Towards correct Blockchain-based business processes

Ikram Garfatta

Université paris XIII - Sorbonne Paris Nord
École Doctorale Sciences, Technologies, Santé Galilée

Université de Tunis El Manar
École Doctorale Sciences et Techniques de l'ingénieur

Vers des Processus Métier Corrects basés sur la Blockchain

(Towards Correct Blockchain-based Business Processes)

Thèse de Doctorat
présentée par :

Ikram GARFATTA

pour l'obtention du grade de :
Docteur en Informatique

soutenue le 13 Décembre 2022 devant le jury d'examen composé de :

BEN AYED Leila,	ENSI Université de la Manouba	(Examinatrice)
GAALOUL Walid,	Institut Polytechnique de Paris	(Examinateur)
GHODOUS Parisa,	Université de Lyon	(Rapportrice)
GRAÏET Mohamed,	Université de Monastir	(Directeur)
KLAI Kaïs,	Université Sorbonne Paris Nord	(Directeur)
MATULEVICIUS Raimundas,	Université de Tartu	(Rapporteur)
PERNELLE Nathalie,	Université Sorbonne Paris Nord	(Présidente)
TUCCI Sara,	CEA LIST	(Examinatrice)

Année académique 2021–2022

"Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little."

Bertrand Meyer

Acknowledgment

It is difficult to put into words how much gratitude I feel towards the people who have helped me along the way in my journey towards obtaining this PhD degree. It is the least I can do to dedicate these few words to them.

First and foremost, my gratitude goes to my amazing supervisors, Kaïs KLAI and Mohamed GRAÏET, who have been incredibly supportive and encouraging throughout the course of my research and without whom none of it would have been possible. Their guidance, expertise, and wisdom have been invaluable, and I could not have achieved this without them.

To Kaïs KLAI, thank you for your brilliant insights, tireless dedication, and exceptional mentorship. Your expertise in formal verification and your ability to challenge and inspire me have been instrumental in shaping this work. But most importantly, thank you for simply being the great person that you are, and as clichéd as this may sound, being the role model that I needed. You have definitely left a mark on me that I will forever cherish.

To Mohamed GRAÏET, thank you for your continuous encouragement, constant support and outstanding teaching. Your knowledge of software engineering and formal methods have helped me develop a rigorous and systematic approach to my research. Thank you for being the one who pushed me out my comfort zone when I needed it and provided me with the opportunities to have the best research experiences.

Together, you have offered me with a unique perspective and a wealth of knowledge that has been essential in the successful completion of this work. I am truly grateful for having crossed your paths.

I am grateful to the members of my thesis committee, who have generously contributed their time and expertise to review and evaluate this work.

To Prof. Nathalie PERNELLE, Prof. Leila BEN AYED, Prof. Sara TUCCI, thank you for your brilliant comments and suggestions that have helped me develop a deeper understanding of the challenges and opportunities in my field of study.

I would like to express my deepest gratitude to Prof. Walid GAALOUL who has been really invested in different stages of this research. Thank you for your constructive feedback and critical evaluation of this work which have been instrumental in improving my work, as well as your exceptional guidance, support, and mentorship throughout my doctoral journey.

To Prof. Parisa GHODOUS and Prof. Raimundas MATULEVICIUS, thank you for your valuable insights and feedback which can only testify of a great and meticulous attention to detail, and which have helped me refine my work and improve the quality of this manuscript.

I would like to thank the staff and members of the computer science department of USPN who have created a supportive and intellectually stimulating environment that allowed me to flourish as a researcher. I would like to also thank their counterparts at the ENIT who have been the most accommodating and helpful with all of my administrative requests despite my constant tendency to flirt with the deadlines.

To my loving family, especially my mother Chadlia, who has been my rock, my constant source of love and encouragement, and who has sacrificed so much for me. Thank you for always believing in me, even when I didn't believe in myself..

To my father Hassen, who instilled in me a love for learning and a never-give-up attitude and has always been a source of inspiration to me. Your everlasting memory in my heart has guided me through some of the toughest times in my life. I hope I have made you proud..

To my dearest sisters and brothers who have been there for me through thick and thin as well as their partners and kids. Thank you for your unconditional love, support, and laughter. I am blessed to have you all in my life.

And last but not least, to my colleagues and friends, who I will most certainly fail to mention all.

To those who were at the end of their PhD journeys when I started mine: the big-hearted Hayet who was amongst the first people I ever met when I first arrived in France and who has always been there for me whenever I needed her, the super-friendly Mehdi who has been a great help for the socially-inept me to integrate the lab and much more, and the kind-hearted Enrico, Davide, Jawher and Hiba with whom I shared my first PhD memories (and long walks in Paris).

To those who were there throughout this journey, to the cheerful Noor and delightful Yasmine who never failed to bring a smile to my face, to Alec-the-Great who never failed to bring a scowl to it (though it is not much of a challenge, I confess), and to the charming Niama, gracious Reda, athletic Dina and vampirish Aloÿs with whom I shared many laughs. Thank you for your unwavering support, hilarious memes, and constant reminders to take breaks and enjoy life. You have made this journey so much more fun and bearable.

To those who were there before the journey even started, to the lively Marwa, the eloquent Hamza, the warm-hearted Ahmed and the classy Mohamed Amine. Thank you for constantly remembering I exist, and putting up with my frequent radio silence. Distance may have tried to drive a wedge between us but you are always on my mind (I am just not that good at showing it).

To all of you who have contributed to this work, I hope this manuscript does you justice. And to those who didn't contribute at all, but are somehow reading this anyway: thanks for stopping by!

Résumé

La blockchain, qui a d'abord été révélée comme la technologie à l'origine du bitcoin, a dépassé le cadre des crypto-monnaies pour s'étendre à un large éventail de domaines d'application, dont la gestion des processus métier (BPM). En effet, ses propriétés intrinsèques, telles que sa structure décentralisée, sa capacité à donner confiance à des parties non dignes de confiance, son immuabilité et sa transparence financière, semblent fournir les instruments nécessaires pour concevoir des solutions adaptées aux problèmes actuels de la BPM, en particulier pour les collaborations. Cette évolution est principalement due à l'introduction du concept de contrat intelligent dans les blockchains. Un contrat intelligent permet l'exécution de séquences de transactions interdépendantes tout en respectant les règles qui y sont établies. D'autre part, un processus métier peut être considéré comme un ensemble d'activités interconnectées par des relations causales dans le but d'atteindre un objectif métier. Par conséquent, les contrats intelligents semblent être d'excellents candidats pour la mise en œuvre et l'automatisation des processus métier (BPs).

Malgré les avancées significatives dans l'adoption de la Blockchain pour le BPM, la technologie n'en est encore qu'à ses débuts, et le déploiement des contrats intelligents pour la mise en œuvre des BPs ne peut être considéré comme sûr. Par conséquent, prouver la correction des contrats intelligents à déployer sur une blockchain est crucial pour l'intégrité des processus métier spécifiés.

Dans notre travail, nous proposons une approche formelle basée sur la transformation des contrats intelligents écrits en Solidity, en tenant compte du contexte BPM dans lequel ils sont utilisés, en un réseau de Petri coloré hiérarchique. Nous exprimons un ensemble de vulnérabilités des contrats intelligents sous forme de formules en logique temporelle et utilisons le vérificateur de modèles *Helena* pour, non seulement détecter ces vulnérabilités tout en discernant leur exploitabilité, mais aussi vérifier d'autres propriétés spécifiques aux contrats basées sur le temps.
L'approche que nous proposons est basée sur la vérification des modèles représentés en CPN et comprend principalement trois phases:

1. transformer le code Solidity des contrats intelligents en sous-modèles CPN correspondant à leurs fonctions.

2. transformer le contexte BPM en un modèle CPN.

3. construire un modèle CPN en référence à une propriété LTL qui peut exprimer: *(i)* une vulnérabilité dans le code ou *(ii)* une propriété spécifique au contrat, en le liant à un modèle CPN représentant le comportement à considérer, et en fournissant le modèle obtenu au vérificateur de modèles pour vérifier la propriété ciblée.

Plus précisément, nous optons pour un modèle CPN hiérarchique pour représenter l'exécution et l'interaction des contrats intelligents considérés dans le cadre de la spécification du contexte BPM éventuellement fournie. Pour ce faire, nous représentons chaque fonction d'un contrat intelligent par une *transition agrégée* qui encapsule un sous-modèle correspondant au flux de travail interne de la première. En fait, notre objectif à cette première étape est d'obtenir des blocs de construction pour le modèle hiérarchique qui sera fourni au vérificateur de modèles. Ensuite, étant donné une spécification de contexte (transformée en CPN) et une propriété LTL à vérifier, le modèle CPN final est construit en *(1)* reliant la transition agrégée représentant la fonction ciblée au modèle comportemental et *(2)* construisant une hiérarchie en représentant explicitement les appels de fonction dans le sous-modèle en question (si la propriété vérifiée le requiert).

Nous avons implémenté un outil graphique appelé *Solidity2CPN* qui met en œuvre et automatise les différentes étapes de l'approche proposée et la rend accessible à un plus grand nombre d'utilisateurs qui ne sont pas nécessairement familiers avec les aspects de la vérification formelle.

Abstract

Blockchain, which was first introduced as the technology driving Bitcoin, has now outgrown the confines of cryptocurrencies to find its way into a wide range of application areas, including Business Process Management (BPM). Indeed, its intrinsic properties, such as its decentralized structure, capacity to give trust among untrustworthy parties, immutability, and financial transparency, appear to provide the necessary instruments for devising suitable solutions for current BPM issues, particularly for collaborations. This evolution has been mainly owed to the concept of smart contracts being introduced to Blockchains. A smart contract allows the execution of interdependent transaction sequences while adhering to the rules established in it. On the other hand, a business process may be considered as a set of activities connected by causal relationships with the purpose of attaining a business goal. As a result, smart contracts appear to be excellent candidates for implementing and automating BPs.

Despite significant advancements in Blockchain adoption for BPM, the technology is still in its infancy, and deploying smart contracts to carry out BPs cannot be deemed safe. As a result, proving the correctness of the smart contracts to be deployed on a blockchain is critical for the integrity of the specified business processes.

In our work we propose a formal approach based on the transformation of Solidity smart contracts, with consideration of the BPM context in which they are used, into a Hierarchical Coloured Petri net. We express a set of smart contract vulnerabilities as temporal logic formulae and use the *Helena* model checker to, not only detect such vulnerabilities while discerning their exploitability, but also check other temporal-based contract-specific properties.
Our proposed approach is based on model checking of CPN models and comprises mainly three phases:

1. transforming the smart contracts' Solidity code into CPN sub-models corresponding to their functions.

2. transforming the BPM context into a CPN model

3. constructing a CPN model w.r.t an LTL property that can express: *(i)* a vulnerability in the code or *(ii)* a contract-specific property, linking it to a CPN model representing the behavior to be considered, and feeding it the model checker to verify the targeted property.

More precisely, we opt for a hierarchical CPN to represent the considered smart contracts' execution and interaction w.r.t the provided BPM context specification. To do so, we represent each function of a smart contract by an *aggregated transition* that encapsulates a sub-model corresponding to the internal workflow of the former. In fact, our aim at this first step is to get building blocks for the hierarchical model that will be fed to the model checker. Then, given a context specification (transformed

into CPN) and an LTL property to be verified, the final CPN model is built by *(1)* linking the aggregated transition representing the targeted function to the behavioral model and *(2)* building a hierarchy by explicitly representing function calls in the submodel in question (if the checked property requires it).

We have implemented a graphical tool called *Solidity2CPN* that automates the different steps of the proposed approach and makes it accessible to a broader range of users who are unfamiliar with the aspects of formal verification.

Table of contents

List of Tables

List of Figures

Chapter 1

General Introduction

Contents

1.1 Research context

It is not often that an emerging technology acquires an impact that goes beyond what it had been initially intended, and yet as much can be said of the Blockchain. Initially featured as the technology behind Bitcoin [78], Blockchain has soon after escaped the box of cryptocurrencies to find its way into a multitude of application domains and become an integral ingredient of our daily lives. In fact, within the span of the two last decades, the Blockchain has known many advances that took it from being merely a database recording transactions between parties to being a computational platform on which small programs can be invoked as transactions. This leap has significantly expanded the power of Blockchain systems, and increased their reach to many application fields. This can be particularly observed in the growing interest Blockchains are gaining as part of IT systems, in domains such as health records, banking, voting, personal identity, etc [92]. According to healthcare statistics [13], Blockchain is in the top-5 priorities of 40% of senior health executives, mainly thanks to its ability to provide a seamless exchange of patient data between healthcare facilities while maintaining its confidentiality. Additionally, 13% of senior IT leaders expressed their intentions to integrate Blockchain into their daily business processes, driven essentially by its prospects to lower their system maintenance costs and raise their security levels [13]. This diversity in application fields, however, does not take away from the importance of cryptocurrencies as in the year 2020, the market size of the Blockchain technology was estimated at $3.7 billion [8]. The technological

research and consulting firm, Gartner, Inc., predicts that Blockchain's business value-add will be over $176 billion by 2025 and will approach $3.1 trillion by 2030. Such numbers are a clear indication of the weight of this technology in the Business world.

Currently, the Blockchain technology has witnessed the rise of three generations: the first one as a digital currency, the second as a digital economy, and the third as a digital society [44]. Conceptually, a Blockchain is a distributed ledger that keeps records of transactions happening among a network of participants. A consensus mechanism is used to confirm valid transactions and reject fraudulent ones. Once a transaction is confirmed and added to a record in the Blockchain, it is no longer possible to alter nor delete it. This definition gives the basic idea of a first-generation Blockchain which focuses solely on cryptocurrencies transfer. The emergence of second-generation Blockchains occurred when the distributed ledgers were embedded by so-called smart contracts that enable them to function as distributed computing platforms. Blockchain was then able to support new economic and financial applications beyond simple payments (e.g., traditional banking and legal services like loans, stocks, contracts, monetized assets, etc). The transition to third-generation Blockchains was only a matter of time as it became clear that what the concept of smart contracts brought to this technology could take its applications beyond any economic activity and into a variety of fields including art, health, science, identity, governance, etc [44]. As a matter of fact, the Blockchain technology is regarded as the second-most important innovation, behind the internet. If the latter could be used to establish online business processes by connecting people, the former resolves the trust problems using peer-to-peer networking and public-key cryptography.

One of the most interesting applications of the Blockchain technology is the domain of Business Process Management (BPM) [70]. The concept of Business processes (BPs) emerged in the early 20^{th} century and has been evolving ever since. By the early 1990s, researchers were already talking about Business Process Management (BPM) as a discipline that reigns upon enterprises seeking both effectiveness and efficiency and striving for innovation and flexibility especially within the world of IT. Many definitions have been proposed to clearly determine the meaning of this discipline and its boundaries. According to [30,35,100], BPM is a discipline that encompasses all types of modeling, automation, execution, control, measurement, and optimization of business activity flows in support of organizational goals, spanning systems, personnel, clients, and partners both inside and outside the enterprise. This discipline is therefore centered around these Business activity flows which are commonly referred to as Business Processes. A Blockchain platform can indeed provide a reliable execution of Business processes even within a trustless network, especially thanks to the concept of smart contracts. In fact, its inherent characteristics, namely its decentralized nature, ability to provide trust among trustless parties, immutability and financial transparency seem to put on the table the right tools to contrive adequate solutions for existing problems in the BPM research field, especially (but not exclusively) when it comes to collaborations [71]. One of the promising integration possibilities

of these two fields is the design of Blockchain-based business processes. So far, the general preference has been to use an existing modeling language for BPs and adopt Blockchain for different aspects of their management. For instance, Lorikeet [96] is a tool that leverages Blockchain as a message exchange mechanism for BP choreographies. Caterpillar [63], on the other hand, is used to implement the BP model and deploy it on the chain. This has been possible thanks to the concept of smart contracts introduced by Ethereum, which allow the execution of sequences of interdependent transactions while complying to the rules implemented within. In general, a BP can be analogously viewed as a sequence of tasks linked by causal relationships with the aim of achieving a particular business goal. Therefore, smart contracts seem to be ideal candidates for the implementation and automation of business processes. Moreover, the intended behaviour of smart contracts can intuitively be represented using a business process representation that would characterize the context in which the smart contracts are supposed to be executed. Such a behavioural context can either come directly as a description of a business model or be derived from a script that would be used to invoke the smart contracts. In fact, smart contracts are pieces of code that act like autonomous software agents, used to enforce management rules on the execution of transactions on the Blockchain. They are stored on and executed by the Blockchain and therefore inherit its characteristics, particularly its immutability. This same feature can, however, turn into a weak spot for such contracts. In fact, as a smart contract cannot be altered once it has been deployed on the Blockchain, it cannot be corrected either, which makes verifying its correctness prior to its deployment an indispensable necessity. Furthermore, the correctness verification is an important aspect for the design of Blockchain-based BPs. The assessment of such processes involves both requirements validation and consistency.

In light of this context, the ultimate goal of this thesis is to provide a solution that would aid smart contract designers, especially (but not exclusively) those involved in the BPM world, to correctly design and implement their contracts. This includes helping them avoid well-known vulnerabilities that may threaten the security of their smart contracts (and therefore their Blockchain-based applications), as well as providing the necessary support to define their own correctness criteria and make sure their smart contracts shall abide by them.

1.2 Motivation and problem statement

Blockchain is still considered an evolving technology whose extent has not been fully revealed. Working towards new ways of exploiting it, though can lead to valuable exploits, is undoubtedly a risky endeavor. In fact, despite the advance in the adoption of Blockchain for the BPM context, its state is still nascent, and using smart contracts to carry on BPs cannot be considered safe. Many attacks with significant consequences on several Blockchain platforms, exploiting hidden vulnerabilities in deployed smart contracts and exposing the defectiveness of the targeted applications bear witness to

such a risk. The first dangerous attack on a Blockchain can be traced back to August 2010, when 92 billion BTC were generated out of thin air by exploiting an integer overflow vulnerability in the Bitcoin Blockchain [9], which resulted in cancelling all relevant transactions and rolling back the Blockchain to a previous state. The DAO (a Decentralized Autonomous Organization built upon Ethereum) attack in June 2016, caused by a reentrancy vulnerability, is one of the most infamous attacks Ethereum has ever had to suffer [91]. On top of the tangible loss that evaluated to 3.6M of stolen ether (around 55M USD at the time) the attack resulted in a hard fork in the Ethereum Blockchain which could have easily resulted in a community fallout, the worst possible nightmare for Ethereum. The Parity multisig wallet has been subject to two substantial attacks. The first happened in July 2017 when more than 150K ETH were stolen (32M USD). The attacker used a vulnerability in the code (a bad practice) that allowed him to change the ownership of an important contract and take possession of its ether. The second attack which happened in November 2017, and which was more of an unintentional accident caused by a self-destruct vulnerability, did not result in stolen funds but caused 513K ETH to be locked in the attacked contracts (160M USD) [93]. The damages were obviously serious, so much as to push Blockchain foundations to propose bug bounty programs [3, 10] offering rewards for uncovering vulnerabilities.

From an academic point of view, numerous methods and tools have therefore emerged to both support the development of secure smart contracts and aid the analysis of already deployed ones. This panoply of studies comprises approaches that use non-formal techniques to detect bugs in certain execution scenarios, as well as approaches based on formal techniques and aim for an automatic formal verification of smart contracts. While informal techniques can test a certain requirement under certain scenarios, they cannot prove the correctness of a smart contract in general (e.g., absence of integer overflow vulnerabilities, deadlock-freedom). That's why researchers turned to formal verification which has proved to be efficient to reach such correctness goals [48].

We note that in our research work, we are interested in Ethereum smart contracts as it is currently the second largest cryptocurrency platform after Bitcoin besides being the inaugurator of smart contracts, and more particularly those written in Solidity [12] as it is the most popular language used by Ethereum. We also note that while Ethereum allows smart contracts to be written in a 'Turing complete' language that facilitates semantically richer applications than Bitcoin which allows very simple forms of smart contracts, the former also enlarges the threat surface, as evidenced by the many high-profile attacks.

As will be detailed later in Chapter 3, many studies have proposed formal approaches for the verification of Solidity smart contracts. The majority of these propositions, however, aim to detect some of the well-known vulnerabilities of Solidity, with only a minority giving the possibility to check for some other predefined properties and none allowing the specification of user-defined properties. Not only is the de-

tection of a number of vulnerabilities not enough to decide the correctness of smart contracts in general, but also the possibility of defining contract-specific properties can come very handy especially if the users have a specification of the behavioural context in which they intend to use their smart contracts. Moreover, no studies have yet treated the verification of smart contracts used in the context of BPM despite the growing use of Blockchain in this discipline.

The work presented in this PhD thesis is motivated by the previously described challenges and shortcomings, and aims to tackle the following research issue: *the formal verification of Solidity smart contracts in a BPM context.* To address this research problem, we need to answer the following main research questions:

- ***RQ1:*** How to verify Solidity smart contracts in general?
- ***RQ2:*** How to describe the behaviour of Solidity smart contracts using a BP context specification?
- ***RQ3:*** How to verify Solidity smart contracts with a BP context specification?

To better clarify the problems that we will be dealing with in this PhD work, we furthermore define other questions that derive from the previous ones:

- ***RQ4:*** How to homogenize a behavioural context specification with that of Solidity smart contracts
- ***RQ5:*** What vulnerabilities can hinder the correct execution of Solidity smart contracts and how to detect them?
- ***RQ6:*** What kind of properties need to be verified to ensure the correctness of Solidity smart contracts used in a BPM context?

1.3 Objectives and contributions

In view of the previously described challenges and shortcomings, the main objectives of this PhD thesis can be summarized as follows:

- ***Obj1:*** provide a formal model for Solidity smart contracts
- ***Obj2:*** provide formal models for possible behavioural contexts that may accompany the smart contracts
- ***Obj3:*** formally define well-known vulnerabilities
- ***Obj4:*** provide a full approach to formally verify smart contracts while supporting the possibility of behavioural context specifications

In this PhD thesis, we propose a formal approach for the verification of Solidity smart contracts with a particular focus on those used in the BPM context. Herein we give a global overview of this approach that combines our contributions that will be detailed in Chapters 4-7, and whose technical elements will be explained in Chapter 2.

Our approach is based on model checking as a formal verification method applied on Coloured Petri Net as a representation formalism and using Linear Temporal Logic to express the properties to be verified. In fact, model checking is technique whose goal is to check the satisfaction of some properties expressed in a formal logic in a system specified as some sort of finite-stat model. We choose to use this technique on Coloured Petri Nets (CPNs) [55]. Thanks to their ability to combine the analysis power of Petri nets with the expressive power of programming languages, CPNs are suitable candidates for the modeling and verification of large and complex systems, and therefore they are employed in our approach to model the smart contracts execution with respect to a behavioral context specification defining the workflow within which they are used. In layman's terms, we use the CPN formalism to represent Solidity smart contracts and behavioural context specifications, and we leverage the model checking method to see if properties defined on these contracts actually hold.

To explain our approach in more detail, we depict its different stages in Figure 1.1. This approach comprises mainly five steps:

1. transformation of the smart contracts' Solidity code into CPN sub-models corresponding to their functions. This step comes as a contribution to achieve ***Obj1*** which is part of the response to ***RQ1***.

2. transformation of the behavioural context specification into a CPN model. This step comes as a contribution to achieve ***Obj2*** which is part of the response to ***RQ2*** and ***RQ4***.

3. expression of the property to be verified in LTL. This step comes as a contribution to achieve ***Obj3*** which is part of the response to ***RQ5*** and ***RQ6***.

4. generation of a Hierarchichal CPN (HCPN) model. This step comes as a contribution to achieve ***Obj4*** which is part of the response to ***RQ3*** and ***RQ4***.

5. model checking of the generated HCPN model w.r.t to the specified LTL property. This is the final step that puts together all the pieces of the approach to achieve our final goal.

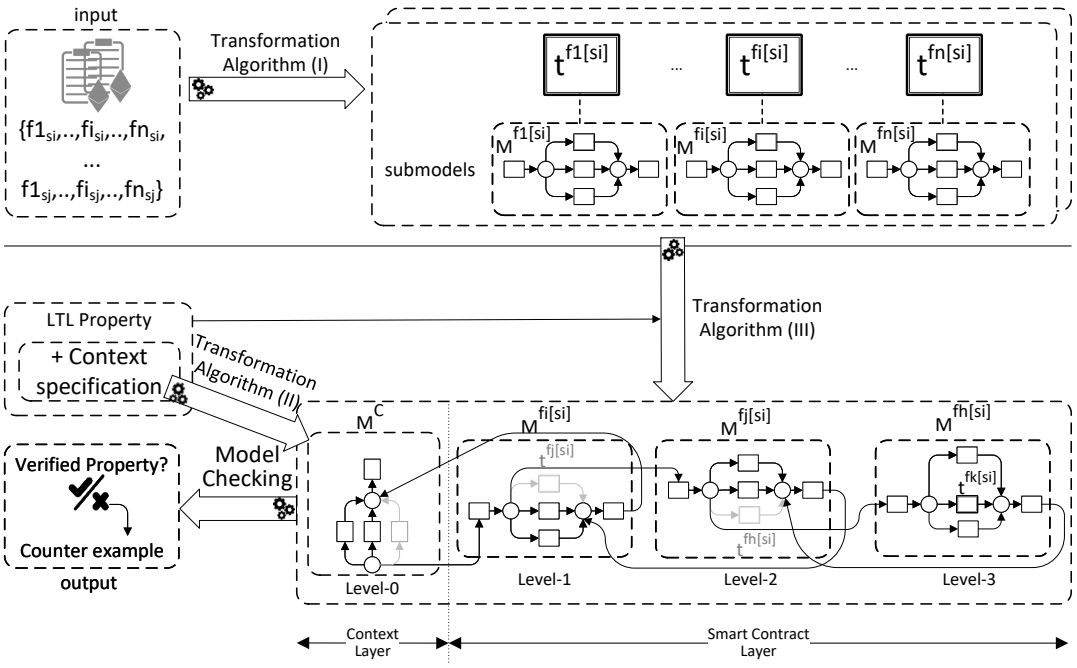

Figure 1.1: Overview of the approach

1.4 Publications

Conference Proceedings

1. Ikram Garfatta, Kaïs Klai, Mohamed Graïet, Walid Gaaloul, Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach, In the 37th ACM/SIGAPP Symposium on Applied Computing, SAC 2022: 316-325, Virtual Event, April 25 - 29.

2. Ikram Garfatta, Kaïs Klai, Walid Gaaloul, Mohamed Graïet, A Survey on Formal Verification for Solidity Smart Contracts, In the Australasian Computer Science Week Multiconference, ACSW 2021: 3:1-3:10, Dunedin, New Zealand, February 1-5.

