end{aligned}$$

$$BRecipient(self) \triangleq init\_brcp(self) \lor BRaskRF(self) \lor BRetrievingAsset(self) \lor BRother(self) \lor BRdirectToS(self)$$

$$\begin{aligned} Next \triangleq\ & Publisher \lor Coordinator \\ & \lor (\exists self \in CSources : Source(self)) \\ & \lor (\exists self \in BSources : BSource(self)) \\ & \lor (\exists self \in CRecipients : Recipient(self)) \end{aligned}$$

$$\vee\ (\exists\, self \in BRecipients : BRecipient(self))$$

$$\begin{aligned} Spec \triangleq\ & \wedge\, Init \wedge \Box[Next]_{vars} \\ & \wedge\, \mathrm{WF}_{vars}(Next) \\ & \wedge\, \mathrm{WF}_{vars}(Coordinator) \\ & \wedge\, \forall\, self \in CSources : \mathrm{WF}_{vars}(Source(self)) \\ & \wedge\, \forall\, self \in CRecipients : \mathrm{WF}_{vars}(Recipient(self)) \end{aligned}$$

END TRANSLATION

References

[1] 101 blockchains. https://101blockchains.com. 38

[2] Azure cosmos db. https://github.com/Azure/azure-cosmos-tla. 71

[3] Dexter flaw. https://forum.tezosagora.org/t/dexter-flaw-discovered-funds-are-safe/2742. 58

[4] Ethereum foundation : Ethereum and oracles. https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/. 100

[5] Foley. https://www.foley.com/en/. 38

[6] Medium. https://medium.com. 38

[7] Parity. https://www.parity.io. 7, 19

[8] Parity wallet hack. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. 7, 19

[9] Tendermint. https://tendermint.com. 185, 186, 188

[10] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018. 44, 45, 46, 47

[11] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, et al. The key tool. *Software & Systems Modeling*, 4(1):32–54, 2005. 48

[12] Wolfgang Ahrendt, Richard Bubel, Joshua Ellul, Gordon J Pace, Raúl Pardo, Vincent Rebiscoul, and Gerardo Schneider. Verification of smart contract business logic. In *International Conference on Fundamentals of Software Engineering*, pages 228–243. Springer, 2019. 47, 49, 50

[13] Algorand. Algorand virtual machine. https://developer.algorand.org/docs/get-details/dapps/avm/. 195

[14] Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the tezos blockchain. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 1–10. IEEE, 2019. 49, 185

[15] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987. 34

[16] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993. 46

[17] Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis*, pages 1–18, Cham, 2014. Springer International Publishing. 115

[18] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018. 47, 48, 49, 50

[19] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019. 52, 57, 58

[20] Emmanuelle Anceaume, Antonella Del Pozzo, Romaric Ludinard, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Blockchain abstract data type. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 349–358, 2019. 38

[21] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018. 7, 18, 40, 48, 185, 188

[22] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer, 2015. 41, 185

[23] Martin Arrivets. Monet: Mobile ad hoc blockchains, 2018. 41, 185, 186, 188

[24] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust*, pages 164–186. Springer, 2017. 7, 19, 44, 51, 97, 98, 104, XXVII

[25] Babble. Babble consensus. https://github.com/mosaicnetworks/babble. 41

[26] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th international conference on software and computer applications*, pages 322–326, 2018. 44, 45, 46, 47

[27] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008. 11, 22

[28] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016. 30

[29] Leemon Baird, Mance Harmon, and Paul Madsen. Hedera: A public hashgraph network & governing council. *White Paper*, 1, 2019. 30

[30] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer, 2011. 48, 65, 72

[31] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. Ieee, 2006. 45

[32] Bernhard Beckert, Jonas Schiffl, and Mattias Ulbrich. Smart contracts: application scenarios for deductive program verification. In *International Symposium on Formal Methods*, pages 293–298. Springer, 2019. 47, 48, 50

[33] Robert Beers. Pre-rtl formal verification: an intel experience. In *Proceedings of the 45th Annual Design Automation Conference*, pages 806–811, 2008. 71

[34] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. 2006. 46

[35] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *arXiv preprint arXiv:2005.14282*, 2020. 8, 20

[36] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. In *International Symposium on Formal Methods*, pages 368–379. Springer, 2019. 47, 49, 50

[37] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987. 28, 53, 126

[38] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016. 44, 47, 49, 50

[39] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007. 30

[40] Matthew Black, Tingwei Liu, and Tony Cai. Atomic loans: Cryptocurrency debt instruments. *arXiv preprint arXiv:1901.05117*, 2019. 53

[41] F Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The alt-ergo automated theorem prover, 2008. 65