3. Ikram Garfatta, Kaïs Klai, Mohamed Graïet, Walid Gaaloul, Model Checking of Solidity Smart Contracts Adopted for Business Processes, In the 19th International Conference on Service-Oriented Computing, ICSOC 2021: 116-132, Virtual Event, November 22-25.

4. Ikram Garfatta, Kaïs Klai, Mohamed Graïet, Walid Gaaloul, A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts, In the 30th IEEE

International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2021: 69-74, Bayonne, France, October 27-29.

5. Ikram Garfatta, Kaïs Klai, Mohamed Graïet, Walid Gaaloul, Blockchain-Based Business Processes: A Solidity-to-CPN Formal Verification Approach, In the 18th International Conference on Service-Oriented Computing, ICSOC Workshops 2020: 47-53, Dubai, United Arab Emirates, December 14-17.

1.5 Thesis outline

The remainder of this PhD thesis manuscript will be organized as follows:

- **Chapter 2: Preliminaries** introduces the basic concepts related to our research and needed to understand the details of the work. In this chapter, we first present the technology of Blockchain, with a focus on the Ethereum platform and its main smart contracts language Solidity. Then, we introduce both imperative and declarative approaches for the representation of Business Processes. Finally, we shift the focus onto the formal methods and models leveraged in our work, namely model checking as a formal verification technique, Coloured Petri Net as a representation formalism and Linear Temporal Logic as a means for the expression of properties to be verified.

- **Chapter 3: State of The Art** provides an exploration and a thorough analysis of the state of the art around the problematic of our research work. Herein we mainly focus on studies related to the formal verification of Solidity smart contracts. We then give an overview of the application of formal methods for the verification of different aspects of Business process models.

- **Chapter 4: Formal Modeling of Solidity Smart Contracts** presents our approach to formally model Solidity smart contracts using Coloured Petri Net as a representation formalism.

- **Chapter 5: Formal Modeling of Behavioural Contexts** introduces our approach to formally represent two kinds of contexts that can be used for the description of the behaviour of smart contracts. We start by the formalization of the free context which is used when no restrictions are provided for the execution of such contracts. Then we give a formalization for a constrained context which is used when some restrictions are provided for the execution of the smart contracts.

- **Chapter 6: Expression of Properties to be Verified using Linear Temporal Logic** details our approach to formalize six of the most common vulnerabilities in Solidity using LTL and explains how this logic can also be used for the specification of contract-specific properties.

- **Chapter 7: Application and Implementation of our Approach** illustrates how the contributions presented in the three previous chapters are put together to bring forth and end-to-end tool for the formal verification of smart contracts with a behavioural context specification.
- **Chapter 8: Conclusion and Future Work** summarizes the proposed contributions and presents an outlook on the potential perspectives that we intend to tackle in the short-medium term.

Chapter 2

Preliminaries

Contents

2.1 Introduction

This chapter introduces the main concepts and background required for the understanding of the contributions described in the remainder of this manuscript. We start by introducing the Blockchain technology in Section 2.2, with a focus on the Ethereum platform and Solidity as a smart contracts implementation language. Then, in Section 2.3 we discuss both the imperative and declarative paradigms for Business process modeling while focusing on Dynamic Condition Response (DCR) Graphs as a representation for BPs. Finally, Section 2.4 comes to introduce the formal methods and models that will be used throughout our proposed contributions, namely model checking, Coloured Petri Net and Linear Temporal Logic.

2.2 Blockchain Technology

The idea for the first decentralized Blockchain goes back to 2008 when Satoshi Nakamoto proposed a design for a peer-to-peer electronic cash network that would solve the problem of double-spending without the need for a trusted third-party. This network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. This idea was then put into use as fundamental part of the Bitcoin cryptocurrency where it serves as the network's main public ledger for all transactions [78]. Numerous applications have emerged since the advent of Bitcoin, all of which aim to make use of the capabilities and principles of the public ledger technology. As a result, a considerable number of Blockchain platforms have emerged over the course of history.

2.2.1 The Ethereum Platform

Co-founder of Bitcoin Magazine in 2011, Vitalik Buterin was among the many engineers who believed that Bitcoin had not yet fully used the potential of the Blockchain technology. He began developing what he thought would be a flexible Blockchain that could serve a variety of purposes in addition to serving as a peer-to-peer network. A crucial turning point in the history of the Blockchain came when Ethereum [2] was introduced as a brand-new public Blockchain in 2013 with more features than Bitcoin. Buterin was able to position Ethereum in a different league from its predecessor Bitcoin by introducing two main additions: the possibility to keep track of assets other than cryptocurrencies, and the renowned concept of smart contracts. These new features were what allowed Ethereum's capabilities to go beyond those of a cryptocurrency to those of a platform for the development of decentralised applications. The Ethereum Blockchain, which was formally introduced in 2015, ended up being one of the most significant uses of Blockchain technology thanks to its smart contracts that can be used for a variety of tasks. The Blockchain technology for Ethereum has also been successful in attracting a vibrant developer community, which has helped it build a real ecosystem. The Ethereum Blockchain conducts the most daily transactions and its market cap has also notably grown [1].

2.2.2 Solidity Smart Contracts

Smart contracts take the famous saying "code is law" into a new perspective where law becomes code. They can be seen as the equivalent of contracts written on paper, where the agreed upon terms are transcribed in lines of code. The faithful execution of a smart contract is governed and guaranteed by the laws of the Ethereum Virtual Machine (EVM) semantics and its immutable nature gives it a sense of finality.
The most commonly-used high-level programming language for Ethereum smart contracts is Solidity [14]. A Solidity smart contract is a collection of code and data,

residing at a specific address on the Ethereum Blockchain, which can be invoked either by an internal account (i.e., a smart contract) or directly by an external account (i.e., user). Every account is characterized by a persistent *storage* (null in case of an external account) and a balance in Ether which is adjusted by transactions. A transaction is message used to send ether from one account to another and/or invoke a smart contract's function if the message includes a payload and the targeted account is an internal one. The execution of such a payload is carried out according to a *stack* machine called the EVM. Every smart contract features a *memory* which is cleared at each message call, and it can access certain properties of the current block (e.g., number, timestamp...). Besides the storage, stack and memory, Ethereum has an externally accessible indexed data structure that can be used by Solidity to implement events and acts as a *log*.

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but they are actually dissimilar since the underlying semantics of Solidity functions differently from traditional programs. This naturally calls on more vigilance from the programmers who might be faced by unconventional security issues as vulnerabilities in smart contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer [24].

Solidity Smart Contract Example - Blind Auction

We consider the following example which we adapted from [12].

Participants in a blind auction have a bidding window during which they can place their bids. A participant can place more than one bid (function *bid* in Listing 2.1 - Lines 17 to 19) and the placed bid is blinded. The bidder has to make a deposit with the blinded bid, with a value that is supposedly greater than the real bid. Once the bidding window is closed, the revealing window is opened. Participants proceed to reveal their bids (function *reveal* in Listing 2.1 - Lines 20 to 30) by sending the actual values of the bids along with the used keys. The system verifies whether the sent values correspond with the placed blinded bids and potentially updates the highest bid and bidder's values. If the revealed value of a bid does not correspond with its blinded value, or is greater than the deposit, the said bid is considered invalid. Once the revealing window is closed, participants can proceed to withdraw their deposits (function *withdraw* in Listing 2.1 - Lines 31 to 42). A deposit made along a non-winning, invalid or unrevealed bid is wholly restored. In case of a winning bid, the difference between the deposit and the real bid is restored. The auction is terminated when all participants withdraw their deposits.

```
1 contract BlindAuction {
2     struct Bid {
3         bytes32 blindedBid;
4         uint deposit;}
5     uint public biddingEnd;
6     uint public revealEnd;
```

```
7     mapping(address => Bid[]) public bids;
8     address public highestBidder;
9     uint public highestBid;
10    mapping(address => uint) pendingReturns;
11    modifier onlyBefore(uint _time) {require(now<_time);_;}
12    modifier onlyAfter(uint _time) {require(now>_time);_;}
13    constructor(uint _biddingTime, uint _revealTime) public {
14        biddingEnd = now + _biddingTime;
15        revealEnd = biddingEnd + _revealTime;
16    }
17    function bid(bytes32 _blindedBid) public payable onlyBefore(
      biddingEnd) {
18        bids[msg.sender].push(Bid({blindedBid: _blindedBid, deposit:
      msg.value}));
19    }
20    function reveal(uint[] values, bytes32[] secrets) public onlyAfter(
      biddingEnd) onlyBefore(revealEnd) {
21        require (values.length == secrets.length);
22        for(uint i = 0; i < values.length && i < bids[msg.sender].
      length; i ++) {
23            var bid = bids[msg.sender][i];
24            var (value, secret) = (values[i], secrets[i]);
25            if(bid.blindedBid == keccak256(value, secret) && bid.
      deposit >= value && value > highestBid) {
26                highestBid = value;
27                highestBidder = msg.sender;
28            }
29        }
30    }
31    function withdraw() public onlyAfter
32        (revealEnd) {
33        uint amount = pendingReturns[msg.sender];
34        if (amount > 0) {
35            if (msg.sender != highestBidder)
36                msg.sender.transfer(amount);
37            else
38                msg.sender.transfer(amount - highestBid);
39            pendingReturns [msg.sender] = 0;}}}stBid) ("");
40            pendingReturns [msg.sender] = 0;
41        }
42    }
43 }
```

Listing 2.1: The Blind Auction smart contract in Solidity

As we have mentioned before in Section 1.1, smart contracts bear some resemblance to Business processes, which stems from the fact that both can be regarded as a set of activities or operations, executed in a certain logical order to achieve a certain functional goal. This, in addition to the compatibility of the Blockchain's characteristics (e.g., immutability, decentralization, etc) with the needs in BPM, makes the integration of these two fields a propitious idea. The next section of this manuscript

will introduce the basics on Business Process representations, these will later on be used as means to the modeling of behavioural contexts for smart contracts.

2.3 Business Process Model Representations

The representation of Business process models follows mainly two paradigms: an *imperative* one which is considered the traditional way to represent Business process models, and a *declarative* one that has been witnessing a widespread use recently [46].

Controversy arises as to whether imperative or declarative modeling approaches are better. An empirical investigation [84] states that while imperative languages can be considered superior in terms of comprehensibility by end-users, this fact's accuracy can be influenced by the experimental subjects' familiarity with imperative modeling languages as they are traditionally and more profusely used in business process modeling. On the other hand, declarative modeling approaches are loosely considered less rigid than their counterpart and therefore more suitable for rapidly evolving business processes. This difference in flexibility emanates from the fact that imperative models represent *how* a process is executed by explicitly defining its control flow while declarative models focus on *why* a process is executed in such a way by implicitly defining its control flow as a set of rules. Consequently, making changes to an imperative model is more time-consuming and complex than altering a declarative one, since the former would entail explicitly adding/deleting execution alternatives (which can call into question the correctness of the model) while the latter could be achieved by adding/deleting constraints from the model to discard/add execution alternatives.

In our work, we do not support any claims for the supposed superiority of any paradigm over the other, but we rather aim to exploit the best in both of them.

2.3.1 Imperative Representations

Imperative process models are also referred to as workflow models since they are a representation of the control flow of the process. A workflow is a description of an algorithm used to carry out a business process. As a result, it necessitates that the modeler be fully aware of every step required to achieve the intended outcomes in each and every scenario. A wide range of graphical process modelling languages has been proposed over the last decades to represent a business process such as Business Process Model and Notation (BPMN) [80], Event-Driven Process Chain (EPC) [90], Yet Another Workflow Language (YAWL) [17], UML Activity Diagram [61], etc. Despite their variances in expressiveness and modelling notations, they all share the common concepts of tasks, events, gateways, artifacts and resources, as well as relations between them, such as transition flows [101].

Business Process Model and Notation

The Business Process Model and Notation (formerly known as Business Process Modelling Notation) (BPMN) was first released in 2004 by the Business Process Management Initiative (BPMI) [80]. BPMN is a standard for business process modelling that allows to create and document process models. It is considered as the *de facto* process modelling notation that is widely used in industry. In its latest versions, BPMN has been enhanced with executable semantics enabling the execution of the modelled process. It provides a rich set of elements to capture different perspectives of the business process at different levels of detail. These elements can be categorized into a *core set* which contains the basic elements to model a business process and an *extended set* which contains more specialized elements to specify more complex business scenarios [75]. Overall, BPMN defines 50 constructs grouped into four categories: *Flow objects*, *Connecting objects*, *Swimlanes* and *Artifacts*. *Flow Objects* allow to model the control flow perspective of a business process in terms of activities, events and gateways. An activity is the main element of a process model and describes the kind of work that must be done. It is graphically represented as a rectangle. An event is something that happens during the execution of a business process. There are three main types of events: Start, Intermediate and End events. An event is graphically represented with a circle. A gateway allows to model the splits and joins in the process model. Three main types are used to represent different behaviours in a business process: AND (parallel forking and synchronization), XOR (exclusive choice and merging) and OR (inclusive choice and merging). Although there exist other more specialized gateways in BPMN such as event-based gateway and complex gateways, they can all be mapped to one of the three main types OR, AND or XOR. The *flow objects* elements are connected through the *Sequence flow* element in the *Connecting objects* category. They determine the order in which the activities are supposed to be performed in a process.

In Figure 2.1, we propose a design for the blind auction system whose smart contract was described in Section 2.2.2 (Listing 2.1) using a BPMN choreography diagram. In this example, we chose to separate the tasks performed by the *Bidder* from those performed by the *Auction Holder*, who in our case is the smart contract itself, hence the use of two swimlanes. The tasks of the *Bidder* represent therefore calls to the functions of the Blind Auction smart contract (e.g., the activity *Place blinded bid* represents a bidder's call to the function *bid* in Listing 2.1), whereas the tasks of the *Auction Holder* represent the execution of these functions (e.g., the activity *Receive blinded bid* represents the execution of the function *bid* in Listing 2.1). *Start Reveal* and *End Reveal* are two intermediate events used to indicate the opening and closing of the revealing window as per the description given for the Blind Auction smart contract. The *XOR* split and join gateways (lozenge shapes with X inside) are used to allow a bidder to make the exclusive choice to either reveal and then withdraw their bid, or directly withdraw it.

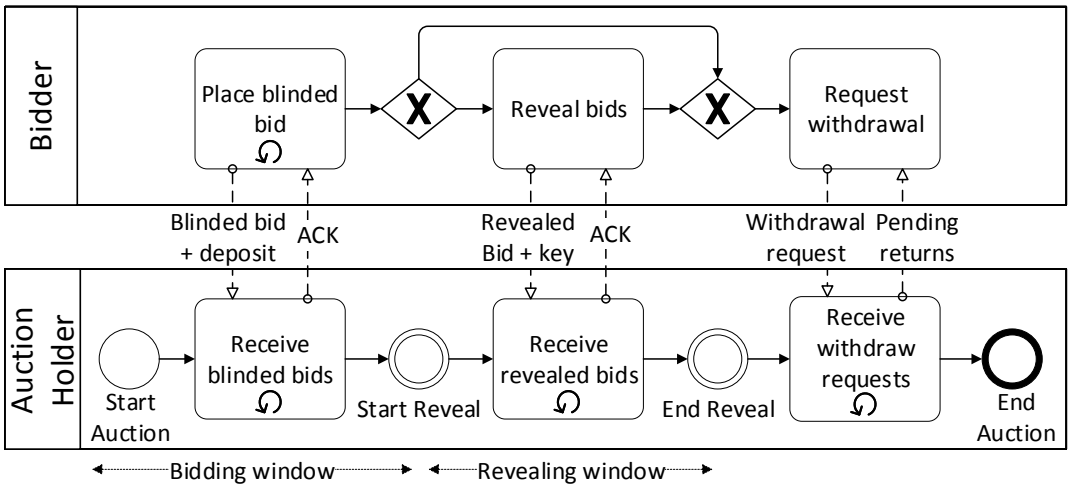

Figure 2.1: Blind Auction Workflow - BPMN Choreography

2.3.2 Declarative Representations

Imperative modeling languages for Business processes are viewed as being somewhat strict and restrictive since they do not offer strategies for dealing with unanticipated circumstances. New adaptive business processes have then been developed in response to the need for more adaptable management systems [36]. This marked a paradigm change compared to the traditionally-used modeling techniques. Declarative languages are distinguished by their ability to express "what" has to be done without determining the specifics of "how" to do so. They therefore enable context-aware decision-making by process participants during the execution of business processes. Contrarily, restricting business rules are established to stop members from engaging in actions that the organization forbids or deems undesirable [82]. ConDec [82] and DecSerFlow [18] are amongst the first declarative languages proposed for process modeling and are both supported by DECLARE [83], a prototype workflow management system. In our work, we are interested in Dynamic Condition Response (DCR) graphs [50] as a means for business process representation as this language was proposed to improve on the execution efficiency problems encountered in the former languages [19].

In the following, we start by the definition of DCR graphs before giving the definition of DCR choreographies.

Dynamic Condition Response Graphs

Syntax

Definition 2.3.1. *A DCR graph is a tuple* $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ *where* $\mathcal{M}(G) =_{def} \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$ *is the set of all markings:*

1. *E is the set of events, ranged over by e.*
2. *$M \in \mathcal{M}(G)$ is the marking of the graph (explained below).*
3. *Act is the set of actions.*
4. *$\rightarrow\bullet, \bullet\rightarrow \subseteq E \times E$ are the condition and response relations, respectively. In general, e is a condition for e' ($e \rightarrow\bullet e'$) means that e must have been executed at least once before executing e'. Having e' as a response for e ($e\bullet\rightarrow e'$) means that e' should be executed at least once after having executed e.*
5. *$\rightarrow+, \rightarrow\% \subseteq E \times E$ are the dynamic include and exclude relations, respectively, satisfying that $\forall e \in E \,.\, e \rightarrow+ \cap\, e \rightarrow\% = \emptyset$.*
6. *$\rightarrow\diamond \subset E \times E$ is the milestone relation.*
7. *$l : E \rightarrow Act$ is a labelling function mapping every event to an action.*

Semantics A marking $M = (Ex, Re, In) \in \mathcal{M}(G)$ is a triplet of event sets where Ex represents the set of events that have previously been executed, Re the set of events that are pending responses required to be executed or excluded, and In the set of events that are currently included. The idea conveyed by the dynamic inclusion/exclusion relations is that only the currently included events are considered in evaluating the constraints. In other words, if e is a condition for e' ($e \rightarrow\bullet e'$), but is excluded from the graph then it no longer restricts the execution of e'. Moreover, if e' is the response for e ($e\bullet\rightarrow e'$) but is excluded from the graph, then it is no longer required to happen for the flow to be acceptable. The inclusion relation $e \rightarrow+ e'$ (resp. exclusion relation $e \rightarrow\% e'$) means that, whenever e is executed, e' becomes included in (resp. excluded from) the graph. The milestone relation is similar to the condition relation in that it is a blocking one. The difference is that it is based on the events in the pending response set. In other words, if e' is a milestone of e ($e' \rightarrow\diamond e$), then e cannot be executed as long as e' is in Re.

In Figure 2.2, we propose a design for the blind auction system whose smart contract was described in Section 2.2.2 (Listing 2.1) using a DCR graph. Visually, such a model can be represented as a directed graph with events (boxes) as nodes and five types of arrows for the five types of relations that can link them. This DCR graph follows the same idea as the BPMN choreography model proposed in Figure 2.1. That is to say, an event in our graph can either represent a call to a function from the Blind Auction smart contract (Listing 2.1) or the execution of that function itself. Relations are used to define which events can or have to be executed at each step of the process. For example, initially only the *Place blinded bid* event can be executed (the initial marking M_0 indicates that we only have that event in the set of included events In). The execution of the said event would include the *Receive blinded bids* and *Reveal bids* events which have the former event as a condition. The *Receive blinded bids* event is

also a milestone for the *Reveal bids* event, which means that at this step, only the *Receive blinded bids* event can be executed. In this proposed graph, the execution of the *Reveal bids* event marks the start of the revealing window by excluding the *Place blinded bid* event, and the execution of the *Request withdrawal* marks the end of that window by excluding the *Reveal bids event*.

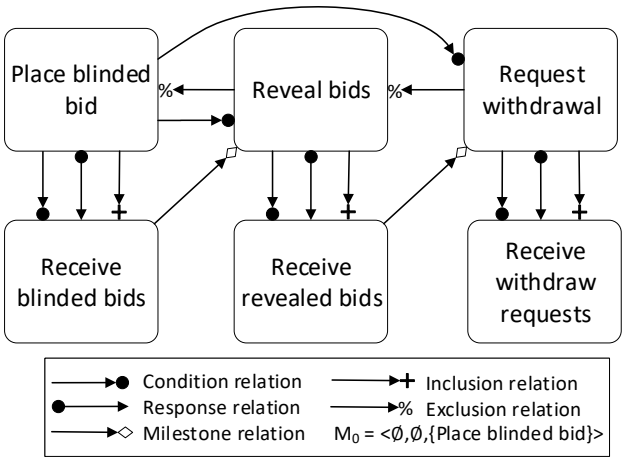

Figure 2.2: Blind Auction Workflow - DCR Graph

Dynamic Condition Response Choreographies

Syntax A DCR choreography [51] is a DCR graph that can be executed in a distributed way between a number of participants. An event in a DCR choreography has an *initiator* and can potentially have one or more *receivers*. In the following, we adapt the definition given in [51] on account of simplicity and better adequacy with our work.

Definition 2.3.2. *A DCR choreography is a couple* (G, R) *where* R *is a set of roles and* G *is DCR graph whose labelling function* l *is instead defined as follows:*

- $l : E \rightarrow (Act \times R \times \mathcal{P}(R))$

Semantics A DCR choreography has the same semantics as a DCR graph with the added condition that only the *initiator* of an event can execute it. For more details on DCR Graphs we refer the readers to [76].

An example of a distributed DCR choreography for our Blind Auction system is given in figure 2.3. As a continuation to the previous DCR graph in Figure 2.2 and to keep the harmony with the BPMN choreography in Figure 2.1, we keep the events and relations of the former and introduce the participants of the latter (i.e.,*Bidder*

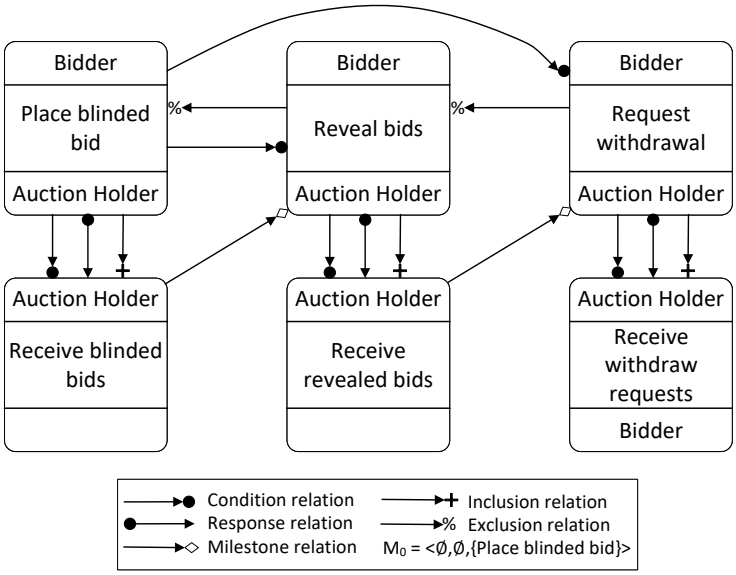

Figure 2.3: Blind Auction Workflow - DCR choreography

and the *Auction Holder*) . We use the upper part of a box to indicate the *initiator* of the event and the bottom part to indicate the *receiver(s)*.

A primordial phase in the life cycle of a business process is its verification. In order to avoid execution mishaps whose gravity may vary with the level of criticality of the modeled system, the designer needs to make sure that the model will actually satisfy a set of requirements/specifications that define the correctness of the system. In the following section, we will present a basic introduction to formal methods and models, with a focus on model checking as a verification technique and coloured Petri net as a formal model.

2.4 Formal Methods and Models

Formal methods are methodologies to specify, develop, and verify software and hardware systems that are rigorously mathematical in nature. The idea that completing adequate mathematical analysis may, like in other engineering disciplines, contribute to the dependability and robustness of a design is what drives the adoption of formal techniques [53]. When it comes to formal verification, such techniques are used to check the conformance of the developed system to a predefined specification. They are based on formal methods of mathematics and are able to provide formal proof of the correctness of the investigated system with reference to its formally specified behavior. We can distinguish two main branches of formal verification methods: first, those based on theorem proving, where a prover is used to discharge proofs on a math-

ematically modeled system. Such approaches however cannot be fully automated, as the user usually has to intervene to assist the prover; and second, those based on model checking. Besides being automatable, model checking techniques can also provide counter examples which are basically execution traces that show how a property of the modeled system is violated. This can be a valuable information for designers especially in reassessing their model. The basic idea of model checking is to model the system as a formal model (e.g., finite state machine), generate its state space, and then explore it to check some specification that is supposed to define the correctness of the system.

In this PhD thesis, we rely on model checking to verify Linear Temporal Logic properties on coloured Petri net models.

2.4.1 Coloured Petri Nets

A Petri net [77] is a formal model with mathematics-based execution semantics. It is a directed bipartite graph with two types of nodes: places (drawn as circles) and transitions (drawn as rectangles). Despite its efficiency in modelling and analysing systems, a basic Petri net falls short when the system is too complex, especially when representation of data is required. To overcome such limitations, extensions to basic Petri nets were proposed, equipping the tokens with colours or types and hence allowing them to hold values. A large Petri net model can therefore be represented in a much more compact and manageable manner using a *Coloured Petri net* [55]. The formal definition of a CPN is given in Definition 2.4.1.

2.4.1.1 Syntax

Definition 2.4.1 (Coloured Petri net)**.** *A* Coloured Petri Net *is a nine-tuple* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*, where:*

1. P *is a finite set of* places*.*
2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*.*
3. $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed* arcs*.*
4. Σ *is a finite set of non-empty* colour sets*.*
5. V *is a finite set of* typed variables *such that* $Type[v] \in \Sigma$*,* $\forall v \in V$*.*
6. $C : P \to \Sigma$ *is a* colour set function *that assigns a colour set to each place.*
7. $G : T \to EXPR_V$*, where* $EXPR_V$ *is the set of expressions with variables in* V*, is a* guard function *that assigns a guard to each transition* t*.*
8. $E : A \to EXPR_V$ *is an* arc expression function *that assigns an arc expression to each arc* a *such that* $Type[E(a)] = C(p)_{MS}$*.*

9. $I : P \rightarrow EXPR_{\emptyset}$ *is an* initialisation function *that assigns an initialisation expression to each place* p *such that* $Type[I(p)] = C(p)_{MS}$.

2.4.1.2 Semantics

For CPN $(P, T, A, \Sigma, V, C, G, E, I)$, we note:

- A *marking* is a function M that maps each place into a multiset of tokens.
- The *initial marking* M_0 is defined by $M_0(p) = I(p)\langle\rangle$ for all $p \in P$.
- The *variables of a transition* t are denoted by $Var(t) \subseteq V$.
- A *binding* of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type[v]$. It is written as $\langle var_1 = val_1, ..., var_n = val_n\rangle$. The set of all bindings for a transition t is denoted $B(t)$.
- A *binding element* is a pair (t, b) such that $t \in T$ and $b \in B(t)$. The set of all binding elements $BE(t)$ for a transition t is defined by $BE(t) = \{(t, b)|b \in B(t)\}$. The set of all binding elements in a CPN model is denoted BE.

A transition is said to be *enabled* if a binding of the variables appearing in the surrounding arc inscriptions exists such that the inscription on each input arc evaluates to a multiset of token colours present on the corresponding input place. *Firing* a transition consists in removing (resp. adding), from each input (resp. to each output) place, the multiset of tokens corresponding to the input (resp. output) arc inscription. The semantics of a CPN can be represented by a graph (*reachability graph*) where the initial state is the initial marking and an arc, labeled with a transition t, connects a marking m to a marking m' when t is firable from m and its firing leads to m'. For more details on CPN we refer readers to [55].

CPN Example

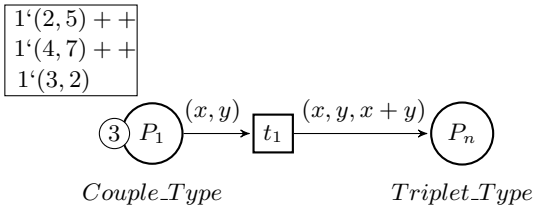

Figure 2.4: A simple example of a CPN model

To better explain the basic concepts of CPN, we use the simple CPN model of Fig. 2.4. *Couple_Type* is defined as the product of two integers and *Triplet_Type* as the product of three integers. x and y are two integer variables. In a CPN model, each

place has a colour that determines the kind of data it can contain. We say that $p1$ is of colour (or type) $Couple\_Type$ and $p2$ is of colour $Triplet\_Type$. Initially, the place $p1$ contains three tokens with different values (three different couples). The expressions on the arcs have to correspond to the colours of their respective places (e.g., the expression on the outgoing arc of $p1$ has to conform to its colour $Couple\_Type$). In this CPN, (x, y) can be bound to any of the tokens in $p1$. For example, if it is bound to the first token $(2, 5)$, the firing of transition $t1$ results in consuming that token from $p1$ and producing a token with the value $(2, 5, 7)$ in $p2$.

Figure 2.5 shows an example of a CPN model that could represent the previously described blind auction system (Listing 2.1). Evidently, the transitions *bid*, *reveal* and *withdraw* (in green) represent the execution of their namesake Blind Auction smart contract functions. The transitions in the user's behaviour box (i.e., *bid Call*, *bid ACK*, etc) are the equivalent of the activities/events that represent the smart contract's functions invocation by the bidder in the previous BPMN and DCR examples.

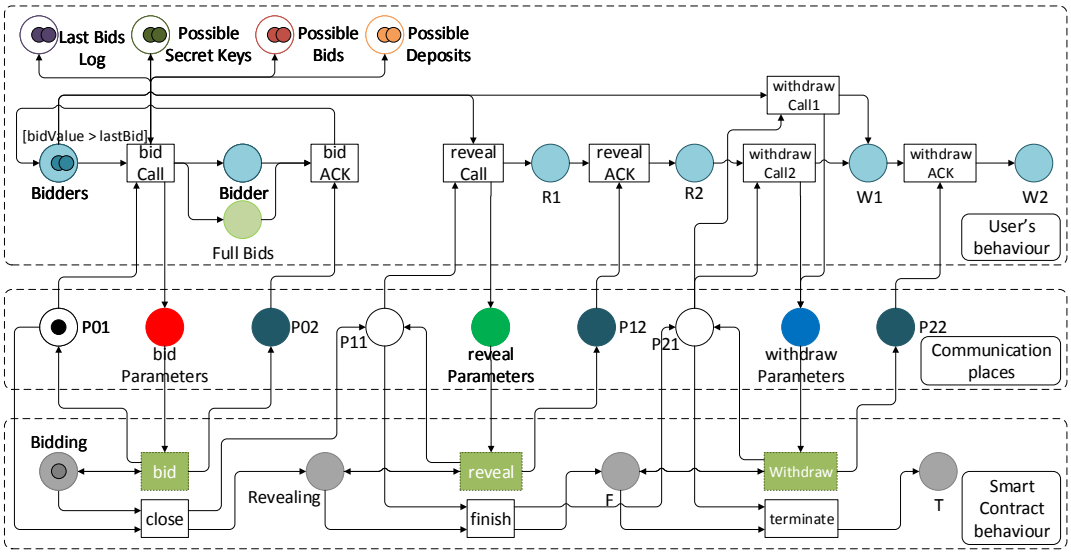

Figure 2.5: Blind Auction - CPN Model

2.4.2 Linear Temporal Logic

The approach presented in this PhD work is primarily based on model checking of CPN models w.r.t formulae expressed in Linear Temporal Logic (LTL). This logic was first introduced in [85] as a means to reason about concurrent programs. In LTL, a classical timeline that starts "now" is considered, where every moment has a unique possible future. In other words, a model of LTL is an infinite sequence of indexed states ($i = 0, 1, 2, ...$) where each point in time has a unique successor. An

LTL formula is evaluated over such a sequence of states starting from an i'th state. It contains a finite set *Prop* of atomic propositions, the usual Boolean operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$, in addition to temporal operators:

- Until ($\mathcal{U}$): $\varphi \,\mathcal{U}\, \psi$ is true if ψ is true *now* or φ is true *now* and remains so until ψ holds.
- Next ($\mathcal{X}$ or $\bigcirc$): $\mathcal{X}\, \varphi$ is true if φ is true in the next step.
- Globally ($\mathcal{G}$ or $\Box$): $\mathcal{G}\varphi$ is true if φ is true in every step.
- Future ($\mathcal{F}$ or $\Diamond$): $\mathcal{F}\, \varphi$ is true if φ is true now or in some future time step.

Definition 2.4.2 (LTL formula)**.** *An LTL formula can be inductively defined as follows:*

- $\forall p \in Prop$, *p is an LTL formula.*
- *If φ and ψ are two LTL formulae, then $\neg\,\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\varphi \,\mathcal{U}\, \psi$, $\mathcal{X}\, \varphi$, $\mathcal{G}\, \varphi$ and $\mathcal{F}\, \varphi$ are LTL formulae too.*

The technique of model checking checks that a system, starting at a start state, satisfies a specification [89].

An atomic proposition could be a state or event-based basic property. In this work, we consider hybrid linear-time temporal logic (hybrid LTL) formulae where both state- and event-based atomic propositions can occur. Therefore, we chose to represent the semantics (behavior) of a system by a *Labeled Kripke Structure* (*LKS*).

Definition 2.4.3 (Labeled Kripke Structure (LKS))**.** *Let* AP *be a finite set of atomic propositions and* Act *be a set of actions. An* LKS *over* AP *is a* 5*-tuple* $\langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ *where:*

- Γ *is a finite set of* states*,*
- $L : \Gamma \rightarrow 2^{AP}$ *is a* labeling (or interpretation) function*,*
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a* transition relation*, and*
- $s_0 \in \Gamma$ *is the* initial state*.*

Given a CPN $\mathcal{N}$ and an initial marking m_0 the reachability graph of $\mathcal{N}$ could be considered as a LKS where the states are labeled with atomic propositions related to the places and where the set of actions Act is the set of transitions.

Definition 2.4.4 (Hybrid LTL)**.** *Given a set of atomic propositions* AP *and a set of actions* Act*, a* hybrid LTL *formula is defined inductively as follows:*

- *each member of $AP \cup Act$ is a formula,*

- *if ϕ and ψ are hybrid LTL formulae, so are $\neg\phi$, $\phi \vee \psi$, $X\phi$ and $\phi U \psi$.*

Other temporal operators, e.g. F (futur) and G (globally) can be derived as follows: $F\phi = true \cup \phi$ and $G\phi = \neg F \neg \phi$.

An interpretation of a hybrid LTL formula is an infinite run $w = s_0 s_1 s_2 \dots$ (of some *LKS*), assigning to each state s_i a set of atomic propositions and a set of actions that are satisfied within that state. A $p \in AP$ is satisfied by a state s_i if it belongs to its label (i.e. $L(s_i)$), while an action $a \in Act$ is said to be satisfied within a state s_i if it occurs from this state in w (i.e. $(s_i, a, s_{i+1}) \in \rightarrow$). In our case, where a single action can occur at a time (i.e. interleaving model of concurrency), at most one action can be assigned to a state of a run.

We write w^i for the suffix of w starting from s_i. Moreover, we say that $p \in s_i$, for $p \in AP \cup Act$, when p is satisfied by s_i. The hybrid LTL semantics is then defined inductively as follows:

- $w \models p$ iff $p \in s_0$, for $p \in AP \cup Act$,
- $w \models \phi \vee \psi$ iff $w \models \phi$ or $w \models \psi$,
- $w \models \neg\phi$ iff not $w \models \phi$,
- $w \models X\phi$ iff $w^1 \models \phi$, and
- $w \models \phi U \psi$ iff $\exists i \geq 0$ such that $w^i \models \psi$ and $\forall\, 0 \leq j < i$, $w^j \models \phi$.

An *LKS* K satisfies a hybrid LTL formula φ ($K \models \varphi$), iff all its runs do.

2.4.3 Model Checking

The goal of model checking is to verify that properties (specified in a temporal logic) are satisfied w.r.t a system (represented as a finite-state model). The general idea here is to construct the state space of the model, and to explore it in order to check a specification that is supposed to define the correctness of the system, and potentially generate counterexamples in case the specification was not met. The *standard* approach to do that would be to generate all the reachable states of the system, represent them individually and then exhaustively explore the state space to check for the specified property. The application of such a method would face a state space explosion problem in case of complex systems which constrains its application. That's why other model checking approaches appeared.

BDD-based symbolic model checking (e.g., SMV [68]) presents a different way to store the states of the system, grouping them into sets of states represented by predicates on its state variables in the form of BDDs (Binary Decision Diagram). Such an approach reduces the size of the state space to be explored, making for a more efficient exhaustive exploration, yet it limits the nature of the variables that can be manipulated. *Bounded model checking* (e.g., SAT [27], SMT [94]) is another form of symbolic

model checking that does not rely on a symbolic representation of the states of the system, but rather on applying decision procedures on prepositional logic. Such an approach turns the verification problem into a satisfiability problem. The goal here is to check if there exists values that can be assigned to the variables in the formula to be verified, so as it evaluates to false, within a certain number of exploration steps. While this approach overcomes the state space explosion problem, it cannot be considered complete since variable assignations under which the evaluation is false could exist beyond the considered search depth.

Complementary Techniques Symbolic model checking is often seen allied to other techniques in order to improve its efficiency or widen its application range.
Abstractions (e.g., [22]) can be used with symbolic model checking to deal with state space explosion in software analysis. An abstraction can be either sound, in which case properties of the abstract specification are also properties of the original one, or complete, in which case properties of the original specification are properties of the abstract one. While a sound (resp. complete) abstraction guarantees false positive-free (resp. false negative-free) results it cannot guarantee the absence of false negatives (resp. false positives).
Symbolic execution (e.g., [59]) can be placed as the crossover between a formal verification technique and a testing technique for programs. Its underlying idea is to represent input variables using symbols over which the program is symbolically executed instead of assigning concrete values, which yields symbolic formulae instead of concrete results. Hence, one result of the symbolic execution encompasses a set of test cases. In such a context, SMT solvers are often used to check for the reachability of some part of the code, which amounts to checking the satisfiability of the conjunction of the formulae encountered on its corresponding path.

In this PhD work, we will be using the *Helena* model checker [45] which is based on the automata-theoretic approach [98] for *explicit* model checking of LTL properties. As described in Figure 2.6, this approach starts by translating the LTL property φ to be verified into a Büchi automaton of its negation $A_{\neg\varphi}$, and generation of the state space automaton A_M of the input model M on the fly. The synchronized product of the two automata is then computed and used for the an emptiness check. This test indicates whether the synchronized product accepts an infinite word, in which case this word is returned as a counter example. We say that φ is satisfied under M if and only if the language of the synchronization product $\mathcal{L}(A_{\neg\varphi} \otimes A_M)$ is empty.

2.5 Conclusion

This chapter provided the background that helps position our contributions in the three main involved research fields. In fact, it introduced Solidity, the smart contracts language of Ethereum in which we are interested in this PhD work. Then, it presented different existing representation approaches for Business process models

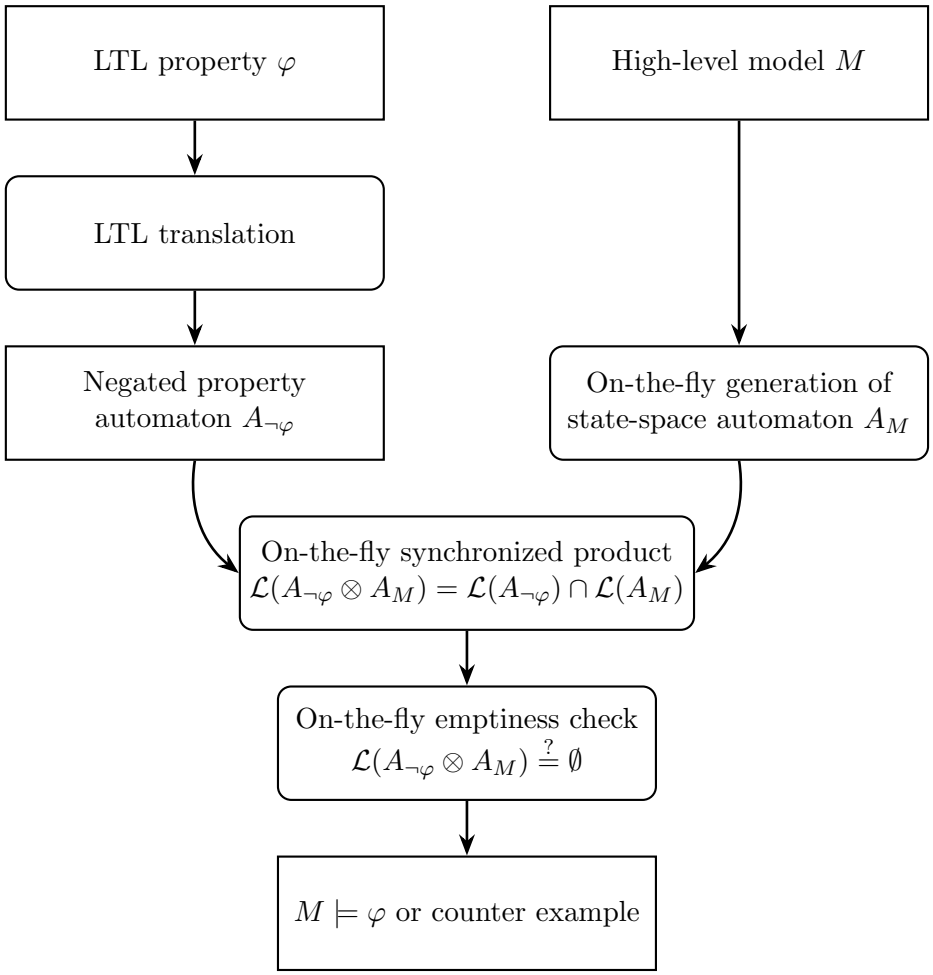

Figure 2.6: Automata-theoretic explicit LTL model checking

while particularly shedding the light on DCR representations which we will be focusing on as a means for the representation of the behavioural context of smart contracts. The last section of this chapter presented the arsenal of techniques that will be used throughout our contributions, namely model checking as formal verification method, Coloured Petri Net as a representation formalism and Linear Temporal Logic for the specification of the properties to be verified for the correctness of smart contracts.

Chapter 3

State of The Art

Contents

3.1 Introduction

In this PhD thesis, we aim to provide a solution to support the formal verification of Solidity smart contracts used in a BPM context. Therefore, in this chapter, we review the current state of the art in order to justify our problem statement and to have a clear position regarding the existing work. We start by identifying various studies touching on the verification of smart contracts in Section 3.2, among which we select a few to present in detail and compare. Then, in Section 3.3 we shift the focus onto studies that treat the formal verification of diverse aspects of Business processes.

3.2 On the Verification of Solidity Smart Contracts

The different attacks on the different second-generation Blockchain platforms brought light on the various vulnerabilities that they may suffer from and acted as an effective incentive for experts to work on finding suitable solutions to the errors that may be at the source of such weaknesses. The efforts put into this quest took different directions (cf. Figure 3.1). Some solutions were based on informal methods while others aimed for more formal verification approaches.

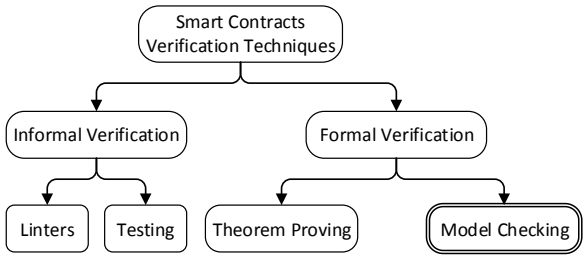

Figure 3.1: Smart Contracts verification techniques

In this section, we start by an overview of the main informal techniques used for the verification of smart contracts and then we explore the formal techniques used for this purpose. We try to put together an exhaustive list of approaches for the verification of Solidity smart contracts. We resorted essentially to two sources in our quest, namely the Google Scholar search engine and the DBLP computer science bibliography, and used combinations of the following keywords for the search: smart contract, formal verification, Solidity and Ethereum. We then recursively pursued the references included in these papers' related work citations. We came out with a plethora of material, from which we selected 13 studies based on their relevance to the subject, the uniqueness of the proposed approach and the number of citations. Our selection was also guided by previous surveys that were conducted on more generic scopes. For instance, some studies did not focus on the formal aspect of the proposed verification approaches for Solidity smart contracts but rather on their analysis capabilities [23], while others chose to cover more smart contract languages [43] to the detriment of being exhaustive in their papers selection. In this section, we only explore studies related to Solidity as it is the language considered in this PhD thesis, and focus on verification proposals that use formal approaches, hence offering a more in-depth analysis of the existing formal verification approaches for Solidity smart contracts.

3.2.1 Informal Verification of Solidity Smart Contracts

Informal techniques are usually associated with validation rather than verification [20]. The most common techniques that fall under this category are testing and simulation.

In fact, one straightforward way to minimize the risk of deploying a vulnerable smart contract is to take advantage of one of the many existing testnets which are, as their name suggests, alternate Blockchains dedicated for testing purposes. A smart contract can, for example, be run on Ropsten [11] before its deployment on the mainnet, which may help with plain defects but not with imperceptible ones.
A user can also resort to security companies such as MagicBlock- chainQA [6]. Such services are, however, both time- and money-consuming and do not guarantee in any case a fault-free smart contract.
The Solcover testing tool [88] was developed to offer a free and automatic testing experience of smart contracts. According to the tool's associated blog article, it "should only be treated as another arrow in a collective quiver", as it is unable to fully ensure the correctness of a smart contract.
Instead of trying out scenarios that may or may not instigate erroneous behaviors, other researchers worked on enforcing security and best practices rules through linters [15,16,28,41], which are tools that analyze the code to identify and flag programming and stylistic errors or suspicious constructs.

Such informal techniques may offer the convenience of quickly checking whether a system would meet some requirements if executed within a specific scenario (or set of scenarios), but they cannot, however, offer any guarantees of correctness as one cannot be expected to manually test all possible scenarios or predict them for that matter.

3.2.2 Formal Verification of Solidity Smart Contracts

While informal methods may reduce the risk of bugs in smart contracts, relying solely on such techniques cannot be enough to get a full insurance that a smart contract would be correct. Formal verification techniques can overcome this weakness, though it may come at the expense of other challenges such as scalability. Such techniques are used to check the conformance of the developed system to a predefined specification. They are based on formal methods of mathematics and are able to provide formal proof of the correctness of the investigated system with reference to its formally specified behavior. We can distinguish mainly two families of formal verification methods, namely those based on *theorem proving* and those based on *model checking*.

3.2.2.1 Verification Approaches Based on Theorem Proving

Some researchers proposed theorem proving-based approaches for the verification of Solidity smart contracts. The authors in [26] propose Solidity* a prototype tool, implemented in OCaml, that allows the translation of a restricted subset of Solidity into F*, a functional programming language for program verification. In order to detect dangerous patterns, the user then needs to define effects in F* code which are discharged by the F* type-checker. They also propose EVM*, a decompiler for EVM bytecode into F*, along which they propose a model for the cost of bytecode

operations which can be used by creating annotations for gas-related violations that can be discharged by the F* type-checker. Using this approach requires, not only expertise in F*, but also an understanding of the proposed translation in order to be able to express the patterns to be checked and understand the generated typechecking errors. Members of the Ethereum community present a prototype for verification integrated into the *Solc* compiler of Solidity [4]. Their proposition leverages the Why3 IDE, a theorem prover which can be used on the WhyML code generated by calling *Solc* with specific attributes. Other partial translations of the EVM bytecode based on assisted proofs like Coq [52] and Isabelle/HOL [21] exist. We note that none of these approaches offer automatic verification of smart contracts.

3.2.2.2 Verification Approaches Based on Model Checking

Other researchers proposed model-checking-based approaches for the verification of Solidity smart contracts. Table 3.1 presents a categorization of these works depending on the used techniques.

Table 3.1: Smart contracts verification approaches categorized by the used methods

Approaches based on Theorem Proving	**Approaches based on Model Checking-related techniques**			
	Symbolic Execution	**Abstraction**	**SAT/SMT solvers**	**Model Checking**
[26] [4] [52] [21]	[79] **[95]** **[64]** [103] [34]	[97] [31]	[79] **[95]** **[64]** [103] [34] **[57]**	**[66, 67]**

Oyente [64] was the first attempt at formal smart contract verification. It uses symbolic execution applied at the EVM bytecode of the contract to generate symbolic execution traces among which it looks for certain conditions that translate the presence of one of the four vulnerabilities it targets. This proposition actually paved the way for other researchers who wanted to do better in several subsequent studies. Some of them reused it as part of their own tools, like in GASPER [34] which exploits the by-product of Oyente (CFG) in its detection of costly bytecode patterns in terms of gas consumption. Other researchers opted for extending Oyente to detect different/additional bugs (e.g., MAIAN [79], SASC [103] and Osiris [95]). Securify [97] is a security analysis tool for Solidity smart contracts. It starts by decompiling the EVM bytecode into a static-single assignment form and symbolically encoding the corresponding dependence graph in stratified DataLog, leverages the Soufflé solver to derive semantic facts on the contract's data- and control-flow dependencies using declarative inference rules and then checks for the presence of predefined patterns that correspond to the properties the user wants to verify. In fact, the authors use a designated DSL to define compliance (resp. violation) patterns for a number of properties to capture sufficient conditions in a given code to satisfy (resp. violate) such properties. Even though the user can define other patterns to check for additional security properties, it is not possible to define patterns that match a contract-specific

property, or arithmetic properties. Besides, in some cases the code does not match any defined pattern and cannot decide on the safety of the contract. Vandal [31] follows the same spirit and adopts a logic-driven program analysis approach. It starts by translating the EVM bytecode into an abstract register transfer language exposing its data- and control-flow structures. This language is then translated into logic relations which are then fed to security analyses written in Soufflé to detect certain vulnerabilities in the contract.

3.2.3 A Selection of Prominent Tools for the Formal Verification of Solidity Smart Contracts

Among the studies that we have collected on formal verification of Solidity smart contracts, four can be categorized as theorem proving-based approaches and 9 make use of model checking-related techniques. We note that, despite this second category representing a majority, only one work proposed an approach fully based on model checking in the proper sense of the word (cf. Table 3.1). Such propositions are rather walking the line between being formal verification-based and testing-based approaches.

In the following subsections, we present four selected approaches proposed for formal verification of smart contracts, presented in bold in Table 3.1. We choose to detail the two approaches presented in [64] and [57], the two approaches with the most cited papers, as well as [66, 67] for being the single approach based on model checking. We also single out the approach in [95] as one of the propositions based on the veteran Oyente [64].

Throughout the rest of this section we will be using the illustrative contracts in Listings 3.1 and 6.2 to test the different tools proposed in the selected studies. We note that these contracts are written for illustration and do not exhibit a logical functionality.

```
1  contract VulnContract {
2    mapping (address=>uint) balances;
3    uint256 result;
4    event started();
5    function Deposit() {
6      balances[msg.sender] += msg.value;
7    }
8    function Withdraw(uint amount) {
9      if(balances[msg.sender] >= amount) {
10       msg.sender.call.value(amount);
11       balances[msg.sender] -= amount;
12     }
13   }
14   function Multiply(uint8 x, uint8 y) returns (uint8) {return x * y;}
15   function NewYear(){
16     if(block.timestamp > 1609459199)
17       emit started();}
```

```
18      function CostlyLoop(uint256 x) {
19        for(uint256 i = 0; i < x; i++)
20          result += i;
21      }
22    }
```

Listing 3.1: A vulnerable smart contract in Solidity

```
contract MaliciousContract {
  uint balance;
  VulnContract vc = VulnContract(0xbf0061dc...);
  function ReentrancyAttack() {
    balance = msg.value;
    vc.Deposit.value(balance)();
    vc.Withdraw.(balance);
  }
  function () payable {
    vc.Withdraw.(balance);
  }
}
```

Listing 3.2: A malicious smart contract in Solidity

3.2.3.1 FSolidM and VeriSolid

Approach In [66] the authors propose an FSM-based approach for the design of secure smart contracts. The premise of their work is that writing smart contracts in a language such as Solidity is error-prone because the smart contract writer may not fully grasp the semantics driving the execution process which often leads to a contract that does not reflect the actual intentions of its creator. They hence aim at closing this semantic gap by developing the FSolidM tool which allows users to design a smart contract as an FSM (Finite State Machine) which is then automatically transformed into a Solidity smart contract. To do so, they propose a definition of a smart contract as an FSM and outline the transformation process that generates the corresponding Solidity smart contract.

To improve the generated smart contract's security, the authors propose so called "plugins" that prevent some common vulnerability patterns [24]. These plugins actually translate into modifiers appended to the contract's functions to be secured. In Solidity, a modifier is used to change the behavior of the functions with which it is associated. In this context, modifiers are used to implement security patterns into the generated Solidity functions, e.g., by adding preconditions to check prior to their execution.

The work presented in [66] was in fact laying ground for the next paper [67] in which the authors present VeriSolid, the improved version of FSolidM. In fact, [67] extends [66] in that it adds formal operational semantics to the formerly proposed FSolidM model and therefore extends the Solidity code generator. This upgrade

introduces the aspect of formal verification into the tool, which provides the user with the ability to specify intended behavior in the form of liveness, deadlock freedom and safety properties. It offers customizable templates to express and check some CTL properties by a backend symbolic model checker.