[42] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 151–165. Springer, 2007. 72

[43] Michael Borkowski, Christoph Ritzer, Daniel McDonald, and Stefan Schulte. Caught in chains: Claim-first transactions for cross-blockchain asset transfers. *Technische Universität Wien, Whitepaper*, 2018. 56, 57, 58

[44] Rory Bowden, Holger Paul Keeler, Anthony E Krzesinski, and Peter G Taylor. Block arrivals in the bitcoin blockchain. *arXiv preprint arXiv:1801.07447*, 2018. 36

[45] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975. 50

[46] Vıctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Zeus: A distributed timed model-checker based on kronos. *Electronic notes in theoretical computer science*, 68(4):503–522, 2002. 45

[47] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016. 40

[48] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016. 185

[49] Daniel Burkhardt, Maximilian Werling, and Heiner Lasi. Distributed ledger. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–9, 2018. 29

[50] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014. 4, 5, 16, 17, 178, 185

[51] Cardano. Cardano blockchain. https://docs.cardano.org/. 185

[52] Cardano. Plutus. the scripting language embedded in the cardano ledger. https://github.com/input-output-hk/plutus. 186

[53] Glenn Carl, George Kesidis, Richard R Brooks, and Suresh Rai. Denial-of-service attack-detection techniques. *IEEE Internet computing*, 10(1):82–89, 2006. 36

[54] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999. 6, 17, 37, 188

[55] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. Nusmv 2.4 user manual. *CMU and ITC-irst*, 2005. 11, 22, 45

[56] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010. 72

[57] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016. 6, 18

[58] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999. 5, 10, 11, 17, 22, 150

[59] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 94–109. Springer, 2015. 103

[60] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *International Conference on Computer Aided Verification*, pages 718–724. Springer, 2012. 46

[61] Sylvain Conchon, Alexandrina Korneva, and Fatiha Zaïdi. Verifying smart contracts with cubicle. In *International Symposium on Formal Methods*, pages 312–324. Springer, 2019. 44, 46, 47

[62] Coq. The coq proof assistant. https://coq.inria.fr. 48, 49, 65

[63] Patrick Cousot. The role of abstract interpretation in formal methods. pages 135 – 140, 10 2007. 11, 22

[64] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012. 10, 22

[65] Luís Pedro Arrojado da Horta, João Santos Reis, Mário Pereira, and Simão Melo de Sousa. Whylson: Proving your michelson smart contracts in why3. *arXiv preprint arXiv:2005.14650*, 2020. 47, 49, 50

[66] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain. 2018. 6, 17, 36, 37

[67] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 48, 65, 72

[68] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015. 52, 58

[69] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. 64

[70] Ian Domowitz. A taxonomy of automated trade execution systems. *Journal of International Money and Finance*, 12:607–631, 1993. 109

[71] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002. 36

[72] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22:84, 1997. 9, 20

[73] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, pages 131–136. IEEE, 2003. 28

[74] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014. 48

[75] Ariel Ekblaw, Asaph Azaria, John D Halamka, and Andrew Lippman. A case study for blockchain in healthcare:medrec prototype for electronic health records and medical research data. In *Proceedings of IEEE open & big data conference*, volume 13, page 13, 2016. 4, 16

[76] S Epp. Proof issues with existential quantification. In *Proceedings of the ICMI study 19 conference: Proof and Proving in Mathematics Education*, volume 1, pages 154–159, 2009. 74

[77] Ethereum. Btc relay. https://github.com/ethereum/btcrelay. 53, 56

[78] Ethereum. Ethereum foundation. the solidity contract-oriented programming language. https://github.com/ethereum/solidity. 6, 18, 44, 96, 186

[79] Ethereum. Simple payment verification (spv). https://electrum.readthedocs.io/en/latest/spv.html. 56

[80] Kuan Fan, Zijian Bao, Mingxi Liu, Athanasios V Vasilakos, and Wenbo Shi. Dredas: Decentralized, reliable and efficient remote outsourced data auditing scheme with blockchain smart contract for industrial iot. *Future Generation Computer Systems*, 110:665–674, 2020. 44

[81] Yi Fang, Kenneth L McMillan, Amir Pnueli, and Lenore D Zuck. Liveness by invisible invariants. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 356–371. Springer, 2006. 196

[82] Antonio Fernández Anta, Chryssis Georgiou, Maurice Herlihy, and Maria Potop-Butucaru. Principles of blockchain systems. *Synthesis Lectures on Computer Science*, (0):1–213, 2021. 37