Tool The FSolidM tool offers four plugins to deal with four types of vulnerabilities: (1) a locking plugin against the reentrancy attack pattern, (2) a transition counter plugin to enforce transition ordering and avoid falling into unpredictable states, (3) an automatic timed transitions plugin to implement time-constraint patterns and (4) an access control plugin to manage authorization for the execution of certain functions.

While VeriSolid extends FSolidM, it does not take into account the same vulnerabilities as the latter since it offers the possibility to express CTL properties through templates such as "transition_b will eventually happen after transition_a" which can be used to check for a denial-of-service vulnerability. Additionally, the authors chose to deal with the reentrancy vulnerability intrinsically by automatically introducing an In-Transition state into which the system goes at the beginning of each transition, thus prohibiting any overlapping calls.

In order to incorporate the formal verification aspect, the authors resort to using the NuXmv symbolic model checker [33] which features SMT-based techniques for the verification of infinite state systems. For that, they opt for augmenting the initial FSM model to take in the semantics of the Solidity functions' statements, transforming the resulting augmented model into a BIP (Behavior-Interaction-Priority) transition system [25] (which is guaranteed to be deadlock-free), using an existing BIP2NuSmv transformation tool and feeding the result to the NuXmv model checker along with the CTL formulae following the provided properties templates. For lack of an underlying

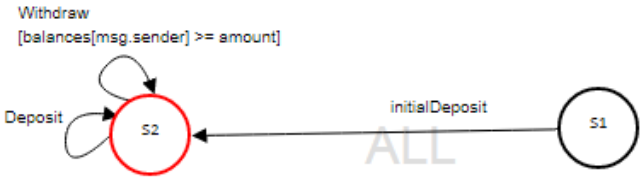

Figure 3.2: Screenshot of the VeriSolid tool - 1

business logic in our example in Listing 3.1, we choose to test the tool on a small example for reentrancy. We note that in this model we distinguish the first call to deposit (InitialDeposit) from the rest to get around a modeling restriction that requires the system to contain a minimum of 2 states.

```
1 contract VulnerableContract {
2   uint private creationTime = now;
3   enum States {InTransition, S1, S2}
4   States private state = States.S1;
5   mapping (address=>uint) balances;
```

```
6   function Deposit () public {
7     require(state == States.S2);
8     state = States.InTransition;
9     balances[msg.sender] += msg.value;
10    state = States.S2;}
11  function InitialDeposit () public {
12    require(state == States.S1);
13    state = States.InTransition;
14    balances[msg.sender] += msg.value;
15    state = States.S2;}
16  function Withdraw (uint amount) public {
17    require(state == States.S2);
18    require(balances[msg.sender] >= amount);
19    state = States.InTransition;
20    msg.sender.call.value(amount);
21    balances[msg.sender] -= amount;
22    state = States.S2;}}
```

Listing 3.3: Solidity code generated by VeriSolid

Figure 3.2 shows the model we proposed and Listing 3.3 the generated Solidity code. We tested this code and confirmed its insusceptibility to reentrancy. Figure 3.3 shows the representation using VeriSolid of the property stating that the function Withdraw can only be called after the InitialDeposit function had been called, along with its verification result.

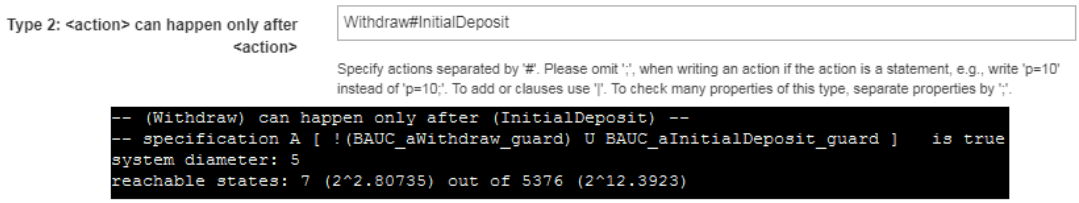

Figure 3.3: Screenshot of the VeriSolid tool - 2

Discussion In the attempt to integrate the formal verification aspect into the approach, the first premise of closing the semantic gap of Solidity got disregarded since the statements constituting the functions' bodies need to be provided by the user in Solidity. On the practical side, it may feel counter-intuitive and even restrictive for the user to have to think about the smart contracts they want to write in terms of states at design time, only to find themselves writing the code themselves nonetheless. Moreover, despite the help that may come with the proposed templates for CTL properties for some users, they might as well be seen as an unnecessary restriction to some other more experienced users who would like to verify more complicated properties that cannot be expressed within the limits of the provided templates. As much as some guidance is appreciated, it should not turn into a barrier to expressivity.

Last but not least, it is important to mention that the models in both FSolidM and VeriSolid do not take into account any variables. Therefore, no properties on the evolution of the values of the variables during the execution of the smart contract can be verified, which also cuts back considerably on the range of properties the user can check.

3.2.3.2 ZEUS

Approach Zeus [57] is a framework based on symbolic model checking for the verification of smart contracts. It takes as input a smart contract written in a high-level language along with a so-called policy that contains the criteria to be checked and which the user needs to specify in an XACML-styled template. The input smart contract code is then instrumented with assertion instructions according to its corresponding policy by means of static analysis and is passed on to a translator that the authors had devised to convert it into a low-level intermediate representation (LLVM bitcode) which is then fed to an existing verification engine in order to assert the safety of the smart contract. This is based on the primary description of the approach. As more details are later presented in the paper, we realize that this is not actually the exact right ordering of steps since the static analysis is afterwards said to be performed on top of the intermediate representation rather than the high-level code and the same goes for the added assertions. Later on, we also realize that the high-to-low level transformation is not straightforward. The authors propose an abstract language into which Solidity code is transformed before undergoing the first transformation into LLVM bitcode.

As to the considered properties, the authors distinguish two main families of vulnerable smart contracts: incorrect and unfair smart contracts. They define correctness as the adherence to safe programming practices and fairness as the adherence to agreed upon higher-level business logic.
An incorrect smart contract can have one of the following vulnerabilities: (1) reentrancy, (2) unchecked sends, (3) failed sends, (4) integer overflow/underflow or (5) transaction state dependence.
An unfair smart contract can have one of the following vulnerabilities: (1) absence of logic, (2) incorrect logic or be (3) logically correct but unfair.
Besides vulnerabilities that fall under these two categories, two more vulnerabilities which can actually be caused by the miner's influence are introduced: (1) block state dependence (BSD) and (2) transaction order dependence (TOD).

Tool Zeus was not made available online, but the authors state that they have implemented a prototype in C++. The tool's main components are the policy builder and the Solidity-to-LLVM bitcode translator. For the former, they leverage the AST output produced by the *Solc* compiler and taint analysis on the source code to extract the information needed to assist the user in forming the conditions to verify. As for

the translator, it takes as input the smart contract and uses existing LLVM APIs to generate the bitcode, which will then be instrumented by adding assertions according to the built policy. As a backend verifier, they opt for Seahorn [49].

Discussion Proposing an abstract interpretation language for Solidity to go through before obtaining an LLVM bitcode contributes in improving the scalability of Zeus. In fact, using over-approximations and reducing functions into summaries and loops into data domains results in a reduced state space for the symbolic model checker to be used later. However, a formal reasoning still needs to be established to prove the actual semantic equivalence between the two languages (Solidity and the proposed abstract language). Furthermore, the authors mention that using the abstract language allows the support of multiple Blockchain platforms, yet we think that using this bridge language constrains the high-level languages the tool can support. To integrate a language other than Solidity, new correspondences with the proposed abstract language would have to be defined (if not the whole language), the translation into LLVM would have to be revised and the automation of the assertions insertions would have to be reimplemented.

Leveraging the use of the LLVM bitcode extends the reach of Zeus in the sense that it can make use of any backend symbolic model checker supporting that standard. Seahorn [49] is the first choice of the authors but not the only one. It was chosen for its ability to generate verification conditions using CHCs (Constrained Horn Clauses) over LLVM bitcode. Other symbolic model checkers can be used, such as SMACK [87], but that may require some modifications on the LLVM bitcode as for example some model checkers might use different lengths for the same type which needs to be taken into account when switching the verifier. The authors state that CHCs are suitable for the representation of verification conditions, but do not elaborate. We think that the tool may be able to verify a wider range of properties if it were to support the representation of properties using other logics besides CHC.

Another point to mention is that Zeus can only account for parameters that can be computed at the source code level and hence cannot verify properties relating to parameters as gas consumption.

3.2.3.3 OYENTE

Approach Besides the smart contracts analysis tool they call Oyente, the authors of [64] also propose refinements/recommendations to Ethereum's protocol in the form of improvements to its operational semantics (e.g., new rules for transactions execution) in order to fix certain security problems. In this survey, we are only interested in the Oyente tool that they propose as a "pre-deployment mitigation". It is based upon symbolic execution and it functions over the contract's bytecode which needs to be provided as input along with Ethereum's global state. The latter would serve as an initialization for the contract's variables. Message call-related variables are however

treated as input symbolic values. The general idea behind Oyente is to symbolically explore a control flow graph corresponding to the contract's bytecode by symbolically executing instructions within states of that graph and using a symbolic constraint solver to decide on the feasibility of branching conditions. The possible presence of certain vulnerabilities is detected by checking for specific conditions in the generated symbolic traces.

This tool targets four specific vulnerabilities: (1) TOD, (2) timestamp dependence, (3) mishandled exceptions and (4) reentrancy.

Oyente signals whether the contract in question has one of the aforementioned problems and provides an example of a symbolic path to illustrate the possible problem to the user. The authors hope to turn Oyente into an interactive debugger as future work.

Tool Oyente [64] is implemented in Python, uses Z3 [74] as a backend SMT solver and detects the four discussed problems (see 3.2.3.3). Its design has four main components:

- CFGBuilder: it outputs a Control Flow Graph of the bytecode. This graph is only partly constructed statically as some edges are later added after symbolic execution.

- Explorer: this is mainly an interpreter loop that symbolically executes one instruction at one state at a time, starting from the entry node of the CFG generated by the previous component. The Explorer actually simulates the behaviour of EVM instructions and makes use of Z3 to decide on path conditions. The loop ends when no more unexplored states exist or when a timeout is reached. The CFG is potentially enriched by the end of this phase and a set of symbolic traces is outputted.

- CoreAnalysis: it in turn comprises four components to detect the four previously introduced bugs. These components work by checking specific conditions when analyzing the symbolic traces resulting from the Explorer in order to flag the possible presence of the corresponding bugs.

- Validator: this step is added to further reduce the rate of false positives. The user, however, still needs to intervene to confirm that the flagged bugs are a real threat.

The tool has been in active development up until May 2018, and some unreported features were added to the updated version. Mainly, in its latest version, Oyente can supposedly detect the following issues (in addition to the previously mentioned issues): (1) integer overflow/underflow, (2) Parity Multisig bug 2 and (3) callstack depth attack.

Tested on our example in Listing 3.1, the paper's version of Oyente was able to detect the timestamp dependency and the reentrancy vulnerabilities, as shown in the truncated results in Figure 3.4.

```
(venv)root@f0aa0167fa2b:/home/oyente/oyente# python oyente.py VulnerableContract.sol
0Contract VulnerableContract:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False

Reentrancy_bug? True

Added True
          Concurrency Bug:       False
          Time Dependency:       True
          Reentrancy bug exists: True
        ====== Analysis Completed ======
```

Figure 3.4: Screenshot of the Oyente tool

Discussion Oyente can be seen as the first attempt at formal smart contract verification, which paved the way for researchers in several subsequent propositions.

Despite its ability to detect important vulnerabilities in smart contracts, Oyente is not a complete verifier. Its major drawback is that its reported errors may be spurious. In other words, its results may contain false positives. One example for that is flagging a false reentrancy vulnerability in a code that uses a send function, which should not pose a threat unless its default gas were altered. This can actually be explained by the fact that Oyente relies on the bytecode of the smart contract, in which both functions send and call are mapped to the same CALL bytecode, which translates into contextual information loss. To detect reentrancy, the tool checks the path condition before each CALL it comes across and checks if it still holds after the bytecode's execution, in which case it is registered as a vulnerability.

3.2.3.4 OSIRIS

Approach Like in the previous study, in addition to proposing a verification approach, the authors in [95] indicate some ways to protect smart contracts against certain attacks by introducing modification to the EVM semantics as well as the Solidity compiler. We are only interested in their main contribution which is the proposed verification framework. This work specifically targets integer vulnerabilities in Solidity smart contracts. More precisely, the authors investigate the presence of 3 types of bugs in such contracts: (1) arithmetic bugs like integer underflows/overflows and bugs caused by divisions where the denominator is zero, (2) truncation bugs which can happen when converting a value into a new type with a shorter range than that of its initial type and (3) signedness bugs that can occur when converting a signed integer typed value into an unsigned integer type (or the opposite).

This approach works on integer bugs detection at the bytecode level and is based on two techniques, namely symbolic execution and taint analysis. It comprises 3 phases:

- Integer type inference: even though Solidity is a statically typed language, typing information is supposed to get lost at the bytecode level. The compiler, however, leaves behind discrete trails (e.g., AND bitmask, SIGNEXTEND opcode, etc) that the authors track down to deduce the size and sign of integers in the bytecode.

- Integer bugs detection: a different detection technique is proposed for each of the targeted integer bug types:

  - arithmetic bugs: a constraint is emitted to the backend solver for each arithmetic instruction. This constraint is formed so that it is only satisfied if a set of predefined in-bounds requirements specific to the instruction in question are not totally met. Consequently, a bug is detected if one of the emitted constraints under some path conditions is found to be satisfiable by the solver.
  - truncation bugs: such bugs are detected by tracking the instructions used by Solidity to perform truncation (i.e., AND and SIGNEXTEND for signed and unsigned integers). A constraint is formed for such instructions as to be satisfied if the input value is larger than the output. Consequently, a truncation bug is detected if one of the emitted constraints under some path conditions is found to be satisfiable by the solver, all while ignoring two specific patterns for intentional truncation corresponding to truncation due to a conversion to type address and truncation as a technique to fit more than one variable into the same storage slot.
  - signedness bugs: for this type of bugs, the authors reuse an approach that was initially proposed for Linux programs [72] and adapt it for Solidity smart contract. The gist of the applied method is to infer information on signed and unsigned types on the values from the executed EVM instructions and spot the symbolic variables that can be assigned both types.

- False positives reduction: the authors actually consider this as two separate steps, since they use two different techniques to reduce the rate of generated false alarms. The first step is to apply taint analysis in order to check only instructions whose input data is tainted (can be manipulated by an attacker) and further validate only the ones that touch sensitive locations (can be harmful in that they may alter the execution path, storage and ether flow). The second step of false positives reduction is recognizing detected integer bugs which originate from unharmful code such as an intentional check (if condition) meant to catch an overflow bug.

Tool The implemented tool called Osiris is written in Python. It operates over the bytecode but can accept Solidity code as input which it internally compiles into bytecode. It consists of 3 main components: (1) *symbolic analysis* is basically a reuse of the previously presented Oyente tool, used to generate the bytecode's CFG and symbolically execute its instructions, (2) *taint analysis* checks, for each executed instruction, whether it pertains to a specific set of instructions defined by the authors as susceptible of being used by an attacker, in which case the locations it affects (in the stack, memory and storage) are tagged and the propagation is carried out according to the EVM semantics. It then checks if this instruction can be impactful on sensible locations and (3) *integer error detection* is called upon the instructions detected by the taint analysis, implements the errors detection methods discussed above and uses Z3 to check for the feasibility of the created constraints.

Figure 3.5 shows the result for running Osiris on our example in Listing 3.1. The tool detected an overflow bug as well as a truncation bug and located them in the code.

```
root@12b001d56c5a:~/osiris# python osiris.py -s VulnerableContract.sol
INFO:root:Contract VulnerableContract.sol:VulnerableContract:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     99.7%
INFO:symExec:     Arithmetic bugs:       True
INFO:symExec:     └> Overflow bugs:      True
VulnerableContract.sol:VulnerableContract:23:4
result += i
^
VulnerableContract.sol:VulnerableContract:15:10
x * y
^
INFO:symExec:     └> Underflow bugs:     False
INFO:symExec:     └> Division bugs:      False
INFO:symExec:     └> Modulo bugs:        False
INFO:symExec:     └> Truncation bugs:    True
VulnerableContract.sol:VulnerableContract:14:2
function Multiply ( uint8 x, uint8 y) returns (uint8 ) {
^
INFO:symExec:     └> Signedness bugs:    False
INFO:symExec:    --- 8.4284760952 seconds ---
```

Figure 3.5: Screenshot of the Osiris tool

Discussion This work focuses on a restrictive range of vulnerabilities, covering specific integer-related bugs. On the one hand, the approach shows better results than other existing approaches dealing in part with such vulnerabilities, yet its range of application is thereby restricted. Additionally, Osiris points out the origin of the detected vulnerability in the analysed code but does not provide an example of an execution that may lead to an error, which would make it easier for the contract writer to revise the code.

3.2.3.5 Comparison and Discussion

The majority of the discussed approaches are based on the analysis of the EVM bytecode instead of the higher-level Solidity source code which can be explained by Solidity's lack of formal semantics. In fact, Solidity comes with a well-specified grammar [5] that defines its syntax, but there is no official source for a formal definition of its semantics which is only informally specified through textual description in the Solidity documentation [12] and directly implemented into its compilers. This has led to recent efforts to provide formal semantics to Solidity as in [56] that proposes an executable operational semantics for Solidity using the K-framework and [65] in which the authors propose an executable denotational semantics for Solidity in the Isabelle/HO theorem prover. In general, relying on the bytecode has its own impediment since it leads to the loss of contextual information, hence limiting the range of properties that can be verified on the contract.

Table 3.2: Vulnerabilities supported by the proposed smart contract verification approaches

		[67]	[57]	[97]	[31]	[64]	[95]	[79]	[34]	[103]
Can express properties as	Predefined templates for CTL expressions	+	-	-	-	-	-	-	-	-
	XACML templates for CHC	-	+	-	-	-	-	-	-	-
	Compliance/violation patterns in a DSL	-	-	+	-	-	-	-	-	-
	Security analysis as logic spec in Soufflé	-	-	-	+	-	-	-	-	-
Reported detected vulnerabilities	Limited stack size	-	-	-	-	+	-	-	-	+
	Arithmetic bugs	-	±	-	-	±	+	-	-	±
	Timestamp dependence	-	+	-	-	+	-	-	-	+
	Transaction order dependence	-	+	+	-	+	-	-	-	+
	Reentrancy	+	+	+	+	+	-	-	-	+
	Self destruction	-	-	-	+	+	-	+	-	-
	Gas run-out	-	-	-	-	-	-	-	-	-
	Other	+	+	+	+	+	-	+	+	+

We notice that most of the proposed approaches, led by the first proposition [64], use symbolic execution to generate the traces that would be used for the verification. Such approaches usually use under-approximation (e.g., by setting loop bounds) which means that critical violations can be overlooked.

A survey on the vulnerabilities in smart contracts [40] reports 49 bugs that can occur in a smart contract, 29 of which were categorized using the Bugs Framework of NIST into 10 bug classes. As shown in Table 3.2, the proposed verification methods only target a limited number of these bugs, with a maximum of 18 claimed by the

commercialized version of Securify [97]. We also note that 4 approaches give the user the ability to express customized properties to check. None of them, however, supports contract-specific properties. Moreover, we underline that only single function reentrancy is considered in all of the existing approaches. Furthermore, none of the proposed approaches deals with the verification of interacting contracts. This means that the verification of smart contracts is a field that, despite having been investigated at an early stage, still needs to be further studied to achieve correctness in smart contracts and consolidate the desired trust in the Blockchain environment.

In the following we report tools comparisons included in their corresponding papers.

Zeus vs Oyente in [57] The evaluation of Zeus was done on a dataset of 1524 smart contracts and its results were compared to Oyente's for the commonly treated vulnerabilities (reentrancy, unchecked send, BSD and TOD). 54 contracts were reported by Zeus to be vulnerable to reentrancy against 265 by Oyente. The undetected bugs by Zeus were said to be false positives caused by Oyente considering reentrancy possible with send calls. This is not totally true, as using send can still be susceptible to reentrancy if the allocated amount of gas were to be manually increased. For the unchecked send vulnerability, Zeus was reported to detect 324 bugs with 3 false positives, against 112 bugs by Oyente with 89 false positives. The results for BSD show more detected bugs by Zeus than Oyente, which is only logical since the former considers multiple block variables while the latter only considers the block's timestamp. Zeus is also reported to detect more TOD bugs (607) than Oyente (126) with a lower false alarm rate.

Osiris vs Zeus in [95] The authors of [95] evaluated their tool using a subset of the dataset of smart contracts previously used by Zeus (they retrieved 883 out of 1524 contracts), and compared it to the latter for their commonly detectable bugs. Their reported results show a big difference in the number of detected integer overflow/underflow bugs with Zeus detecting 628/883 and Osiris detecting 172/883. They claim that this difference can be explained by Zeus prioritizing completeness over the real exploitability of its reported bugs. They also bring into question Zeus's soundness by manually investigating 5 contracts that were reported as containing bugs by Osiris but not by Zeus and confirming their unsafety.

SASC vs Oyente in [103] In their comparison with Oyente, the authors in [103] report more detected timestamp dependence bugs using their tool (866 by SASC against 292 by Oyente out of 2952 contracts). In fact, this can be explained by the different ways both tools use to target such a vulnerability. While Oyente detects the use of timestamp whenever it is related to ether transfer only, SASC proceeds differently by targeting its use in other operations as well. We also note that SASC

is able to locate the bugs in the corresponding code, unlike Oyente that only signals their presence.

Securify vs Oyente in [97] Results were compared to Oyente in [97] for the detection of reentrancy, TOD and mishandled exceptions. The authors report better results overall for Securify. Reentrancy was detected in the same number of contracts by both tools, with presence of false positives with Oyente but not with Securify. As for the other two vulnerabilities, Securify was reported to detect more valid occurrences than Oyente and no false negatives at all, albeit with a slightly higher number of false positives.

3.2.3.6 Coloured Petri Nets for Smart Contracts

More recently, attempts have been made to use Coloured petri net for the verification of smart contracts. The work in [62] shows an example of verification of behavioural properties on a CPN model for a crowdfunding smart contract. It does not, however, propose a complete and generic approach to apply on any smart contract as the work merely presents case study of a manual passage from the use case's contract to the CPN model without defining general rules for the transformation. Another CPN-based proposition was presented in [42]. The authors start from the contract's bytecode and use Hoare's logic to generate the corresponding CPN model which is then used for the security analysis of the contract. This approach, despite being based on CPN, cannot be used for the verification of data-flow related properties as the generated model focuses on the representation of the workflow extracted from the contract's CFG.

3.3 On the Verification of Business Process Models

A primordial phase in the life cycle of a business process is its verification. In order to avoid execution mishaps whose gravity may vary with the level of criticality of the modeled system, the designer needs to make sure that the model will actually satisfy a set of requirements/specifications that define the correctness of the system. This matter has been studied extensively in the case of imperative representations, as can be seen by the survey conducted in [73]. BPMN being one of the most widely used modeling languages for business processes, it is fairly easy to find in the literature many studies that focus on the formal verification of various aspects of BPMN models [32, 39, 47, 81, 86, 102]. In such studies, it is frequent to encounter Petri nets as a formalism used to add a formal aspect to the semantics of BPMN. In [39], the authors propose a mapping of the core elements of the BPMN language into labelled Petri nets patterns. This mapping was then implemented into a tool that automated the generation of the Petri net and helped the semantic analysis of the modeled process. Another Petri-net-based approach was proposed in [81] with the aim of analysing the

feasibility of BPMN processes, this time using Colored Petri nets as a target formal representation and a modified BPEL4WS representation as a pivot language. YAWL is another important modeling language for business processes. It is based on Petri nets and extended with specific concepts to easily model complex workflows. An automation of the transformation of BPMN models into YAWL was proposed in [102] with the aim of verifying them using specific tools such as Woflan [99]. The aforementioned studies merely mark the starting point of the research on the verification of business processes. Several other studies followed on the same path, using extensions of Petri nets to take into account more complex models [58], and focusing on specific aspects such as in [29, 54].

Other formal methods have been used for the verification of business processes like π-calculus in [86], event calculus in [47] and theorem proving in [32].

Similar to the imperative languages of business process modelling, the emergence of the declarative modelling mode needs to be accompanied by suitable tools that would allow the verification of its models. In fact, the implicit nature of the representation of the workflow makes it less obvious to be interpreted by the designers and therefore makes it easier for simple modelling errors to pass unnoticed. Moreover, the dynamic and complex nature of the domains that require flexible models puts more at stake and adds to the importance of having a verification tool that would insure the correctness of the process model. However, compared to the rich state of the art on the verification of imperative business process models, we notice a significant scarcity in research on the verification of declarative models.

3.4 Conclusion

This chapter serves to present different approaches that are relevant to our work. First, we reviewed the existing work on the verification of Solidity smart contracts. We gave an overview of the solutions that use informal techniques and focused on those that use formal techniques. We classified those into two main categories: *(1)* approaches based on theorem proving and *(2)* approaches based on model checking, which we then sub-categorized based on *i* the exact technique used for the verification and *ii* their ability to verify the common vulnerabilities and possibility to verify other properties. We selected four studies based on their relevance to our work and their citation score, to present in detail, discuss and compare. Then we provided an overview of the use of formal verification in context of BPM, showcasing the importance of this phase in relation to different aspects of the Business processes.

Despite the multiple studies proposing formal verification approaches for Solidity smart contracts, we showed that most of them use symbolic execution which leads to incomplete verification results, and that none of them gives the users the possibility to freely express the properties to be verified on their smart contracts, especially properties that can depend on the data-flow of these contracts. This last point can be very useful particularly if the behavioural context in which the smart contracts

are to be used is well specified, in which case it would be interesting to verify the smart contracts with regard to properties expressed on this context. Given that such a context can intuitively be specified as a Business process, we also showed the significant attention bestowed on the formal verification of Business process models and highlighted the contrast between the literature on imperative representations against that on declarative representations to conclude that the formal verification of declarative models is still not mature. This further corroborates the need for the verification of smart contracts especially when accompanied with a behavioural context specification.

Chapter 4

Formal Modeling of Solidity Smart Contracts

Contents

4.1 Introduction

This chapter represents the first contribution of our work, which consists in the first step towards our end-goal approach (Figure 4.1). With this contribution we aim to achieve ***Obj1*** which is part of the response to ***RQ1***. Herein we are interested in the transformation of Solidity smart contract functions into what we call CPN building blocks or sub-models. To do so, we propose generalized algorithms that transform each type of statement into such CPN sub-models.

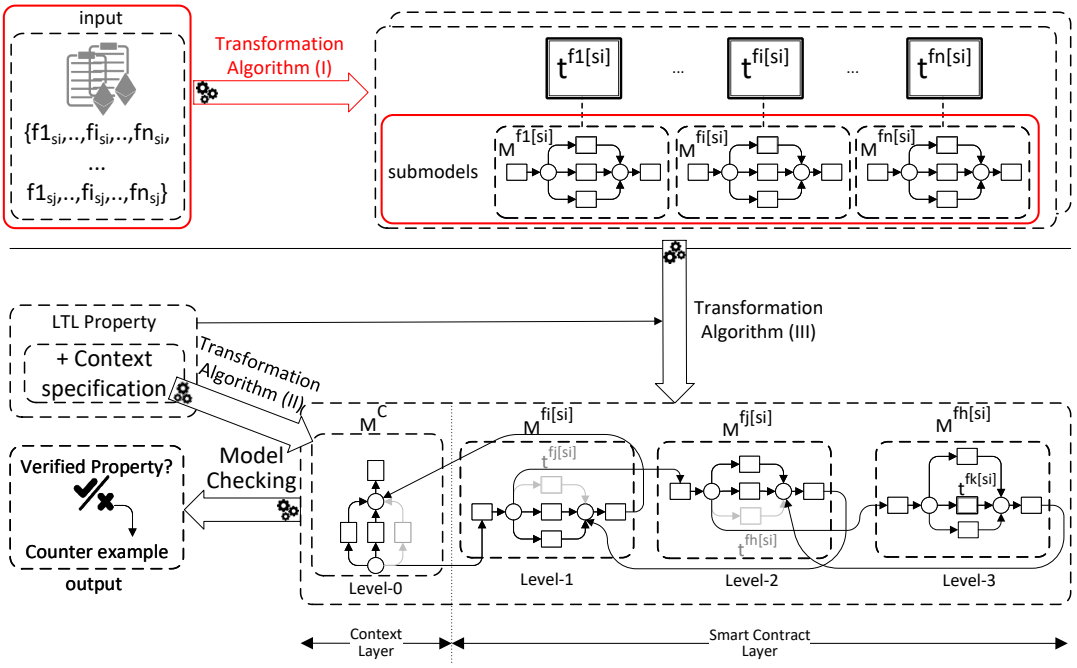

Figure 4.1: Approach overview - Step 1

We lead off by setting some notations on the elements of the model in Section 4.2 before getting into the algorithms in Section 4.3.

4.2 Defining the Elements and Notations for our CPN Models

In this section, we give details on the different elements that we use in our proposed algorithms for the construction of the our CPN building blocks.

4.2.1 Transitions T

We distinguish two types of transitions in the sub-models:

1. aggregated transitions (T^A): used for the modular representation of function calls. They are transitions that can be substituted by sub-models.

2. regular transitions (T^R): are simple unsubstitutable CPN transitions.

For a transition $t \in T$ we note:

- $t.name$, the name of the transition t
- $t.statement$, the Solidity code associated with transition t
- $t.metaColour$, the metaColour associated with the control flow places of transition t (if $t \in T^A$)
- $t.data$, the set of data places associated with transition t (if $t \in T^A$)
- $t.sub-model$, the CPN sub-model associated with transition t (if $t \in T^A$), with $t.submodel.inTransitions$ designating its input (source) transitions and $t.submodel.outTransitions$ designating its output (sink) transitions
- $t.guard$, the guard of the transition t

4.2.2 Places P

We define 4 types of places according to the role they play:

- *Control flow places* P_{CF} are places created to implement the order of execution of the workflow. We also use them to carry data related to the state of the smart contract which can be defined by its balance and the values of its state variables. Such places have a *metaColour* defined at each aggregated transition of level-0 as the concatenation of the *state* and the *input parameters*: *[uint: contractBalance,* $type_{v_1}$*:* $stateVariable_1$*, ...,* $type_{v_n}$*:* $stateVariable_n$*,* $type_{p_1}$*:* $inputParameter_1$*, ... ,* $type_{p_n}$*:* $inputParameter_n$*].*
- *Data places* P_{data} (for internal local variables) where each place is of a colour corresponding to the represented variable's type.
- *Parameter places* P_P that convey potential inputs of function calls. Each function call has an associated parameter place whose colour is as follows *[*$type_{p_1}$*:* $inputParameter_1$*, ...,* $type_{p_n}$*:* $inputParameter_n$*].*
- *Return places* P_R that communicate potential functions' returned data. Each function call has an associated return place whose colour corresponds to the return type of the called function.

4.2.3 Expressions E

An expression is a construct that can be made up of literals, variables, function calls and operators, according to the syntax of Solidity, that evaluates to a single value. For ease of representation, we define three types of expressions:

- expressions with variables E_V: are expressions that make use of at least one local variable. In such an expression e_v, the set of variables used is accessible via $e_v.vars$.
- expressions with function calls E_F: are expressions that make use of at least one function call. In such an expression e_v, the set of function calls used is accessible via $e_v.fctCalls$
- constant expressions E_C: are expressions that do not make use of any variables nor function calls.

We note that an expression e can of course have both variables and function calls ($e \in E_V \cap E_F$).

4.2.4 Statements S

In this work, we cover the core features of the Solidity language according to the definition of its grammar [5]. While we do not take into account *all* the Solidity elements, the ones that we support can be considered exhaustive in the sense that they can be used to rewrite smart contracts that include more *syntactic sugar*. We note however, that we do not support smart contracts that include raw assembly code in them as our main focus is the verification of the Solidity high-level code.

A statement $st \in \mathbb{S}$ can be either a compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ (where $\forall i \in [1..N]$, $st[i] \in S$), or a simple statement (st_{LHS}, st_{RHS}) (where $st_{LHS} \in E$ and $st_{RHS} \in E$), or a control statement. A simple statement can be:

- a function call statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$ designates the set of variables used in the arguments of the call (if $st_{RHS} \in E_V$)
- an assignment statement, where:
  - $st_{LHS} \in E_V$ and $st_{LHS}.vars$ contains only one variable that designates the assigned one
  - $st_{RHS}.vars$ designates the set of variables used in the assignment expression (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls used in the assignment expression (if $st_{RHS} \in E_F$)

- a variable declaration statement, where:
  - $st_{LHS} \in E_V$ and $st_{LHS}.vars$ contains one variable that designates the declared one
  - $st_{LHS}.type$ designates the type of the declared variable
  - $st_{RHS}.vars$ designates the set of variables used in the variable initialization expression (if the variable is initialized and $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls used in the variable initialization expression (if the variable is initialized $st_{RHS} \in E_F$)
- a sending statement, where:
  - st_{LHS} designates the destination account
  - $st_{RHS}.vars$ designates the set of variables in the expression of the value to be sent (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls in the expression of the value to be sent (if $st_{RHS} \in E_F$)
- a returning statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$ designates the variables in the expression of the returned value (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the function calls in the expression of the returned value (if $st_{RHS} \in E_F$)

A control statement can be:

- a requirement statement of the form $require(c)$
- a selection statement which can have:
  - a single-branching form: $if(c)\ then\ st_T$
  - a double-branching form: $if(c)\ then\ st_T\ else\ st_F$
- a looping statement which can be:
  - a for loop: $for(init; c; inc)\ st_T$
  - a while loop: $while(c)\ st_T$
- where:
  - c is a boolean expression
  - $c.vars$ designates the set of variables used in the condition (if $c \in E_V$)
  - $c.fctCalls$ designates the set of function calls used in the condition (if $c \in E_F$)
  - st_T, st_F, $init$ and inc are statements (or blocks of statements)

4.2.5 Additional Notations

For a smart contract SC we note:

- $SC.vars$, the set of state (i.e., global) variables of SC
- $SC.fcts$, the set of functions of SC

For a place $p \in P$ we note:

- $p.name$, the name of the place p
- $p.colour$, the clour of the place p

For a transition $t \in T$ we note:

- $\bullet t[cf] \in P_{CF} \cup P_S$, the input control flow place of t
- $\bullet t[input] \in P_P$, the input parameters place of t
- $\bullet t[data] \subseteq P_{data}$, the input data places of t
- $t \bullet [cf] \in P_{CF} \cup P_S$, the output control flow place of t
- $t \bullet [output] \in P_R$, the output return place of t
- $t \bullet [data] \subseteq P_{data}$, the output data places of t

4.3 Solidity-to-CPN: Preparing the Building Blocks

In the following, we present our proposed algorithms for the transformation of each of the statement types in Solidity. It is worth mentioning that in our approach we assume that the input smart contracts are syntactically correct as our aim is to verify behavioural properties rather than syntactical ones. If the input contracts are syntactically incorrect then our algorithms will evidently generate incorrect CPN sub-models. We do not think that this is a heavy assumption as any Solidity compiler is capable of detecting syntactical errors.

4.3.1 Generation of the Aggregated Transitions

The first step is to create an aggregated transition for each of the contract's functions. To do so, we propose the algorithm GENERATEAGGREGATIONS.

1: **procedure** GENERATEAGGREGATIONS(SC)
2: **Input:** a Solidity smart contract SC
3: **Output:** the aggregated transitions the CPN model of SC
4: $metaColour \leftarrow [uint : contractBalance]$
5: **for** $v \in SC.vars$ **do**

```
6:        add (v.type : v.name) to metacolour
7:    end for
8:    for f ∈ SC.fcts do
9:        create aggregated transition t^a
10:       t^a.name ← f.name
11:       t^a.statement ← f.body
12:       newColour ← metaColour
13:       for p ∈ f.params do
14:           add (p.type : p.name) to newColour
15:       end for
16:       t^a.metaColour ← newColour
17:   end for
18: end procedure
```

4.3.2 Preparing the Data Places

getLocalVariables creates a set of places to be used in the sub-model of a transition t^a, corresponding to the local variables used in its function. To do so, the statements in the function's body are recursively investigated in search for variable declaration statements. For each variable declaration statement found, a place bearing the name of the variable and its type as its name and colour is created and added to the set P_{data}. In addition to standalone variable declarations, we note that we can also find variables declared in the initialization of a For loop.

We opt for the construction of this set of places beforehand, as opposed to on the fly during the construction of the sub-model, for the following reason. In Solidity, a variable can be used before its declaration (as long as a declaration does exist). Creating its corresponding place on the fly while creating the sub-model of a transition would consequently require testing for its existence every time the variable is used in a statement, as the creation of the place in question may have to happen prior to the declaration statement, in any other statement using it (as part of st_{LHS} or st_{RHS}) for the first time. On this account, we judge it more efficient to sweep the code first for the construction of P_{data}.

```
1: procedure getLocalVariables(st; P_data)
2:    Input: statement st, set of places P_data being created
3:    Output: updated P_data with the set of places corresponding to local variables
      in the statement st
4:    if st is a variable declaration statement then
5:        create place p
6:        p.name ← st_LHS.vars.name
7:        p.colour ← st_LHS.type
8:        add p to P_data
9:    else if st is a selection statement then
```

```
10:         getLocalVariables(st_T, P_data)
11:         if st is a double-branching selection statement then
12:             getLocalVariables(st_F, P_data)
13:         end if
14:     else if st is a looping statement then
15:         if st is a for statement: for(init; c; inc)st_T then
16:             getLocalVariables(init, P_data)
17:             getLocalVariables(st_T, P_data)
18:         else if st is a while statement: while(c)st_T then
19:             getLocalVariables(st_T, P_data)
20:         end if
21:     else if st is a compound statement {st[1]; st[2]; ...; st[N]} then
22:         for i = 1..N do
23:             getLocalVariables(st[i], P_data)
24:         end for
25:     end if
26: end procedure
```

4.3.3 Creation of the Building Blocks

We see a smart contract function as a set of statements. To each one of the statement types we define a corresponding pattern in CPN, according to which a snippet of a CPN model is generated. The resulting snippets are linked according to the function's internal workflow. The *createSubModel* implements such correspondences[1].

```
1: procedure createSubModel(t; st; p_in; p_out)
2:     Input: transition t, statement st, control flow input place p_in, control flow
       output place p_out
3:     Output: sub-model of transition t
4:     switch st do
5:         case compound statement {st[1]; st[2]; ...; st[N]}
6:             buildCompoundStatement(t;st;p_in;p_out)
7:         case simple statement
8:             switch st do
9:                 case assignment statement
10:                    buildAssignmentStatement(t;st;p_in;p_out)
11:                case variable declaration statement
12:                    buildVariableDeclarationStatement(tst;;p_in;p_out)
13:                case sending statement
14:                    buildSendingStatement(t;st;p_in;p_out)
```

[1]We note that in case a place does not exist ($p = \emptyset$) then any arc creation involving that place does not take effect.

```
15:                 case returning statement
16:                     buildReturningStatement(t;st;p_in;p_out)
17:                 case function call statement
18:                     buildFunctionCallStatement(t;st;p_in;p_out)
19:             end switch
20:         case control statement
21:             switch st do
22:                 case requirement statement
23:                     buildRequirementStatement(t;st;p_in;p_out)
24:                 case selection statement
25:                     buildSelectionStatement(t;st;p_in;p_out)
26:                 case looping statement
27:                     switch st do
28:                         case for statement
29:                             buildForLoopStatement(t;st;p_in;p_out)
30:                         case while statement
31:                             buildWhileLoopStatement(t;st;p_in;p_out)
32:                     end switch
33:             end switch
34:     end switch
35: end procedure
```

createSubModel browses the body of the transition's corresponding function recursively, statement by statement, and creates snippets of a CPN model according to the type of the processed statement (cf. figures 4.2-4.12) that interconnect to create the transition's sub-model. In the following we give the transformation algorithm for each of the statement types, as well as their corresponding CPN patterns.

4.3.3.1 The Compound Statement Block

Compound statement $\{st[1]; st[2]; \ldots; st[N]\}$: the algorithm is re-executed on each component statement $st[i]$, after creating $N-1$ control flow places (of the *metaColour* colour) to interconnect the resulting CPN snippets while merging the entering point of the snippet of $st[1]$ with the entering point of the snippet of st and the exiting point of $st[N]$ to that of the snippet of st.

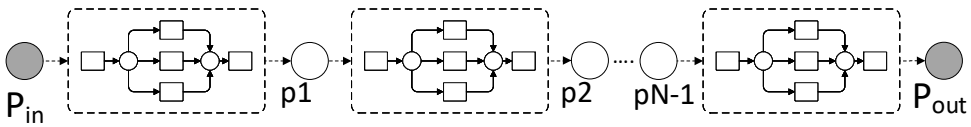

Figure 4.2: Compound Statement Pattern

1: **procedure** BUILDCOMPOUNDSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a compound statement $st = \{st[1]; st[2]; \ldots; st[N]\}$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: **for** $i = 1..N-1$ **do**
5: create place p_i
6: **end for**
7: CREATESUBMODEL*(t;st[1];p_{in};p_1)*
8: **for** $i = 2..N-1$ **do**
9: CREATESUBMODEL*(t;st[i];p_{i-1};p_i)*
10: **end for**
11: CREATESUBMODEL*(t;st[N];p_{N-1};p_{out})*
12: **end procedure**

4.3.3.2 The Assignment Statement Block

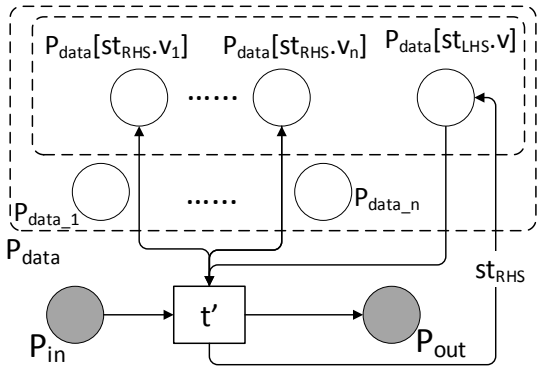

Figure 4.3: Local variable assignment

Assignment statement (st_{LHS}, st_{RHS}): to represent such a statement, a transition t' is created with input and output links to, respectively, the input (p_{in}) and output (p_{out}) places passed as parameters. t' is also connected to the places in P_{data} that correspond to the variables used in the statement's RHS with input/output links (to read the data). In case of a local variable assignment (Figure 4.3), an input/output link is created with the place corresponding to the assigned variable in the statement's LHS with the new value (st_{RHS}) inscribed on the output link. In case of a state variable assignment (Figure 4.4), the new value (st_{RHS}) is given in the variable's corresponding placement in the inscription on the link to the output (p_{out}) place.

1: **procedure** BUILDASSIGNMENTSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, an assignment statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st

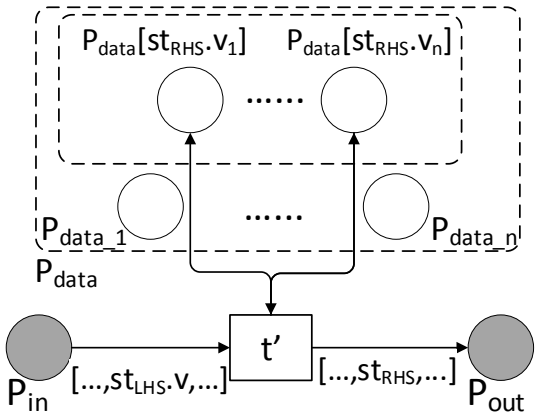

Figure 4.4: Global variable assignment

4: create transition t'
5: create arc from p_{in} to t'
6: connectLocalVariables*($st_{RHS}.vars \backslash \{st_{LHS}.vars\}$;t;t')*
7: connectFunctionCalls*($st_{RHS}.fctCalls$;t)*
8: **if** $st_{LHS}.vars$ is a local variable **then**
9: create arc from $t.data[st_{LHS}.vars]$ to t'
10: create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
11: create arc from t' to p_{out}
12: **else**
13: create arc from t' to p_{out} with inscription $outInsc \leftarrow inInsc$ in which the variable corresponding to $st_{LHS}.vars$ is replaced by st_{RHS}
14: **end if**
15: **end procedure**

4.3.3.3 The Variable Declaration Statement Block

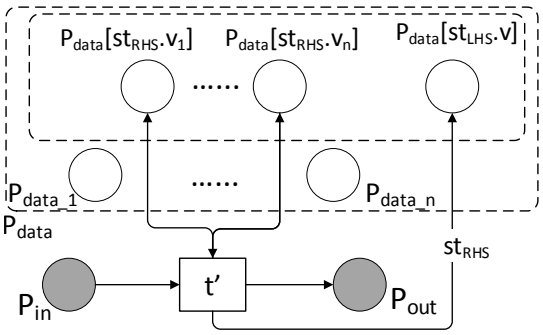

Figure 4.5: Variable Declaration Statement Pattern

Variable Declaration statement (st_{LHS}, st_{RHS}): the algorithm creates a transition t' with input and output links to, respectively, the input (p_{in}) and output (p_{out}) places passed as parameters. Input/output links are created from and to the places corresponding to the variables used in the statement's RHS. An output link is also created to the place representing the declared variable with st_{RHS} as inscription.

1: **procedure** BUILDVARIABLEDECLARATIONSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a variable declaration statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create transition t'
5: create arc from p_{in} to t'
6: CONNECTLOCALVARIABLES*($st_{RHS}.vars$;t;t')*
7: CONNECTFUNCTIONCALLS*($st_{RHS}.fctCalls$;t)*
8: create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
9: create arc from t' to p_{out}
10: **end procedure**

4.3.3.4 The Sending Statement Block

Sending statement (st_{LHS}, st_{RHS}): the CPN snippet for a sending statement is generated in a way that is similar to that of the case of a global variable assignment, except that instead of updating the assigned variable's counterpart on the output link to p_{out} the algorithm updates the *contractBalance* and *balance* values from the input of t' by decreasing the first and increasing the second by st_{RHS}.

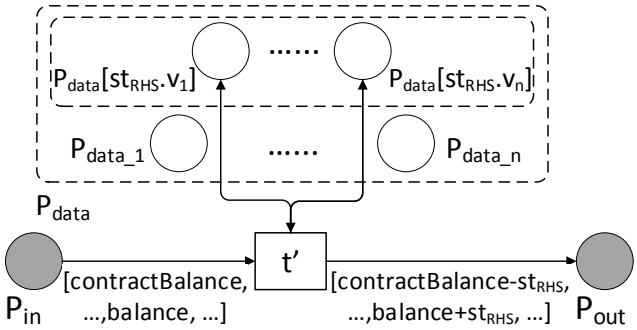

Figure 4.6: Sending Statement Pattern

1: **procedure** BUILDSENDINGSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a sending statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create transition t'
5: create arc from p_{in} to t'

6: connectLocalVariables$(st_{RHS}.vars;t;t')$
7: connectFunctionCalls$(st_{RHS}.fctCalls;t)$
8: create arc from t' to p_{out} with inscription $outInsc \leftarrow inInsc$ in which the variable corresponding to the sender's (respectively the contract's) *balance* is incremented (respectively decremented) by st_{RHS}
9: **end procedure**

4.3.3.5 The Returning Statement Block

Returning statement $(-, st_{RHS})$: this is also treated similarly to a global variable assignment statement, but with the following differences. t' is linked to $t\bullet[cf]$ instead of p_{out} with the same inscription of its input link with p_{in}, and has an additional output link with the output return place $t\bullet[output]$ with the return value set to st_{RHS} in its inscription, with the same sender's address and balance from its input inscription.

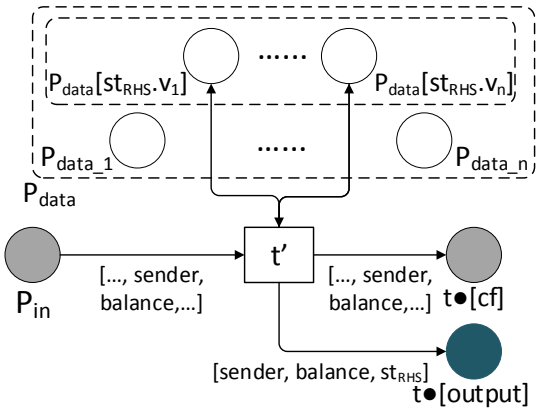

Figure 4.7: Returning Statement Pattern

1: **procedure** buildReturningStatement(t; st; p_{in}; p_{out})
2: **Input:** transition t, a returning statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create transition t'
5: create arc from p_{in} to t'
6: connectLocalVariables$(st_{RHS}.vars;t;t')$
7: connectFunctionCalls$(st_{RHS}.fctCalls;t)$
8: create arc from t' to $t\bullet[cf]$
9: create arc from t' to $t\bullet[output]$ with inscription $outInsc \leftarrow [inInsc.sender, inInsc.balance, st_{RHS}]$
10: **end procedure**

4.3.3.6 The Function Call Statement Block

1: **procedure** BUILDFUNCTIONCALLSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a function call statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create transition t^f
5: create place p_{param_f}
6: create arc from p_{in} to t^f
7: create arc from p_{param_f} to t^f
8: CONNECTLOCALVARIABLES$(f_{RHS}.vars;t;t^f)$
9: CONNECTFUNCTIONCALLS$(f_{RHS}.fctCalls;t)$
10: create arc from t^f to p_{out} with a placeholder inscription
11: **end procedure**

4.3.3.7 The Requirement Statement Block

Requirement statement $require(c)$: such a statement is transformed by creating two transitions t_{revert} and $t_{!revert}$, with $!c$ and c as respective guards, input links with p_{in} and input/output links with the places representing local variables used in c. Transition t_{revert} has $\bullet t[cf]$ as output place whereas p_{out} is the output place of $t_{!revert}$. We note that, as a requirement statement is often placed at the beginning of the function, p_{in} and $\bullet t[cf]$ are usually the same.

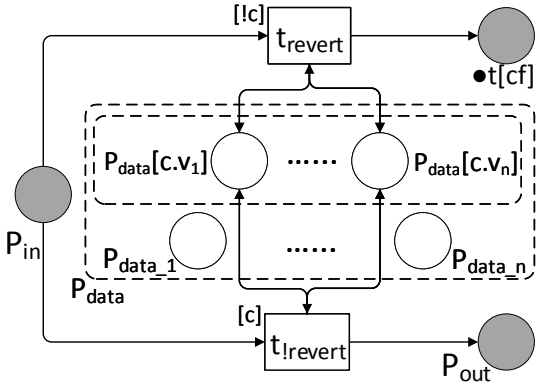

Figure 4.8: Requirement Statement Pattern

1: **procedure** BUILDREQUIREMENTSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a requirement statement $st = require(c)$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create transition t_{revert}
5: $t_{revert}.guard \leftarrow !c$

6: create arc from p_{in} to t_{revert}
7: create arc from t_{revert} to $\bullet t[cf]$
8: CONNECTLOCALVARIABLES*(c.vars;t;t_{revert})*
9: CONNECTFUNCTIONCALLS*(c.fctCalls;t_{revert})*
10: create transition $t_{!revert}$
11: $t_{!revert}.guard \leftarrow c$
12: create arc from p_{in} to $t_{!revert}$
13: create arc from $t_{!revert}$ to p_{out}
14: CONNECTLOCALVARIABLES*(c.vars;t;$t_{!revert}$)*
15: CONNECTFUNCTIONCALLS*(c.fctCalls;$t_{!revert}$)*
16: **end procedure**

4.3.3.8 The Selection Statement Block

Selection statement $if(c)$ *then* st_T [*else* st_F]: the algorithm creates two transitions, t_T with guard c and t_F with guard $!c$, that respectively represent the activation of the true and false (or default) branches of the selection statement. Both transition are linked to the input place p_{in} and any places representing local variables used in c. CREATESUBMODEL is then recursively called for the true-branch statement. In case of a one-branch selection, t_F is linked directly to the output place p_{out}. In case of a double-branch selection, a new place p_F is created to be the output place of t_F and the input place in a recursive call for CREATESUBMODEL on the false-branch statement.