[83] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013. 10, 22, 65, 96

[84] George Foroglou and Anna-Lali Tsilidou. Further applications of the blockchain. *Columbia University PhD in Sustainable Development*, 10, 2015. 4, 16

[85] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *International Conference on Formal Engineering Methods*, pages 581–596. Springer, 2010. 11, 22

[86] Daniel Fullmer and A Stephen Morse. Analysis of difficulty control in bitcoin and proof-of-work blockchains. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 5988–5992. IEEE, 2018. 36

[87] Peter Gammie and Ron van der Meyden. Mck: Model checking the logic of knowledge. In *International Conference on Computer Aided Verification*, pages 479–483. Springer, 2004. 58

[88] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1257–1262. IEEE, 2020. 53, 56, 57, 58

[89] Peter Gaži, Aggelos Kiayias, and Dionysis Zindros. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 139–156. IEEE, 2019. 56

[90] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. Introduction to iot. *International Advanced Research Journal in Science, Engineering and Technology*, 5(1):41–44, 2018. 30

[91] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994. 127

[92] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, mar 2006. 28

[93] Gideon Greenspan. Multichain. https://www.multichain.com/download/MultiChain-White-Paper.pdf. 42, 185, 188

[94] BitFury Group. Public versus private blockchains. https://bitfury.com/content/downloads/public-vs-private-pt1-1.pdf, 2015. 38

[95] Joël Gugger. Bitcoin-monero cross-chain atomic swap. *IACR Cryptol. ePrint Arch.*, 2020:1126, 2020. 52, 53, 55, 57, 58

[96] Kim Guldstrand Larsen, Paul Pettersson, and Wang yi. Uppaal in a nutshell. 1:134–152, 12 1997. 11, 22

[97] Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Fault-Tolerant Distributed Computing*, pages 201–208. Springer, 1990. 28

[98] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 161–179. Springer, 2019. 44, 47, 48, 49, 50

[99] Yucen He, Xinyi Zhu, Fangfang Xu, Yulu Wu, Xiang Fan, Xin Cui, Xiangrui Kong, and Bobinson Kalarikkal Bobby. A novel cross-chain mechanism for blockchains. In *International Conference on Smart Blockchain*, pages 139–148. Springer, 2018. 57, 58, 122

[100] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on {Bitcoin's}{Peer-to-Peer} network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015. 57

[101] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018. 52, 53, 54, 57, 58, 59, 122, 124

[102] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain deals and adversarial commerce. *Proceedings of the VLDB Endowment*, 13(2):100–113, 2019. 52, 53, 54, 57, 58, 122, 124, 177

[103] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017. 47, 48, 49, 50

[104] Yoichi Hirai. Blockchains as kripke models: An analysis of atomic cross-chain swap. In *International Symposium on Leveraging Applications of Formal Methods*, pages 389–404. Springer, 2018. 51, 122

[105] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 65

[106] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003. 11, 22, 44

[107] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004. 10, 21, 62, 63

[108] Soichiro Imoto, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Atomic cross-chain swaps with improved space and local time complexity. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 194–208. Springer, 2019. 52, 58, 59

[109] Bart Jacobs and Frank Piessens. The verifast program verifier. Technical report, Technical Report CW-520, Department of Computer Science, Katholieke ..., 2008. 10, 22

[110] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994. 28, 32

[111] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018. 44, 45, 46, 47

[112] Dmitry Khovratovich and Jason Law. Sovrin: digital identities in the blockchain era. *Github Commit by jasonalaw October*, 17:38–99, 2017. 41

[113] Igor Konnov, Marijana Lazić, Ilina Stoilkovska, and Josef Widder. Tutorial: Parameterized verification with byzantine model checker. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 189–207. Springer, 2020. 58

[114] Sakthi Kumaresh. Academic blockchain: An application of blockchain technology in education system. In Neha Sharma, Amlan Chakrabarti, Valentina Emilia Balas, and Jan Martinovic, editors, *Data Management, Analytics and Innovation*, pages 435–448, Singapore, 2021. Springer Singapore. 44

[115] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014. 178

[116] Jae Kwon and Ethan Buchman. Cosmos whitepaper. *A Netw. Distrib. Ledgers*, 2019. 185, 186, 188

[117] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, and Yongdae Kim. Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash? In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 935–951. IEEE, 2019. 37

[118] Nomadic Labs. Emmy+: an improved consensus algorithm. https://research-development.nomadic-labs.com/emmy-an-improved-consensus-algorithm.html. 187

[119] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994. 71

[120] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002. 11, 22, 59

[121] Leslie Lamport. The pluscal algorithm language. In *International Colloquium on Theoretical Aspects of Computing*, pages 36–60. Springer, 2009. 72