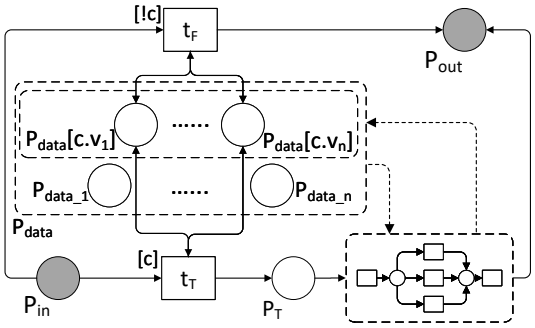

Figure 4.9: One-branch selection Statement Pattern

1: **procedure** BUILDSELECTIONSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a selection statement $st = if(c)$ *then* st_T [*else* st_F], control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create place p_T
5: create transition t_T
6: $t_T.guard \leftarrow c$

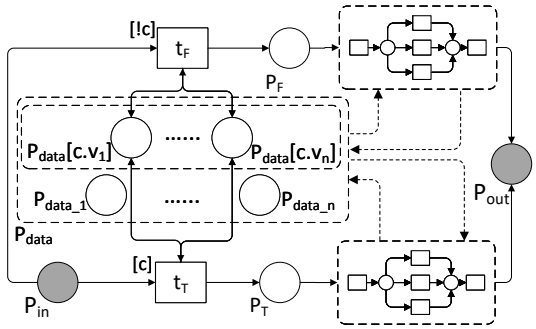

Figure 4.10: Double-branch selection Statement Pattern

7: create arc from p_{in} to t_T
8: create arc from t_T to p_T
9: connectLocalVariables*(c.vars;t;t_T)*
10: connectFunctionCalls*(c.fctCalls;t_T)*
11: createSubModel*(t;st_T;p_T;p_{out})*
12: create transition t_F
13: $t_F.guard \leftarrow !c$
14: create arc from p_{in} to t_F
15: connectLocalVariables*(c.vars;t;t_F)*
16: connectFunctionCalls*(c.fctCalls;t_F)*
17: **if** *st* is a selection statement: $if(c)\ then\ st_T$ **then**
18: create arc from t_F to p_{out}
19: **else if** *st* is a selection statement: $if(c)\ then\ st_T\ else\ st_F$ **then**
20: create place p_F
21: create arc from t_F to p_F
22: createSubModel*(t;st_F;p_F;p_{out})*
23: **end if**
24: **end procedure**

4.3.3.9 The For Loop Statement Block

For Looping statement $for(init; c; inc)\ st_T$: three places, p_{init}, p_c and p_T, are created to respectively represent initialization, increments and one pass of the loop's body. Transitions t_T with guard c and t_F with guard $!c$ are created and linked to the input place p_{in} and any places representing local variables used in c. t_T is linked to p_c as output to trigger a counter increment while t_F is linked to p_out as output to leave the loop. createSubModel is recursively called with three statements: *init* which is usually a variable declaration statement with initialization, st_T to develop the CPN snippet for one run of the loop, and *inc* which is usually an assignment to a local variable type of statement.

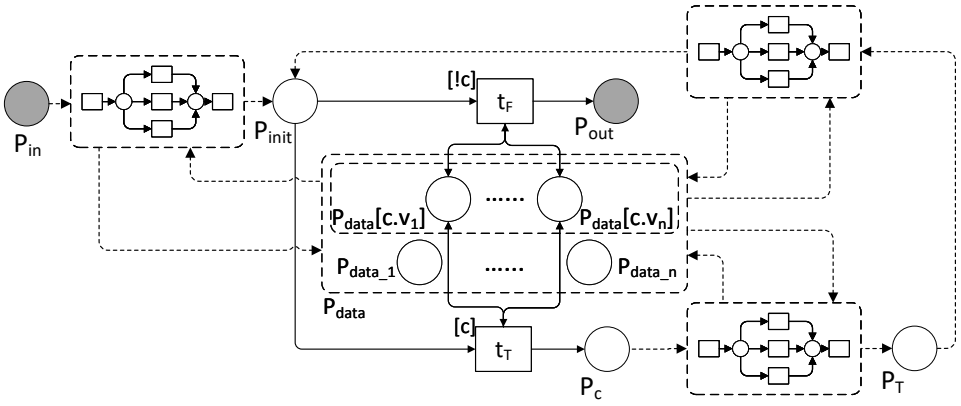

Figure 4.11: For Looping Statement Pattern

1: **procedure** BUILDFORLOOPSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a *for* looping statement $st = for(init; c; inc)\ st_T$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create place p_{init}
5: create place p_c
6: create place p_T
7: CREATESUBMODEL*(t;init;p_{in};p_{init})*
8: create transition t_T
9: $t_T.guard \leftarrow c$
10: create arc from p_{init} to t_T
11: CONNECTLOCALVARIABLES*(c.vars;t;t_T)*
12: CONNECTFUNCTIONCALLS*(c.fctCalls;t_T)*
13: create arc from t_T to p_c
14: create transition t_F
15: $t_F.guard \leftarrow !c$
16: create arc from p_{init} to t_F
17: CONNECTLOCALVARIABLES*(c.vars;t;t_F)*
18: CONNECTFUNCTIONCALLS*(c.fctCalls;t_F)*
19: create arc from t_F to p_{out}
20: CREATESUBMODEL*(t;st_T;p_c;p_T)*
21: CREATESUBMODEL*(t;inc;p_T;p_{init})*
22: **end procedure**

4.3.3.10 The While Loop Statement Block

While Looping statement $while(c)\ st_T$: for this loop the algorithm proceeds by creating one new place p_T which will be the output for a new transition t_T with guard c.

Another transition t_F is also created with $!c$ as guard and p_{out} as output place. Both transition are linked to the input place p_{in} and any places representing local variables used in c. A recursive call to createSubModel on st_T with p_T for input and p_{in} for output places is responsible for the generation of the loop's body corresponding CPN snippet.

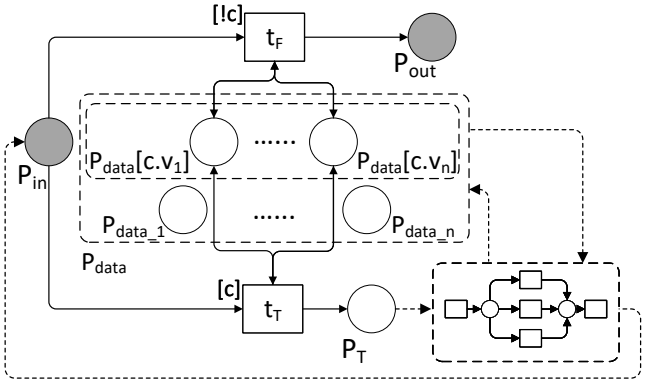

Figure 4.12: While Looping Statement Pattern

1: **procedure** buildWhileLoopStatement(t; st; p_{in}; p_{out})
2: **Input:** transition t, a *while* looping statement $st = while(c)\ st_T$ st_T, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** sub-model for statement st
4: create place p_T
5: create transition t_T
6: $t_T.guard \leftarrow c$
7: create arc from p_{in} to t_T
8: connectLocalVariables*(c.vars;t;t_T)*
9: connectFunctionCalls*(c.fctCalls;t_T)*
10: create arc from t_T to p_T
11: create transition t_F
12: $t_F.guard \leftarrow !c$
13: create arc from p_{in} to t_F
14: connectLocalVariables*(c.vars;t;t_F)*
15: connectFunctionCalls*(c.fctCalls;t_F)*
16: create arc from t_F to p_{out}
17: createSubModel*(t;st_T;p_T;p_{in})*
18: **end procedure**

4.3.4 Connecting the Data Places

1: **procedure** connectLocalVariables(V; t; t')

2: **Input:** set of local variables V, transition t, transition t'
3: **Output:** sub-model with connections to local variables
4: **for** $v \in V$ **do**
5: create arc from $t.data[v]$ to t'
6: create arc from t' to $t.data[v]$
7: **end for**
8: **end procedure**

4.3.5 Connecting the Function Calls

1: **procedure** CONNECTFUNCTIONCALLS(FC; t)
2: **Input:** set of function calls FC, transition t
3: **Output:** sub-model with connections to function calls
4: **for** $f \in FC$ **do**
5: create transition t^f
6: create place p_{return_f}
7: create place p_{param_f}
8: CONNECTLOCALVARIABLES($f_{RHS}.vars;t;t^f$)
9: create arc from p_{param_f} to t^f with inscription in which every element of f_{RHS} is replaced by its corresponding argument
10: create arc from t^f to p_{return_f} with a placeholder inscription
11: **end for**
12: **end procedure**

Example of Application

Figure 4.13 represents the resulting sub-model obtained by applying our transformation algorithms (initiated by the CREATESUBMODEL algorithm) on the function *withdraw()* of the *BlindAuction* contract (Listing 2.1 - Lines 31 to 42).

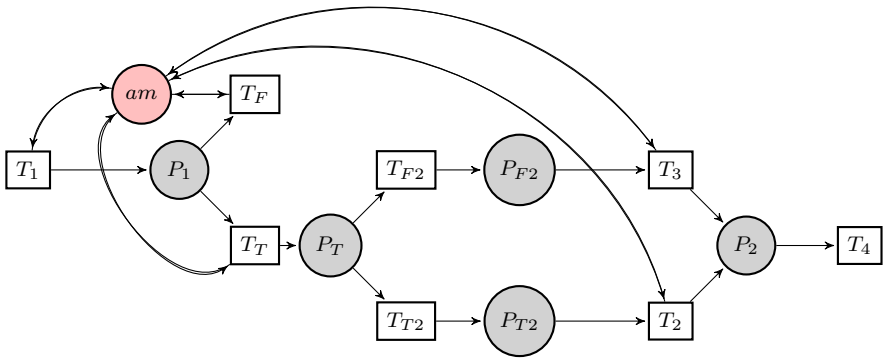

Figure 4.13: CPN sub-model of the *withdraw()* function in the *Blind Auction* contract

The BUILDCOMPOUNDSTATEMENT is first called on the body of the function, creating the places $P1$ and $P2$ and then recursively calling CREATESUBMODEL on all the statements it comprises. The algorithm corresponding to the statement's type is invoked each time, adding the necessary places and transitions in conformance to the defined patterns.

4.4 Conclusion

In this chapter, we achieved our objective ***Obj1*** mentioned in Section 1.3 which is providing a formal model for Solidity smart contracts. To do so, we have defined a CPN model for a Solidity smart contract by defining a CPN pattern for each statement type of the language. We have automated this transformation by providing a set of transformation algorithms that would generate the corresponding CPN sub-model according to the type of the statement they get as input and its corresponding CPN pattern. These CPN sub-models will be used as building blocks in the rest of our approach.

Chapter 5

Formal Modeling of Behavioural Contexts

Contents

5.1 Introduction

This chapter represents the second contribution of our work, which consists in the second step towards our end-goal approach (Figure 5.1). With this contribution we aim to achieve ***Obj2*** which is part of the response to ***RQ2*** and ***RQ4***.

We recall that the desired behavior of smart contracts can be intuitively depicted using a business process representation that would describe the context in which the smart contracts are intended to be used. Such a behavioral context may easily originate from a description of a business model. Herein we are interested in the generation of the context's CPN sub-model for such business models. We consider two types of behavioral context specifications for smart contracts:

1. *completely-free* if no information is provided on the execution context of a contract (Section 5.2)

2. *constrained* if the context in which a smart contract is used is provided (e.g., as a DCR Graph or a BPMN model)(Section 5.3).

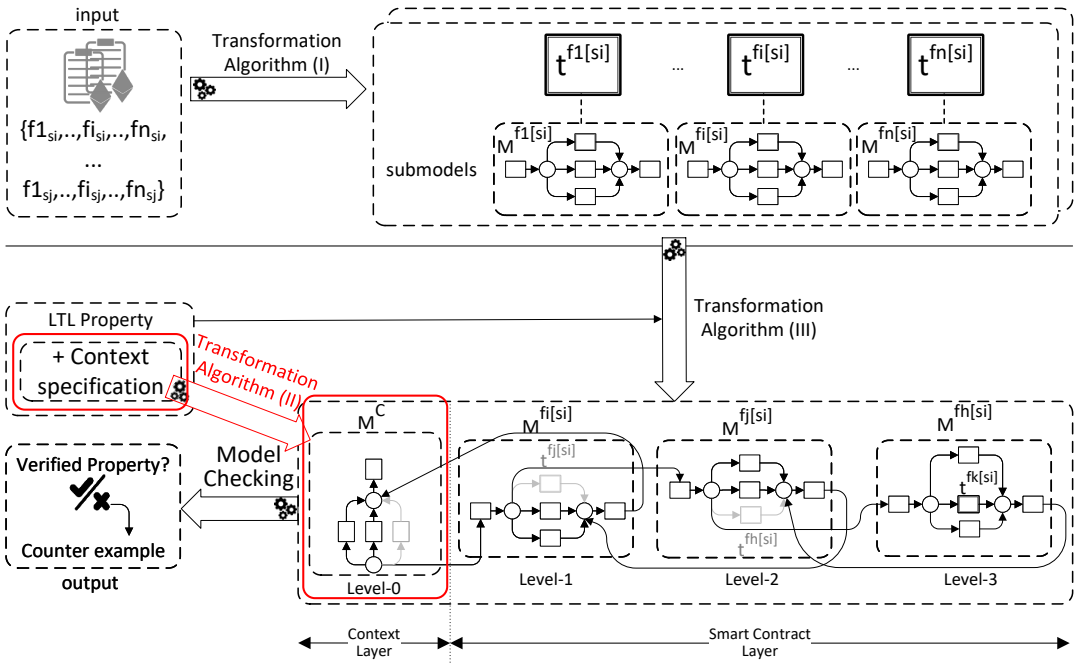

Figure 5.1: Approach overview - Step 2

5.2 Completely Free Behavioural Context

In case no behavioural context specification is provided with the smart contracts to be verified, we define a CPN model to represent the execution of these contracts in a completely-free way. In such a model (see Figure 5.2) a place S is used to represent the global state of the blockchain environment shared by all of the smart contracts' functions. For each function f_i a place P_i is used to represent its input parameters. The initial marking of a place P_i corresponds to all the possible calling arguments for f_i. In the following we give the formal definition of CPN4Free, the CPN model that we propose for the representation of a a free context.

Definition 5.2.1 (CPN4Free). *Given a set of smart contracts* $SSC = \{SC_i, \forall i \in [1,n]\}$, *such that* n *is the number of smart contracts to be verified and* $\forall SC_i \in SSC, SC_i = (f_{ji}, v_{hi}), \forall j \in [1,n_i], \forall h \in [1,m_i]$ *where* n_i *is the number of functions in* SC_i *and* m_i *is the number of global variables in* SC_i, *we denote by* $Param_{ji} = \{param^k_{ji}, \forall k \in [1,n_{ji}]\}$, *the set of input parameters of the function* f_{ji} *such that* n_{ji} *is the number of such parameters. A corresponding CPN model* $CPN4Free = (P,T,A,\Sigma,V,C,G,E,I)$ *(depicted in figure 5.2) is defined as follows:*

- $P = \{S\} \cup P_{param}$, *where* $P_{param} = \bigcup_{i\in[1,n]} P_i$, *such that* $\forall i \in [1,n], P_i = \{p_{ji}, \forall j \in [1,n_i]\}$

- $T = \bigcup_{i \in [1,n]} T_i$, *such that* $\forall i \in [1,n], T_i = \{t_{ji}, \forall j \in [1, n_i]\}$
- $A = \{(t_i, S), \forall t_i \in T\} \cup \{(S, t_i), \forall t_i \in T\} \cup \{(p_{ji}, t_{ji}), \forall p_{ji} \in P_i, \forall t_{ji} \in T_i, \forall i \in [1,n]\}$
- $\Sigma = \{C_S\} \cup \{C_{P_{ji}}, \forall i \in [1,n], \forall j \in [1, n_i]\}$, *where* $C_{P_{ji}} = [type_{param^1_{ji}} : param^1_{ji}, ..., type_{param^{n_{ji}}_{ji}} : param^{n_{ji}}_{ji}]$ *and* $C_S = [uint : contractBalance, type_{v_{11}} : v_{11}, ..., type_{v_{m_n n}} : v_{m_n n}] \times C_{P_{11}} \times ... \times C_{P_{m_n n}}$
- $V = \{x, x'\} \cup \{vp_{ji}, \forall i \in [1,n], \forall j \in [1, n_i]\}$, *with* $Type[x] = Type[x'] = C_S$ *and* $Type[vp_{ji}] = C_{P_{ji}}, \forall i \in [1,n], \forall j \in [1, n_i]$
- $C = \{S \to C_S\} \cup \{p_{ji} \to C_{P_{ji}}, \forall p_{ji} \in P_i, \forall i \in [1,n]\}$
- $G = \emptyset$
- $E = \{a \to x, \forall a \in A \cap (\{S\} \times T)\} \cup \{a \to x', \forall a \in A \cap (T \times \{S\})\} \cap E_{param}$, *where* $E_{param} = \bigcup_{i \in [1,n]} \{a \to p_{ji}, \forall j \in [1, n_i]\}$
- $I = \{p \to init_p, \forall p \in P\}$, *where* $init_p$ *is a predefined initialisation that depends on the type of the place.*

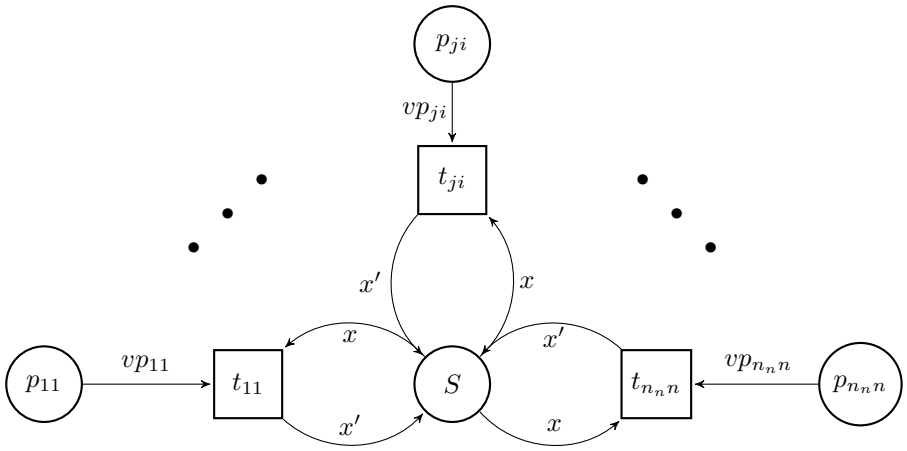

Figure 5.2: CPN model for a completely-free behavioural context

5.3 Constrained Behavioural Context - A CPN Model for DCR Choreographies

The behavior of smart contracts can be captured either imperatively or declaratively. Existing BPMN-to-CPN transformations [38, 69], as well as CPN models could be leveraged for an imperative representation, which is why in the following we will

focus on the integration of declarative representations, namely DCR graphs, into our work. The approach that we propose for the verification of smart contracts with a DCR graph/choreography as a behavioral context specification is based on the transformation of the latter into a CPN model that holds the same semantics.

To this end, we start by presenting, in Section 5.3.1 a CPN model for a DCR graph. This model being only adequate for the verification of state-based LTL properties, we propose an extension in Section 5.3.2, that can be seen as a generalization to the first model and that would allow the verification of both state- and event-based LTL properties. Finally, we present in Section 5.3.3 a second extension to our model that consists in a transformation of a DCR choreography by taking into account the concept of *roles*.

5.3.1 CPN4DCR - Initial Model

In the following we give the formal definition of the initial CPN4DCR model that we propose and showcase its capabilities and limitations for LTL model checking.

Definition 5.3.1 (CPN4DCR)**.** *Given a DCR graph* $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$*, a corresponding CPN model* $CPN4DCR = (P, T, A, \Sigma, V, C, G, E, I)$ *(depicted in figure 5.3) is defined as follows:*

- $P = \{S\}$
- $T = \{t_i, \forall i \in [1, n]\}$*, with* $n = |E|$ *the number of events in* G
- $A = \{(t_i, S), \forall t_i \in T\} \cup \{(S, t_i), \forall t_i \in T\}$
- $\Sigma = \{C_E, (C_E \times C_E \times C_E)\}$*, where* C_E *is a colour defined as an integer type* ($C_E = range\ INT$) *where each event* $e_i \in E$ *is represented in* C_E *by its index.*
- $V = \{Ex, Re, In, Ex', Re', In'\}$*, with* $Type[v] = C_E, \forall v \in V$
- $C = \{S \rightarrow (C_E \times C_E \times C_E)\}$
- $G = \{t_i \rightarrow guard_i, \forall i \in [1, n]\}$*, with* $n = |E|$
- $E = \{a \rightarrow \langle Ex, Re, In\rangle, \forall a \in A \cap (P \times T)\} \cup \{a \rightarrow \langle Ex', Re', In'\rangle, \forall a \in A \cap (T \times P)\}$ *with*
  - $Ex' = Ex \cup e_i$,
  - $Re' = (Re \backslash e_i) \cup e\bullet\rightarrow$ *and*
  - $In' = (In \cup e_i \rightarrow +) \backslash e \rightarrow \%$
- $I = \{S \rightarrow \langle S_1, S_2, S_3\rangle\}$ *with* $\langle S_1, S_2, S_3\rangle$ *the initial marking* M *of* G

For all $t_i \in T$ *representing an event* e_i *in the DCR graph, we further precise that:*

- *$guard_i$ is the conjunction of the conditions defining the enabling of the corresponding event e_i:*
  1. $i \in In$,
  2. $(\rightarrow\bullet i \cap In) \in Ex$ *and*
  3. $(\rightarrow\diamond i \cap In) \in E \backslash Re$
- *the expression $\langle Ex', Re', In' \rangle$ on its output arc is defined such that:*
  1. $Ex' = Ex \cup i$,
  2. $Re' = (Re \backslash i) \cup i\bullet\rightarrow$ *and*
  3. $In' = (In \cup i \rightarrow +) \backslash i \rightarrow \%$

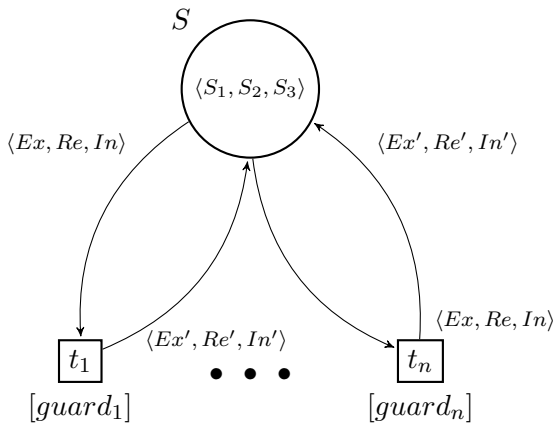

Figure 5.3: Initial CPN4DCR

The application of this definition on the DCR graph that would be obtained from the DCR graph example in figure 2.2 is shown in figure 5.4.

Definition 5.3.2 (Marking Equivalence)**.** *A marking $M^G = \langle Ex, Re, In \rangle$ of a DCR graph G is said to be equivalent to a marking $M^C = \langle S \rightarrow \langle S_1, S_2, S_3 \rangle \rangle$ of a CPN model C iff*

- $\forall e_i \in Ex$ *(respectively Re and In)*, $\exists i \in S_1$ *(respectively S_2 and S_3), and*
- $\forall i \in S_1$ *(respectively S_2 and S_3)*, $\exists e_i \in Ex$ *(respectively Re and In)*

We note $M^G \equiv M^C$.

Definition 5.3.3 (Execution Sequence Equivalence)**.** *An execution sequence of length k, $\sigma_k^G = \langle e_i, ..., e_j \rangle$ of a DCR graph G is said to be equivalent to an execution sequence of length k, $\sigma_k^C = \langle t_i, ..., t_j \rangle$ of a CPN model C iff*

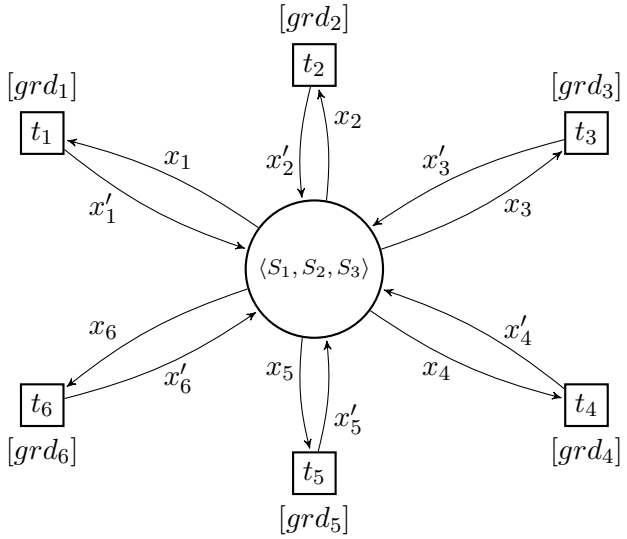

Figure 5.4: Initial CPN4DCR for the Blind Auction example

- $(M_1^G \equiv M_1^C \wedge M_1^G \xrightarrow{\sigma_k^G} M_2^G \wedge M_1^C \xrightarrow{\sigma_k^C} M_2^C) \implies M_2^G \equiv M_2^C$

We note $\sigma_k^G \equiv \sigma_k^C$.

Proposition 5.3.1. *Let G be a DCR graph and C the corresponding CPN model generated as per definition 5.3.1, then G and C are semantically equivalent.*

Proof. Let G be a DCR graph and C the corresponding CPN model generated as per definition 5.3.1. In order to prove that G and C are semantically equivalent we need to prove that

1. $\forall \sigma_k^G = \langle e_1, ..., e_k \rangle, \exists \sigma_k^C = \langle t_1, ..., t_k \rangle$, and
2. $\forall \sigma_k^C = \langle t_1, ..., t_k \rangle, \exists \sigma_k^G = \langle e_1, ..., e_k \rangle$

such that $\sigma_k^G \equiv \sigma_k^C$, $\forall k \in [1, m]$ with m the length of the longest execution sequence. We start by proving *(1)*:

- Let $P(n)$ be the statement: $\forall \sigma_n^G = \langle e_1, ..., e_n \rangle, \exists \sigma_n^C = \langle t_1, ..., t_n \rangle$ such that $\sigma_n^G \equiv \sigma_n^C$.
- $P(1) : \forall \sigma_1^G = \langle e_1 \rangle, \exists \sigma_1^C = \langle t_1 \rangle$ such that $\sigma_1^G \equiv \sigma_1^C$. This can be derived from Definition 5.3.1. In fact, the initial marking of C (M_0^C) being defined as equivalent to that of G (M_0^G), and the guard of each transition $t_i \in T$ being defined as to correspond to the enabling conditions of the relative event $e_i \in E$, we can deduce that the set of fireable transitions ($M_0^C \rightarrow$) corresponds to the set

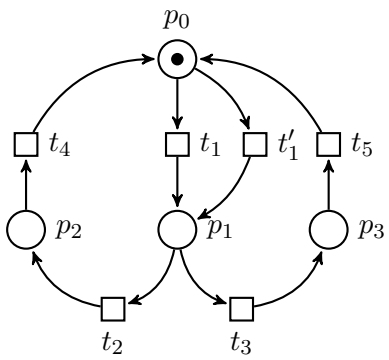

Figure 5.5: State- vs event-based LTL property example

of enabled events ($M_0^G \rightarrow$). Additionally, the marking M_i^C obtained by firing t_i is equivalent to that obtained by executing e_i ($M_i^C \equiv M_i^G$) since the elements of M_i^C are defined as to correspond to the effect of the execution of e_i in G.

- Assume that $P(k) : \forall \sigma_k^G = \langle e_1, ..., e_k \rangle, \exists \sigma_k^C = \langle t_1, ..., t_k \rangle$ such that $\sigma_k^G \equiv \sigma_k^C$ is true for some $k \in [2, m-1]$. We will prove that $P(k+1) : \forall \sigma_{k+1}^G = \langle e_1, ..., e_{k+1} \rangle, \exists \sigma_{k+1}^C = \langle t_1, ..., t_{k+1} \rangle$ such that $\sigma_{k+1}^G \equiv \sigma_{k+1}^C$ is true.

$$\sigma_{k+1}^G \equiv \sigma_{k+1}^C \implies \exists e_{k+1} \in E, t_{k+1} \in T \text{ such that } \sigma_k^G \cdot e_{k+1} \equiv \sigma_k^C \cdot t_{k+1} \tag{5.1}$$

$$\sigma_k^G \equiv \sigma_k^C \iff (M_0^G \xrightarrow{\sigma_k^G} M_k^G \wedge M_0^C \xrightarrow{\sigma_k^C} M_k^C \wedge M_k^G \equiv M_k^C) \tag{5.2}$$

Analogously to the reasoning in the previous point, we can deduce that:

$$\begin{aligned} &\forall e_{k+1} \in E \text{ such that } M_k^G \xrightarrow{e_{k+1}} M_{k+1}^G, \\ &\exists t_{k+1} \in T \text{ such that } (M_k^C \xrightarrow{t_{k+1}} M_{k+1}^C \wedge M_{k+1}^G \equiv M_{k+1}^C) \end{aligned} \tag{5.3}$$

And therefore:

$$\forall \sigma_{k+1}^G = \langle e_1, ..., e_{k+1} \rangle, \exists \sigma_{k+1}^C = \langle t_1, ..., t_{k+1} \rangle \text{ such that } \sigma_{k+1}^G \equiv \sigma_{k+1}^C \tag{5.4}$$

The second part *(2)* is provable following a similar reasoning. □

5.3.1.1 Discussion

It is evident to see that a DCR graph is a model based on *events* or *tasks*, which in our corresponding CPN model we represent by transitions. It is therefore sensible to assume that the users would want to formulate the properties to be verified based on those tasks. Such properties can refer to *(i)* the execution of an event, which would translate into a property about the firing of its corresponding transition in the CPN

model, or to *(ii)* the possibility of executing an event, which would translate into a property about the fireability of its corresponding transition. The nuance between the two cases might seem subtle, but the difference is rather plain when it comes to their expression in LTL. To explain this, we will refer to the simple example in figure 5.5. In fact, it would be easy to express a property on the firing of a transition (e.g., if transition t_1 is fired, transition t_2 *will eventually* be fired in the future) using event-based LTL. Such a property would be, in that case, simply expressed as follows: $\mathcal{G}(t_1 \implies \mathcal{F}t_2)$. Expressing the same property using state-based LTL cannot be accomplished in such a straightforward and intuitive way. In general, to express the firing of a transition in state-based LTL we would have to resort to *markings*. Thus, "a transition t is fired" is expressed by the fact that we reach a marking m' that is the immediate successor of a marking m and where the difference between the two markings corresponds to the effects of firing t. More rigorously, t is fired if $\exists m'$ s.t $m \rightarrow m'$ with $m \wedge \mathcal{X}m - pre(t) + post(t)$. On the other hand, it can be intuitive to express properties on the fireability of a transition (e.g., transition t_2 *may* be fired in the future) using state-based LTL as follows: $\mathcal{G}(\mathcal{F}p_1)$. In general, the fireability of a transition is expressed as having the conditions for its firing satisfied and therefore as a property on its input places. Trying to express the same property in event-based LTL would result in a formula on the transitions that need to be fired for the transition in question to be fireable which can be very hard to do in complex models as it is a counter-intuitive reasoning. To be able to express both kinds of properties, we would ideally have a model checker that supports both event-based and state-based LTL formulae. In reality, model checkers only support one kind of LTL properties. To get around this technical limitation, we propose a second CPN model for DCR graphs that allows to easily and efficiently express event-based properties as state-based ones (without having to resort to markings). Consequently, we allow the users to express both kinds of properties on the events of their DCR graph and support their verification without forcing the users to express them in a counter-intuitive way.

5.3.2 CPN4DCR - Generalization for event-based properties

If we go back to the example in figure 5.5, we note that, if we remove the transition t'_1, we can find a much simpler state-based LTL property to express the event-based one that we had given as example ($\mathcal{G}(t_1 \implies \mathcal{F}t_2)$). In fact, we would not have to resort to expressing a property on the markings of the model, and the following expression: $\mathcal{G}(p_1 \implies \mathcal{F}p_2)$ would be enough to express an equivalent state-based property to the former one. This is mainly due to the fact that by removing the transition t'_1, the place p_1 can only be marked by the firing of t_1, and therefore having a token in that place forcibly implies that t_1 has been fired.
The idea behind our second model is inspired by this particular case, and it consists mainly in introducing a set of additional places P^T to the first proposed model, such that we would have one and only one place $p^T \in P^T$ marked when a transition

(corresponding to an event in the DCR graph) is fired. We also add a set of *fake* transitions T^F whose role is solely to connect the added places to the main place S of our first model. The definition of our second model can therefore be given as follows:

Definition 5.3.4 (CPN4DCR2)**.** *Given a DCR graph* $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ *and its corresponding initial CPN4DCR model* $CPN4DCR = (P, T, A, \Sigma, V, C, G, E, I)$ *as defined in definition 5.3.1, an event-oriented generalization CPN4DCR model* $G\_CPN4DCR = (P', T', A', \Sigma', V', C', G', E', I')$ *(depicted in figure 5.6) is defined s.t.:*

- $P' = P \cup P^T$
- $P^T = \{p_i^T, \forall i \in [1, n]\}$*, with* $n = |E|$ *the number of events in* G
- $T' = T \cup T^F$
- $T^F = \{t_i^F, \forall i \in [1, n]\}$*, with* $n = |E|$ *the number of events in* G
- $A' = \{(S, t_i), \forall t_i \in T\} \cup \{(t_i, p_i^T), \forall t_i \in T$ *and* $p_i^T \in P^T\} \cup \{(p_i^T, t_i^F), \forall p_i^T \in P^T$ *and* $t_i^F \in T^F\} \cup \{(t_i^F, S), \forall t_i^F \in T^F\}$
- $\Sigma' = \Sigma$
- $V' = V \cup \{X\}$*, with* $Type[X] = (C_E \times C_E \times C_E)$
- $C' = C \cup \{p_{T_i} \rightarrow (C_E \times C_E \times C_E), \forall p_i^T \in P^T\}$
- $G' = G$
- $E' = \{a \rightarrow \langle Ex, Re, In\rangle, \forall a \in A \cap (\{S\} \times T)\} \cup \{a \rightarrow \langle Ex', Re', In'\rangle, \forall a \in A \cap (T \times P^T)\} \cup \{a \rightarrow X, \forall a \in A \cap (P^T \times T^F) \cup (T^F, \{S\})\}$ *with (1)* $Ex' = Ex \cup e_i$*, (2)* $Re' = (Re \backslash e_i) \cup e\bullet\rightarrow$ *and (3)* $In' = (In \cup e_i \rightarrow\!\!+) \backslash e \rightarrow\!\!\%$
- $I' = I$

5.3.2.1 Optimization of the model

Adding *fake* transitions and places to our CPN model would indeed increase the size of its state space since we are basically creating intermediate states between the original states from the initial model. This would affect the performance of the model checker that's supposed to take in charge the verification of the model. In order to minimize the number of extra states to be created, we only create *fake* transitions for the events involved in the property to be verified. For instance, lets consider the following property on the *Blind Auction* example in figure 2.2: $prop_{DCR} : \mathcal{G}(\neg RevealCall \vee \neg \mathcal{F} BidCall)$ which basically expresses the fact that no bids can be placed once the revealing window is opened (i.e., the first revealing bids call happened). The application of this definition on this DCR graph with the aim of verifying the property $prop_{DCR}$ is shown in figure 5.7.

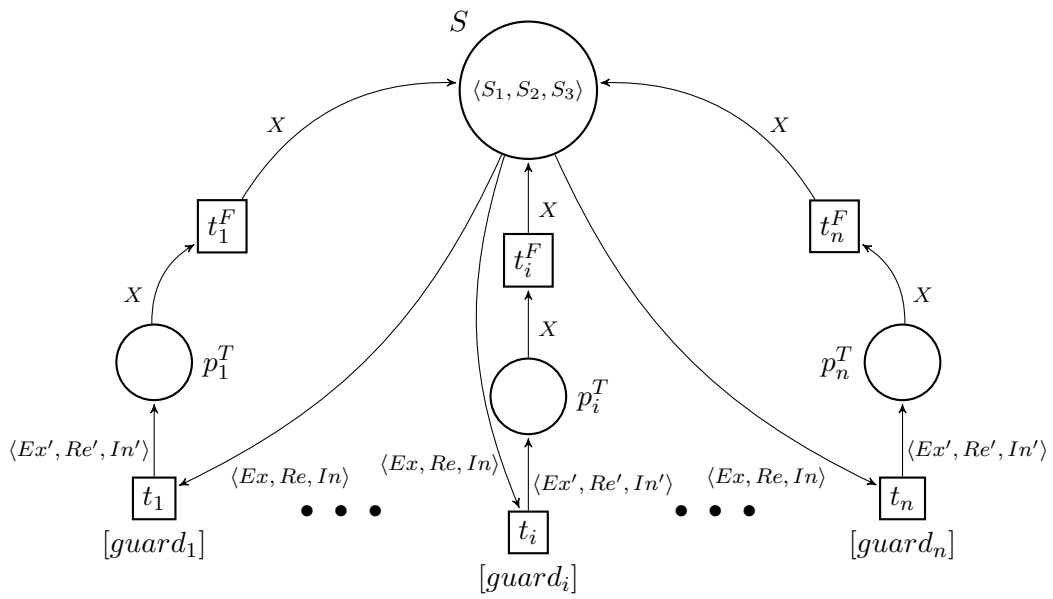

Figure 5.6: Generalized CPN4DCR

5.3.3 CPN4DCR - Extension for DCR choreographies

In order to be able to verify properties on DCR choreographies, we need to take into account the concept of *roles* in our CPN4DCR model. To do so, we mainly add a place R, initially containing the set of $\langle initiator, receivers \rangle$ in the DCR choreography model, linked to all transitions representing events.

Definition 5.3.5 (CPN4DCR3)**.** *Given a DCR choreography* (G, R) *where* $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$*, the corresponding initial CPN4DCR model for* G*:* $CPN4DCR = (P, T, A, \Sigma, V, C, G, E, I)$ *as defined in definition 5.3.1, and its generalization* $G\_CPN4DCR = (P', T', A', \Sigma', V', C', G', E', I')$*, the extended CPN4DCR model for choreographies* $E\_CPN4DCR = (P'', T'', A'', \Sigma'', V'', C'', G'', E'', I'')$ *(depicted in figure 5.8) is defined s.t.:*

- $P'' = P' \cup \{R\}$
- $T'' = T'$
- $A'' = A' \cup \{(R, t_i), \forall t_i \in T\} \cup \{(t_i, R), \forall t_i \in T\}$
- $\Sigma'' = \Sigma \cup \{C_R, (C_R \times \mathcal{P}(C_R))\}$*, where* C_R *is a colour defined as a string type* $(C_R = STRING)$ *to represent* roles *in DCR*
- $V'' = V' \cup \{V_R, V_{SR}\}$*, with* $Type[V_R] = C_R$ *and* $Type[V_{SR}] = \mathcal{P}(C_R)$
- $C'' = C' \cup \{R \rightarrow (C_R \times \mathcal{P}(C_R))\}$

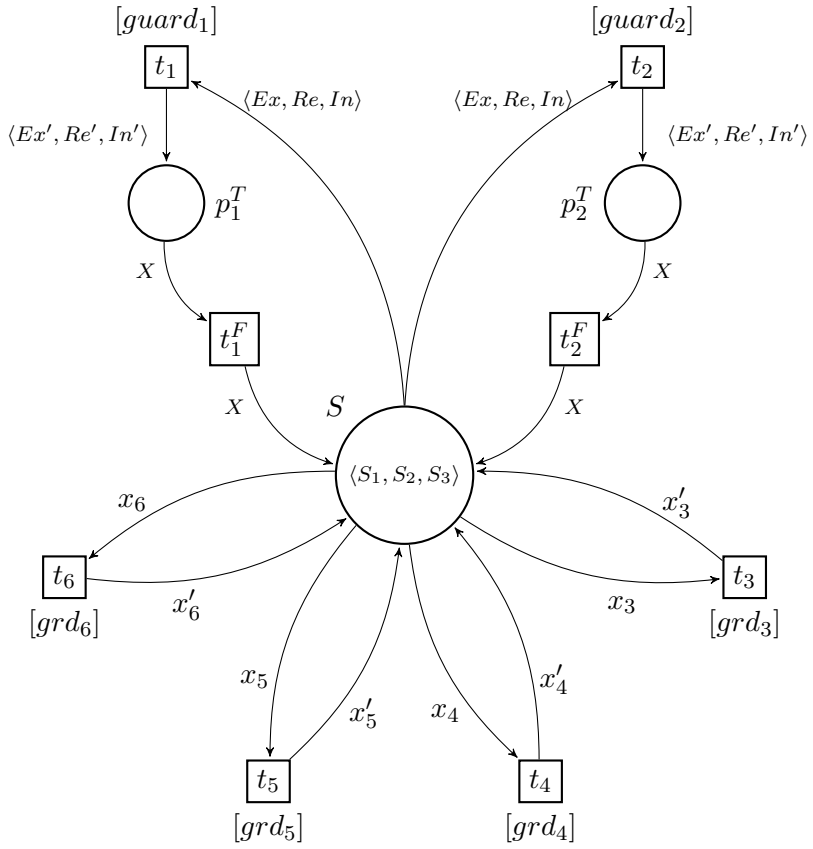

Figure 5.7: Generalized CPN4DCR of the Blind Auction example

- $G'' = G'$
- $E'' = E' \cup \{a \rightarrow \langle V_R, V_{SR}\rangle, \forall a \in A'' \cap (\{R\} \times T) \cup (T \times \{R\})\}$
- $I'' = I' \cup \{R \rightarrow \{r_i, \forall i \in [1..k]\}$, *where each* r_i *being a token representing the initiator ans potential receivers of an event* $e \in E$ *with* $k = |\bigcup_{j=1}^{|E|} l(e_j)|$.

we further precise that $\forall t_i \in T$ *representing an event* e_i *in the DCR graph, the expression* $\langle V_R, V_{SR}\rangle$ *on the arcs connecting* t_i *to* R *is defined such that* $\langle V_R, V_{SR}\rangle = l(e_i)$.

The application of this definition on the DCR choreography in figure 2.3 with the same property as in the previous example in Figure 5.7 gives the CPN model shown in figure 5.9.

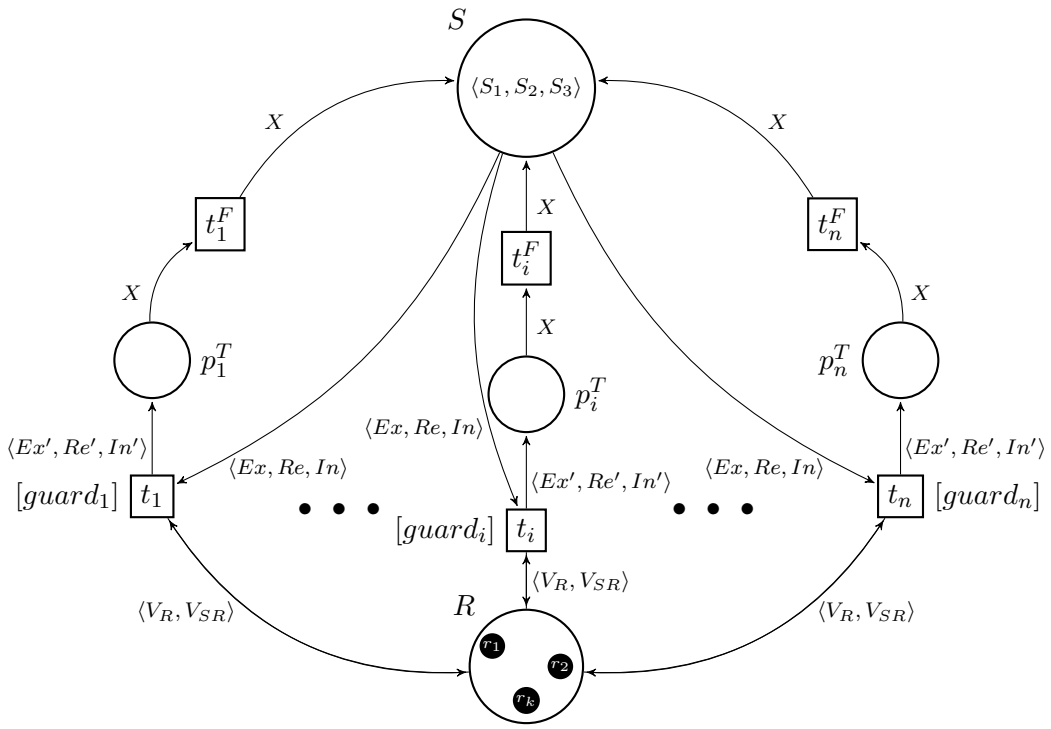

Figure 5.8: Extended CPN4DCR for choreographies

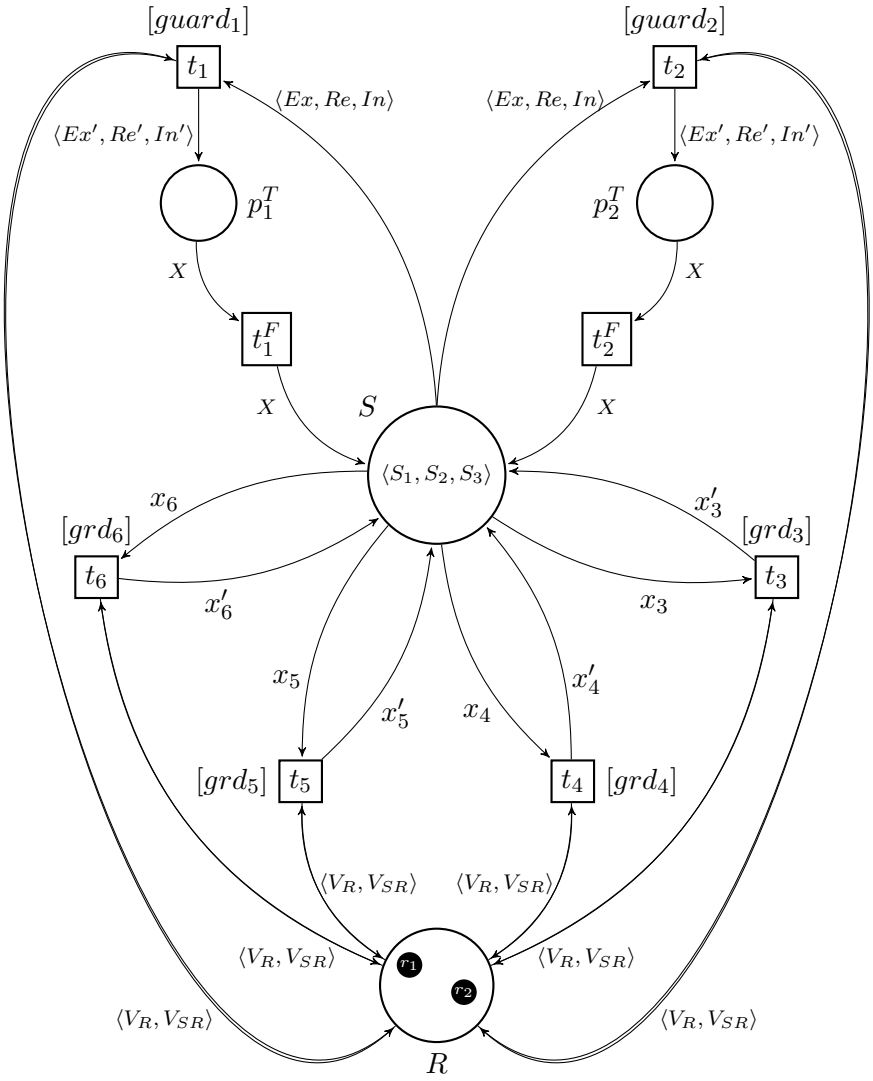

Figure 5.9: Extended CPN4DCR of the Blind Auction example

5.4 Conclusion

In this chapter, we achieved our objective ***Obj2*** mentioned in Section 1.3 which is providing a formal model for possible behavioural contexts that may accompany the smart contracts. To do so, we have mainly considered two types of behavioural contexts. The first is what we called *completely free* context, and which is used in case of absence of a behavioural context for the smart contracts in question. In other words, if the users have no idea on how they will be using their smart contracts, we will consider this type of context and accordingly build a CPN model (CPN4Free) to represent a free behaviour based only on the elements of the smart contracts to be verified. The second type of context is what we called *constrained* context, which is used when the users have restrictions on the way they intend to use their smart contracts and can define this behaviour in the form of a DCR graph or choreography. In this case, we define a CPN model (CPN4DCR) that would be built on the elements of the smart contracts according to the provided DCR model. Regardless of the type of context, by the end of this step we have a CPN model that will represent the level-0 model in the final HCPN to be generated as will be further explained in the following chapters.

Chapter 6

Expression of Properties to be Verified using Linear Temporal Logic

Contents

6.1 Introduction

This chapter represents the third contribution of our work, which consists in the third step towards our end-goal approach (Figure 6.1). With this contribution we aim to achieve ***Obj3*** which is part of the response to ***RQ5*** and ***RQ6***. Herein we are interested in the expression of the properties to be verified on the Solidity smart contracts in question. In our PhD work, we essentially aim to propose an approach

that allows the verification of contract-specific properties on smart contracts as it is important to be able to specify the correctness of smart contracts depending on their intended application, especially (but not only) if their behavioural context is known. This does not, however, mean that the detection of vulnerabilities is irrelevant to our work. For this reason, we propose an LTL formalization of six common Solidity vulnerabilities to showcase the capability of our approach to be used to incidentally detect the presence of vulnerabilities in Solidity smart contracts.

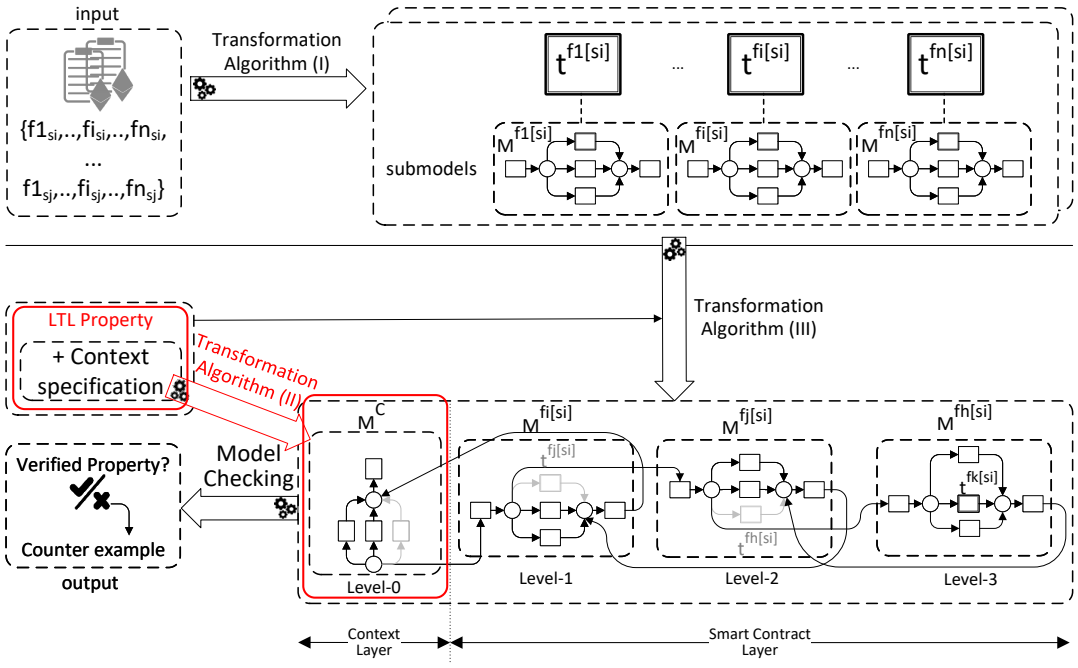

Figure 6.1: Approach overview - Step 3

6.2 Vulnerabilities Through Examples

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but they are actually dissimilar since the underlying semantics of Solidity is different from traditional programs. This naturally calls on more vigilance from programmers who might be faced by unconventional security issues as vulnerabilities in smart contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer [24].

In the following we present two use cases to explain the vulnerabilities treated in this work to provide a better understanding of how we verify them later.

One of the most widespread smart contract applications is delivering gambling services. In fact, thanks to Blockchain's decentralized nature and the transparency of

transactions within it, players can have a clear view of the behaviour of the game and therefore are led and incentivized to put their trust in the system which is determined by the rules implemented by its smart contracts.