[122] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019. 28, 31

[123] Leslie Lamport and Lawrence C Paulson. Should your specification language be typed. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):502–526, 1999. 82

[124] Leslie Lamport and Fred B Schneider. The"hoare logic"of csp, and all that. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):281–296, 1984. 33

[125] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019. 30, 37, 55, 178, 188

[126] Rongjian Lan, Ganesha Upadhyaya, Stephen Tse, and Mahdi Zamani. Horizon: A gas-efficient, trustless bridge for cross-chain transactions. *arXiv preprint arXiv:2101.06000*, 2021. 53, 55, 58

[127] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420. Citeseer, 1998. 49

[128] Matthias Lehmann. Who owns bitcoin: Private law facing the blockchain. *Minn. JL Sci. & Tech.*, 21:93, 2019. 34

[129] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008. 48

[130] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010. 10, 22

[131] Michael Leuschel, Michael Butler, et al. Prob: A model checker for b. In *FME*, volume 2805, pages 855–874. Springer, 2003. 11, 22

[132] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, volume 49, pages 227–238. ACM, 2014. 103

[133] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016. 44, 51, 97

[134] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996. 9, 20, 28, 31

[135] Laurent Mauborgne. Astrée: Verification of absence of runtime error. In *Building the Information Society*, pages 385–392. Springer, 2004. 103

[136] Roger Maull, Phil Godsiff, Catherine Mulligan, Alan Brown, and Beth Kewell. Distributed ledger technology: Applications and implications. *Strategic Change*, 26(5):481–489, 2017. 28

[137] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018. 51

[138] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993. 45

[139] Esther Mengelkamp, Johannes Gärttner, Kerstin Rock, Scott Kessler, Lawrence Orsini, and Christof Weinhardt. Designing microgrid energy markets: a case study: the brooklyn microgrid. *Applied Energy*, 2017. 105

[140] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016. 188

[141] Yiannis Moschovakis. Paradoxes and axioms. *Notes on Set Theory*, pages 19–31, 2006. 71

[142] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, 2014. 48

[143] Multi-chain. Smart filters. https://www.multichain.com/developers/smart-filters/. 187

[144] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019. 4, 5, 16, 17, 30, 36, 37, 38, 178, 185

[145] ES Negara, AN Hidyanto, R Andryani, and D Erlansyah. A survey blockchain and smart contract technology in government agencies. In *IOP Conference Series: Materials Science and Engineering*, volume 1071, page 012026. IOP Publishing, 2021. 44

[146] Zeinab Nehaï and François Bobot. Deductive proof of industrial smart contracts using why3. In *International Symposium on Formal Methods*, pages 299–311. Springer, 2019. 12, 23, 51

[147] Zeinab Nehaï, François Bobot, Sara Tucci-Piergiovanni, Carole Delporte-Gallet, and Hugues Fauconnier. A tla+ formal proof of a cross-chain swap. In *23rd International Conference on Distributed Computing and Networking*, pages 148–159, 2022. 13, 24, 59

[148] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. Model-checking of smart contracts. In *The 2018 IEEE International Conference on Blockchain*. IEEE, 2018. 44, 45, 46, 47, 105, 117

[149] Neo. Neo virtual machine. https://docs.neo.org/docs/en-us/basic/neovm.html. 195

[150] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015. 71

[151] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. 48

[152] Haroon Shakirat Oluwatosin. Client-server model. *IOSRJ Comput. Eng*, 16(1):2278–8727, 2014. 28

[153] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014. 40

[154] Thomas Osterland and Thomas Rose. Model checking smart contracts for ethereum. *Pervasive and Mobile Computing*, 63:101129, 2020. 44, 45, 46, 47

[155] Woong Sub Park, Hyuk Lee, and Jin-Young Choi. Formal modeling of smart contract-based trading system. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pages 48–52. IEEE, 2022. 44, 46, 47

[156] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977. 63

[157] Julien Polge, Jérémy Robert, and Yves Le Traon. Permissioned blockchain frameworks in the industry: A comparison. *Ict Express*, 7(2):229–233, 2021. 38

[158] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. 39

[159] Serguei Popov. The tangle. *White paper*, 1(3), 2018. 30

[160] Tahmid Hasan Pranto, Abdulla All Noman, Atik Mahmud, and AKM Bahalul Haque. Blockchain and smart contract for iot enabled smart agriculture. *PeerJ Computer Science*, 7:e407, 2021. 7, 18

[161] Quorum. Consensys quorum. https://github.com/ConsenSys/quorum/blob/master/docs/Quorum20Whitepaper20v0.2.pdf. 40, 185, 188

[162] Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, and Flemming Nielson. The logic of xacml. *Science of Computer Programming*, 83:80–105, 2014. 45

[163] Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Elsevier, 2001. 5, 17, 150

[164] Jesus Ruiz. *Public-permissioned blockchains as common-pool resources*. PhD thesis, Alastria Blockchain Ecosystem, 2020. 38

[165] Brahmananda Sapkota, Dumitru Roman, Sebastian Ryszard Kruk, and Dieter Fensel. Distributed web service discovery architecture. In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, pages 136–136. IEEE, 2006. 28

[166] Fabian Schär. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review*, 2021. 4, 7, 16, 18

[167] A Secure. The zilliqa project: A secure, scalable blockchain platform. 2018. 178, 185, 188

[168] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018. 186

[169] Narges Shadab, Farzin Houshmand, and Mohsen Lesani. Cross-chain transactions. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020. 52, 58, 122

[170] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008. 48

[171] Ralf Steinmetz and Klaus Wehrle. *Peer-to-peer systems and applications*, volume 3485. Springer, 2005. 28

[172] Nick Szabo. Smart contracts. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. 6, 18, 186

[173] Tezos. Michelson: the language of smart contracts in tezos. https://tezos.gitlab.io/active/michelson.html. 7, 18, 49, 186

[174] Shobha Tyagi and Madhumita Kathuria. *Study on Blockchain Scalability Solutions*, page 394–401. Association for Computing Machinery, New York, NY, USA, 2021. 39

[175] Ron van der Meyden. On the specification and verification of atomic swap smart contracts. *arXiv preprint arXiv:1811.06099*, 2018. 58

[176] Rob van Glabbeek, Vincent Gramoli, and Pierre Tholoniat. Cross-chain payment protocols with success guarantees. *arXiv preprint arXiv:1912.04513*, 2019. 52, 56, 57, 58, 122

[177] Lauren van Haaften-Schick and Amy Whitaker. From the artist's contract to the blockchain ledger: New forms of artists' funding using equity and resale royalties. *Available at SSRN 3842210*, 2021. 44

[178] Pavel Vasin. Blackcoin's proof-of-stake protocol v2. *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, 71, 2014. 6, 17, 36, 37, 178

[179] Eric Verhulst, Raymond T Boute, José Miguel Sampaio Faria, Bernhard HC Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems*. Springer Science & Business Media, 2011. 71

[180] Meng Wang, Junfeng Tian, and Hong Zhang. Umc4m: A verification tool via program execution. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 88–98. Springer, 2019. 46

[181] Xiaobing Wang, Cong Tian, Zhenhua Duan, and Liang Zhao. Msvl: a typed language for temporal logic programming. *Frontiers of Computer Science*, 11:762–785, 2016. 46

[182] Xiaobing Wang, Xiaoyu Yang, and Chunyi Li. A formal verification method for smart contract. In *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, pages 31–36. IEEE, 2020. 44, 46, 47

[183] Peter Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996. 8, 19

[184] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008. 72

[185] The *Why3* development team. *Why3* documentation - release 1.4.0. http://why3.lri.fr/manual.pdf. 69, XXVII

[186] Wiki. Hash time locked contracts. https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts. 52

[187] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014. 4, 6, 16, 18, 38, 97, 104, 113, 116, XXVII

[188] Brent Xu, Dhruv Luthra, Zak Cole, and Nate Blakely. Eos: An architectural, performance, and economic analysis. *Retrieved June*, 11:2019, 2018. 41, 185, 188

[189] Fan Yang, Wei Zhou, QingQing Wu, Rui Long, Neal N Xiong, and Meiqi Zhou. Delegated proof of stake with downgrade: A secure and efficient blockchain consensus algorithm with downgrade mechanism. *IEEE Access*, 7:118541–118555, 2019. 36

[190] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999. 11, 22, 71

[191] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847*, 2019. 9, 20, 52, 53, 54, 57, 58, 124, 126

[192] Victor Zakhary, Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. Towards global asset management in blockchain systems. *arXiv preprint arXiv:1905.09359*, 2019. 35

[193] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William Knottenbelt. Xclaim: Trustless, interoperable, cryptocurrency-backed assets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 193–210. IEEE, 2019. 8, 20, 53, 55, 58

[194] Zcash. zk-snarks. https://z.cash/technology/zksnarks/. 56

[195] Jean-Yves Zie, Jean-Christophe Deneuville, Jérémy Briffaut, and Benjamin Nguyen. Extending atomic cross-chain swaps. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 219–229. Springer, 2019. 52, 53, 57, 58, 122

List of Figures

List of Tables

Listings