Our first Solidity example (Listing 6.1) is based on a published contract[1] implementing a lottery game. It has been tweaked to illustrate more vulnerabilities without altering its purpose. A player participates in this game by sending an amount of ether equal to the *TICKET_AMOUNT* through *playTicket()*, which is then added to the game's *pot*. The winner is determined based on a *random* value calculated using the block's timestamp and the *LottoLog* is updated accordingly to keep track of the winners. The winner then gets paid by calling *getPot()* and the game's host (*bank*) can start a new round of lotto using *RestartLotto()*. This contract may seem fair to inexperienced Solidity developers, but it actually presents multiple vulnerabilities as we will later explain.

```
1  contract EtherLotto {
2      address public bank;
3      struct GameRecord {
4        address winner;
5        uint amount;
6      }
7      uint8 gameNum;
8      GameRecord[] LottoLog;
9      bool won;
10     uint constant TICKET_AMOUNT = 10;
11     uint constant FEE_AMOUNT = 1;
12     uint public pot;
13     function EtherLotto() {
14         bank = msg.sender;
15         won = false;
16         gameNum = 0;
17     }
18     function RestartLotto() {
19       require(msg.sender == bank);
20       require(won == true);
21       require(pot == 0);
22       won = false;
23       gameNum += 1;
24     }
25     function playTicket() payable {
26         require(msg.value == TICKET_AMOUNT);
27         require(won == false)
28         pot += msg.value;
29         uint random = uint(sha3(block.timestamp)) % 2;
30         if (random == 0) {
31             bank.call.value(FEE_AMOUNT)("");
32             won = true;
33             GameRecord gr;
34             gr.winner = msg.sender;
```

[1] https://etherscan.io/address/0xa11e4ed59dc94e69612f3111942626ed513cb172

```
            gr.amount = pot - FEE_AMOUNT;
            LottoLog[gameNum] = gr;
        }
    }
    function getPot() {
        require(won == true);
        if(msg.sender == LottoLog[gameNum].winner){
            msg.sender.call.value(LottoLog[gameNum].amount)("");
            pot = 0;
        }
    }
}
```

Listing 6.1: Solidity example: EtherLotto.sol

```
contract MaliciousContract {
    uint ticket;
    EtherLotto el = EtherLotto(0xbf0061dc...);
    EtherMilestone em = EtherMilestone(0xc50164dfa...);
    function playLotto() {
        ticket = msg.value;
        el.playTicket.value(ticket)();
        el.getPot();
    }
    function playMilestone() {
        em.play.value(1)();
    }
    function getRevenge ( ) {
        selfdestruct(em);
    }
    function () payable {
        el.getPot();
    }
  }
```

Listing 6.2: A malicious smart contract in Solidity

We consider a second Solidity example[2] (Listing 6.3) to emphasize on the harmful effect the self-destruction vulnerability (see next subsection) can have on a contract. It implements another gambling game whereby a player sends 1 ether to the contract by calling *play()* in hopes to be the one to hit a milestone. Once the game is over (i.e., the *finalMileStone* is reached) winners can claim their rewards through *claimReward()*.

```
contract EtherMilestone {
    uint public payoutMileStone1 = 6 ether;
    uint public mileStone1Reward = 4 ether;
    uint public payoutMileStone2 = 10 ether;
    uint public mileStone2Reward = 6 ether;
    uint public finalMileStone = 20 ether;
    uint public finalReward = 10 ether;
```

[2] https://gist.github.com/vasa-develop/415a17c709d804a4d351485cd1b7c981

```
8     mapping(address => uint) redeemableEther;
9     function play() public payable {
10        require(msg.value == 1 ether);
11        uint currentBalance = this.balance + msg.value;
12        require(currentBalance <= finalMileStone);
13        if (currentBalance == payoutMileStone1) {
14            redeemableEther[msg.sender] += mileStone1Reward;
15        }
16        else if (currentBalance == payoutMileStone2) {
17            redeemableEther[msg.sender] += mileStone2Reward;
18        }
19        else if (currentBalance == finalMileStone ) {
20            redeemableEther[msg.sender] += finalReward;
21        }
22        return;
23    }
24    function claimReward() public {
25        require(this.balance == finalMileStone);
26        require(redeemableEther[msg.sender] > 0);
27        redeemableEther[msg.sender] = 0;
28        msg.sender.call.value(redeemableEther[msg.sender])("");
29    }
30 }
```

Listing 6.3: Solidity example: EtherMilestone.sol

6.2.1 Integer Overflow/Underflow:

Due to Solidity's lack of safeguards on mathematical operators, errors such as overflows and underflows may occur as a result of violation of value limitations of integer data types. For instance, the *uint8 gameNum* variable in the *EtherLotto* contract can be the source of such a vulnerability when the game exceeds 256 rounds. In fact, at the 257^{th} round, and due to Solidity's wrapping in two's complement representation for integers, *gameNum* will be set back to 0, causing data errors/overwriting into the critical *LottoLog* variable.

6.2.2 Reentrancy:

This is by far the most notorious vulnerability since it led to the infamous DAO attack. An attack of this type can take several forms (e.g, we can talk about a single function reentrancy attack or a cross-function reentrancy attack), but the main idea behind it is that a function can be interrupted in the middle of its execution and then be safely called again before its initial call completes. Once the second call completes, the initial one resumes correct execution. The simplest example is when a smart contract uses a variable to keep track of balances and offers a withdraw function. A vulnerable contract would make a transfer of funds prior to updating the corresponding balance which an attacker can take advantage of by recursively

calling this function and eventually draining the contract. This can be illustrated by a call to the function *playLotto()* with a value of 10 in the *MaliciousContract* (Listing 6.2) which would start by playing a ticket in the *EtherLotto* contract by invoking its *playTicket()* function and then attempting its *getPot()* function. In the instance where attacker's ticket is a winning one and the contract holds more than twice the amount of the *pot* in that round, a reentrancy attack can happen. In fact, by sending the jackpot to the winner (line 17 in Listing 6.1), the *EtherLotto* contract invokes the *fallback* function of the *MaliciousContract*, which is an unnamed function used to receive data or Ether. This is where the control flow is handed over to the latter contract whose *fallback* function recursively calls *getPot()*, which is allowed since the conditions on its execution are still valid, until the *EtherLotto* contract's balance is less than the current *pot*'s amount.

6.2.3 Self-Destruction:

The *selfdestruct(address)* function, when implemented in a contract, removes all bytecode from the contract's address to render it inaccessible and sends all its ether to the specified *address*. The latter can be another contract's address, in which case, the ether transfer happens forcibly, regardless of the recipient's code (i.e., without invoking its *fallback* function). Getting back to our second example *EtherMilestone*, we note the use of *this.balance* in lines 8 and 16. A player who missed a milestone, could vengefully send an amount of ether using *selfdestruct()* (e.g., function *getRevenge()* in *MaliciousContract*) as to push the contract's balance above the *finalMileStone*, locking all of the contract's ether and denying the winners who had already reached some milestones their rewards since *claimReward()* would revert.

6.2.4 Timestamp dependence:

Since the execution on a Blockchain needs to be deterministic for all the miners to get the same results and reach a consensus, users usually resort to block-related variables such as timestamp as a source of entropy. Sharing the same view on the Blockchain, miners would generate the same result, albeit being unpredictable. Even though this seems to be safe, it gives the miners a small room for manipulation given that they can choose a timestamp within a certain range for the new block, which gives them the possibility to tamper with the results and put some bias towards a certain user for example. Such a vulnerability can be exploited by any contract relying on a time constraint to determine its course of action. In our *EtherLotto* example, the function *playTicket()* is timestamp-dependent.

6.2.5 Skip Empty Literal:

The source of this vulnerability is the way the encoder of the Solidity compiler treats the arguments in a function call. In fact, when a function call's argument is an empty

string literal, it affects the following arguments which are shifted to the right by 32 bytes. This results in a function call with corrupted data.

6.2.6 Uninitialized Storage Variable:

Solidity stores state variables sequentially. So in *EtherLotto*, the variable *bank* is stored in slot 0. Since Solidity uses storage for complex data types like structs by default when declared as local variables, they become pointers to storage. Because *gr* is uninitialized (line 12 in Listing 6.1), it would actually point to the same slot as *bank*. When setting *gr.winner* to the first winner's address, this is effectively changing the address stored in *bank* to the winner's, which results in an unexpected behaviour by this contract. In our example, we present this vulnerability as an error unintentionally introduced by the contract's owner and unintentionally exploited by the first winner. It can, however, be intentionally injected in a contract's code or intentionally exploited by a user, as is the case in the *OpenAddressLottery*[3] honeypot.

6.3 Expressing Vulnerabilities in LTL

In the following, $M_{s_i}^{f}$ designates the CPN sub-model corresponding to function f in smart contract s_i. We note that sometimes we use parameterized propositions to indicate that they are applied to an unspecified aggregated transition. Concretely, such propositions need to be explicitly defined for each transition to be verified[4].

6.3.1 Integer Overflow/Underflow

In our CPN model, we define correspondences between the types used in the Solidity language and those offered by *helena* so that they cover the same ranges. The model checker is therefore able to detect when the smart contract contains an out-of-range expression. It does not, however, pinpoint the source of the anomaly, so the user does not have much information to go on to track it and try to correct it. To overcome this deficiency, we propose to model integer overflows/underflows as a safety LTL property that can be verified on a specific variable x:

$$IUO_x = \Box\neg xIsOutOfRange$$

Where *xIsOutOfRange* is a proposition that evaluates to true if the value of x is not included in the range of its type which we delimit by defining lower and higher thresholds (*minThreshold* and *maxThreshold* respectively).

$$\begin{aligned} xIsOutOfRange = &(x < minThreshold) \\ &\vee (x > maxThreshold) \end{aligned}$$

[3] https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9ed18902

[4] Not to be confused with first order predicates

6.3.2 Reentrancy

This vulnerability is related to functions that contain instructions responsible for Ether transfer, and therefore is applied w.r.t a function containing a *sending* statement. Given such a function, we propose two LTL properties. The first is a safety property defined as follows:

$$Reentrancy_{M^f_{s_i}} = \Box\neg reentrant_{M^f_{s_i}}$$

Where $reentrant_{M^f_{s_i}}$ is true if the necessary condition under which a reentrancy vulnerability can be detected in the function f in the smart contract s_i is valid. This condition can only be defined when the user indicates the variable x serving as a record for balances and whose assignment should be watched. Such a condition expresses the presence of a *sending* statement which is not preceded by an assignment to x:

$$reentrant_{M^f_{s_i}} = (\neg XAssignment)\mathcal{U}\ Sending$$

Where $XAssignment$ is true when a statement is an assignment to the variable x and $Sending$ is true when a statement is a sending one. A vulnerability is detected when $Reentrancy(t^f_{s_i})$ evaluates to false. This property is used when we only have the code of the smart contract to be verified (i.e., a totally free behaviour). If the code of the interacting contract s_j is available, we propose the following LTL property:

$$\begin{aligned} Reentrancy_{M^f_{s_i}} =& SendingTo_{s_j} \rightarrow \bigcirc\Box((\neg SendingTo_{s_j}) \\ & \mathcal{U}\ endOfFallback_{s_j}) \end{aligned}$$

Using this property we can verify that once the sending statement is executed ($SendingTo_{s_j}$ is true), it cannot be executed again until the fallback function of the receiving contract has finished ($endOfFallback_{s_j}$ is true) i.e., no reentrancy breach can happen.

6.3.3 Self-destruction

It is checked by detecting the presence of a test containing *this.balance* in the code of the function:

$$selfDestruction_{M^f_{s_i}} = \neg testOnBalance_{M^f_{s_i}}$$

This detection process can be further enhanced when the code of the interacting smart contract is available. In that case, given a function g in s_j that contains a self destruction instruction directing Ether to s_i, a function f in s_i is considered safe against this vulnerability if it does not contain a test on *this.balance* or if g cannot be executed prior to f under inspection, which is expressed by the LTL property:

$$\begin{aligned} selfDestruction_{M^f_{s_i}} =& (\neg testOnBalance_{M^f_{s_i}}) \\ & \vee (\neg selfDestruct_{M^g_{s_j}}\mathcal{U}\ start_{M^f_{s_i}}) \end{aligned}$$

We note that even though these properties can detect the presence of the self destruction vulnerability, more information on what the function exactly does needs to be provided in order to be able to assess its harmfulness on the execution. This can still be checked by evaluating a contract-specific property.

6.3.4 Timestamp Dependence

In order to check for this vulnerability, we propose an LTL property to detect the accessibility of *block.timestamp* or its alias *now*:

$$TSD_{M^f_{s_i}} = \Box\neg TimestampDependantStatement$$

Where *TimestampDependantStatement* is true if a *timestamp* is used in a statement. Similarly to the self destruction vulnerability, the presence of timestamp dependence can be detected using the proposed property, but to check the harm it may incur a more appropriate contract-specific property needs to be evaluated.

6.3.5 Skip Empty String Literal

This can be checked for a function f containing function calls by verifying that for any function call with n arguments $\{a_1, .., a_n\}$ no empty string is passed as an argument (except for the last one a_n). We express this as follows:

$$SkipEmpty_{M^f_{s_i}} = \Box\neg FunctionCall$$

Where *FunctionCall* is true when the statement is a function call with an empty argument a_i $(i \neq n)$.

6.3.6 Uninitialized Storage Variable

This is verified for a function f where a variable x of a complex type is defined, by checking the following property:

$$UnintializedVariable_{M^f_{s_i}} = \neg readX\mathcal{U}writeX$$

Where *readX* is true when x is read in a statement and *writeX* is true when it is assigned.

6.4 Conclusion

In this chapter, we achieved our objective ***Obj3*** mentioned in Section 1.3 which is the formal definition of well-known vulnerabilities. To do so, we have selected and defined six common Solidity vulnerabilities, namely integer overflow/underflow, reentrancy, self-destruction, timestamp dependence, skip empty literal and uninitialized

storage variable. We have then proposed a formalization for these vulnerabilities using Linear Temporal Logic. We recall that our main aim is to allow the verification of LTL properties that reflect characteristics that depend on the smart contracts to be verified. Through the formalization of these vulnerabilities in LTL, we prove that our approach is *also* capable of detecting the presence vulnerabilities in Solidity smart contracts.

Chapter 7

Application and Implementation of our Approach

Contents

7.1 Introduction

This chapter represents the fourth contribution of our work, which consists in the last step towards our end-goal approach (Figure 7.1). With this contribution we aim to achieve ***Obj4*** which is part of the response to ***RQ3*** and ***RQ4***. Herein we are interested in the generation of the final Hierarchical CPN model that will be passed on to the model checker. We will furthermore explain how all of our contributions are linked together to give rise to our full approach, and present our tool *Solidity2CPN* which automates our approach while offering a user-friendly interface.

7.2 Generation of the Final Hierarhical Coloured Petri Net Model

We recall that the end-goal of this PhD work is to implement a full approach for the formal verification of Solidity smart contracts used in a BPM context. Our approach comprises mainly two steps:

1. A pre-verification phase: consists in *(i)* transforming the smart contracts' Solidity code into CPN sub-models corresponding to their functions and *(i)* transforming the behavioural context specification into

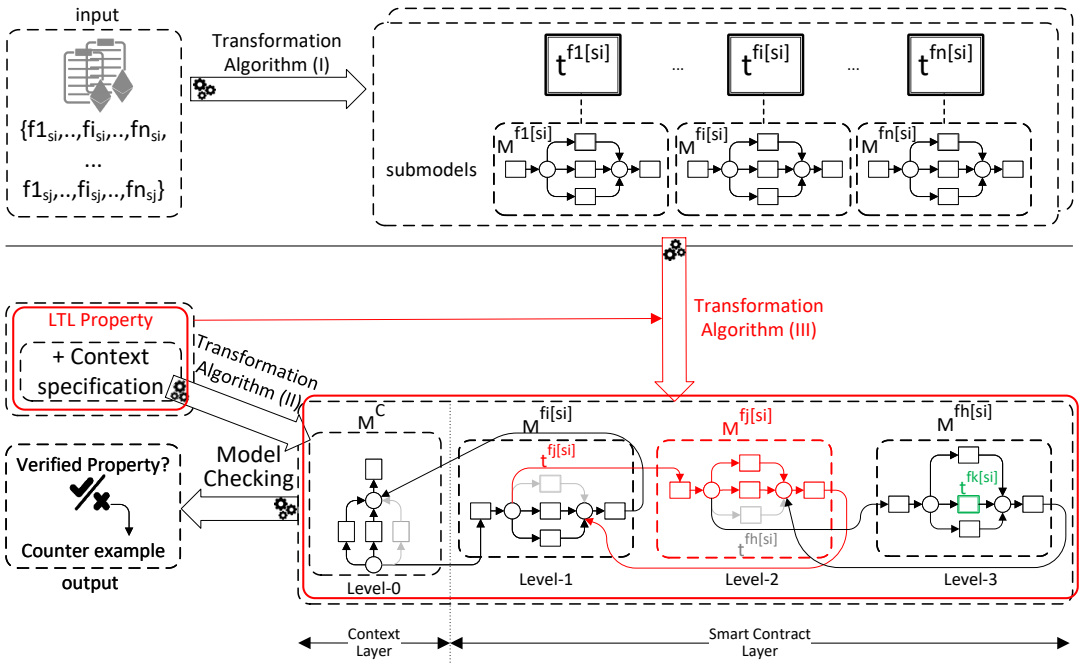

Figure 7.1: Approach overview - Step 4

2. A verification phase: consists in constructing a CPN model with regard to an LTL property that can express: *(i)* a vulnerability in the code or *(ii)* a contract-specific property, linking it to a CPN model representing the provided behavioral context to be considered, and feeding it the model checking to verify the targeted property.

More precisely, we opt for a hierarchical CPN model to represent the considered smart contracts' execution and interaction with respect to the provided behavioral context specification. As shown in Figure 1.1, we represent each function of a smart contract by an *aggregated transition* that encapsulates a CPN sub-model corresponding to the internal workflow of the former. These sub-models are initially represented disjointedly. In fact, our aim at this pre-verification phase is to get building blocks for the hierarchical model that will be fed to the model checker. Then, given a behavioral context specification and an LTL property to be verified, the final CPN model is built by *(1)* linking the aggregated transition representing the targeted function to the behavioral context's model and *(2)* building a hierarchy by explicitly representing function calls in the sub-model in question (if the checked property requires it). In fact, function calls are initially abstracted and therefore represented by aggregated transitions in the model (e.g., $t^{fj[si]}$ in Figure 1.1) under the assumption that they do not present behavioral problems (deadlock-free and strong-livelock-free) which can be separately verified for each function. Depending on the property to be verified, an

aggregated transition may need to be *unfolded* if any of its corresponding function's instructions or variables are involved in this property, hence the multi-level hierarchy in the model (e.g., $t^{fj[si]}$ in $M^{fi[si]}$ is hidden and replaced by its sub-model $M^{fj[si]}$). It is kept *folded* otherwise (e.g., $t^{fk[si]}$ in $M^{fh[si]}$). This abstraction leads to a reduction in the size of the final CPN model, and consequently (in general) a reduction in the size of the state space the model checker needs to explore.

This unfolding mechanism is implemented by the following algorithm.

The unfoldTransition Algorithm

procedure UNFOLDTRANSITION(t^a;p_{in};p_{out})
  Input: aggregated transition t^a, input place p_{in}, output place p_{out}
  Output: submodel replacement of transition t^a
  for $t' \in t^a.submodel.inTransition$ **do**
    replicate (arc from p_{in} to t^a) to t'
    replicate (arc from $\bullet t[input]$ to t^a) to t'
    for $p \in \bullet t^a[data] \cup \bullet t^a[output]$ **do**
      replicate (arc from p to t^a) to t'
    end for
  end for
  for $t' \in t^a.submodel.outTransition$ **do**
    replicate (arc from t^a to p_{out}) to t' with the placeholder inscription replaced by values from $\bullet t'[cf]$
  end for
  hide transition t^a and all arcs linked to it
end procedure

7.3 Application of the Approach

The full application of our end-to-end approach consists in linking the different contributions presented in the previous chapters of this manuscript. Given a set of Solidity smart contracts to be verified, a behavioural context specification that describes the way they are intended to be used, a property that we want to check for, we start by applying our transformation algorithms presented in Chapter 4 on the functions of the smart contracts to generate the different building blocks of the final HCPN model. Then, depending on the type of the provided context specification, we apply the definitions given in Chapter 5 to generate what we call the level-0 CPN model. A hierarchical CPN model is then built upon this initial level-0 CPN model following the unfolding mechanism previously explained in Section 7.2 and using the previously generated building blocks. The hierarchy of this HCPN is derived from the property to be verified, which can be a contract-specific property or a predefined property dedicated to the detection of a vulnerability in the smart contracts as per the for-

malizations given in Chapter 6. Once all these steps are carried out, the verification of the targeted property on the contracts would come down to its verification on the final HCPN model by passing the latter to a model checker. For model checking, we chose *Helena* [45] which is a High LEvel Nets Analyzer available as a command line tool. It offers explicit model checking support for an on-the-fly verification of state and LTL properties over CPN models described in *Helena*'s specification language.

We have automated our approach through a web application that we present in the following section.

7.4 The Solidity2CPN Tool for Smart Contracts Verification

We have implemented our approach through a web application[1]. The front-end was developed in Javascript with the VueJS framework. As for the back-end, it was developed using the python Django platform. The different modules have been implemented in C++.

7.4.1 Tool Architecture

As shown in figure 7.2, the tool has basically three inputs:

- *Solidity Smart contracts:* this consists in the *.sol* file(s) of the Smart contract(s) to be verified.
- *Context DCR/CPN:* it is a representation of the behavioral context of the provided smart contract(s). It can be either a declarative representation (i.e., a DCR graph/choreography), or an imperative representation (i.e., a CPN model). This input is optional.
- *Property to verify:* this is the property the user would like to verify on their smart contract(s). This property can either be *(i)* a Solidity vulnerability, or *(ii)* a property specific to the smart contract(s).

The tool basically consists of seven modules:

- *Solidity2CPN:* This is the module responsible for the generation of the CPN sub-models corresponding to the functions of the smart contracts given as input.
- *DCR2CPN:* This is the module responsible for the generation of the CPN sub-model corresponding to the DCR representation of the context (optionally) given as input.

[1] https://depot.lipn.univ-paris13.fr/soliditycpn/soliditycpn

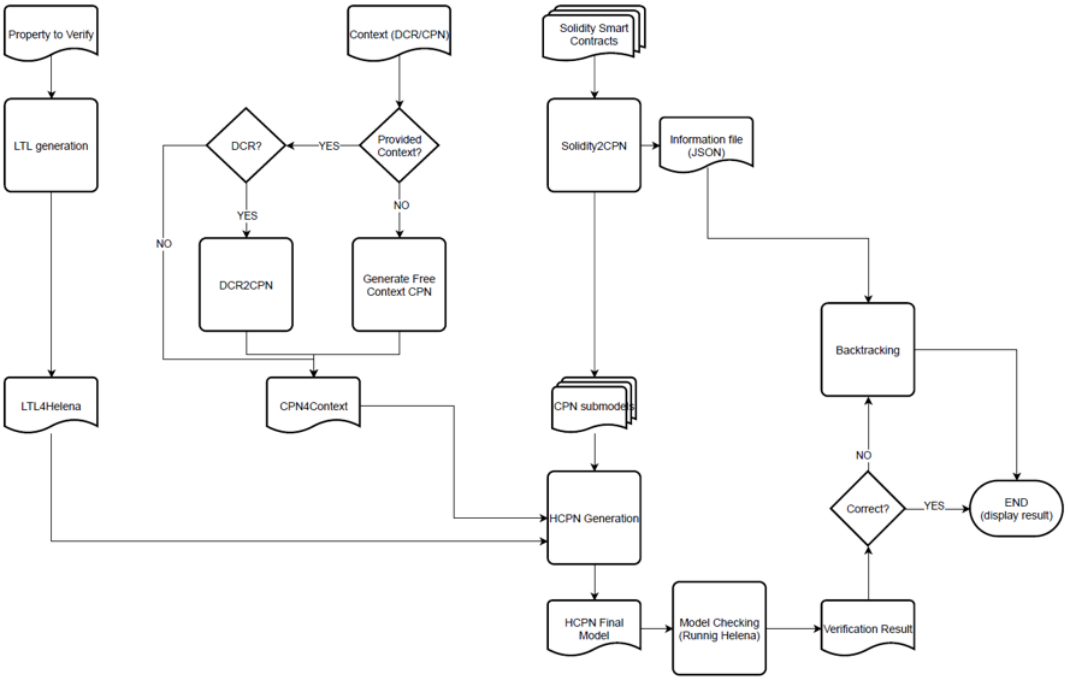

Figure 7.2: Architecture of the Solidity2CPN tool

- *Generate Free Context CPN:* This is the module responsible for the generation of the CPN sub-model corresponding to a free context. In other words, it is the module invoked when the user has no constraints on its contracts (i.e., does not provide a representation for the context).

- *LTL Generation:* This is the module responsible for the transformation of the property to be checked, given by the user, and expressed using the elements of the input artifacts (i.e., the contracts and possibly the context), into an *LTL* property expressed using the elements of the CPN sub-models generated by the other modules, and written according to the *Helena* model checker specification language.

- *HCPN Generation:* This is the module responsible for the generation of the final hierarchical CPN model. Initially based on the CPN model of the context, this module unfolds (i.e., replaces an aggregate transition by a CPN sub-model) the transitions concerned by taking into account the LTL property to be checked and using the corresponding CPN sub-models of the smart contracts functions.

- *Model Checking:* This is the module responsible for invoking the *Helena* model checker with the final HCPN model and the LTL formula to be checked as inputs.

- *Backtracking:* This is the module responsible for the interpretation of the result returned by *Helena*. Indeed, in the case where the model does not satisfy the verified property, *Helena* returns a counter-example in the form of a sequence of transitions and firings. This sequence should then be transformed into an execution trace expressed on the elements of the smart contracts and the context (if it exists) previously provided as input so that the user can understand, and interpret the verification result and possibly correct his smart contracts accordingly. We note that at the time of writing, this module has not been yet implemented.

7.4.2 Tool Workflow

The tool's workflow can be described by a typical scenario as follows:

- The user begins a verification session by selecting, from his list of smart contracts, the contract(s) concerned by the verification (Fig. 7.3).

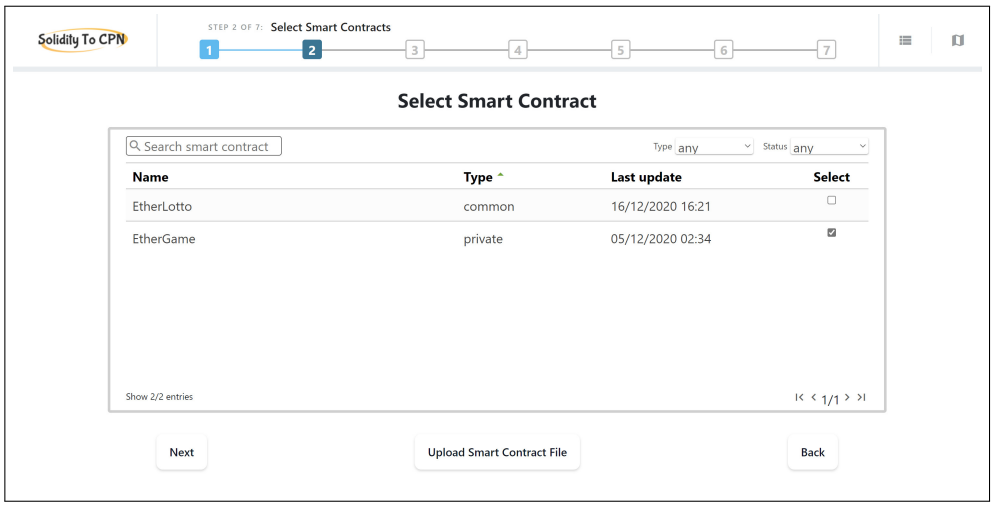

Figure 7.3: Smart contracts selection interface

- If the user does not yet have contracts in his list, or if the contract(s) concerned by this verification session do(es) not appear on it, he has the possibility of adding it/them by uploading the corresponding *.sol* file(s).

- The user is then prompted to select a context for his selected contract(s). As was the case for the selection of contracts, the user can either select from his list of contexts or upload a file (*.xml* in the case of a DCR context, or *.lna* in the case of a CPN context). Since the context is an optional artifact, the user may as well skip this step (Fig. 7.4).

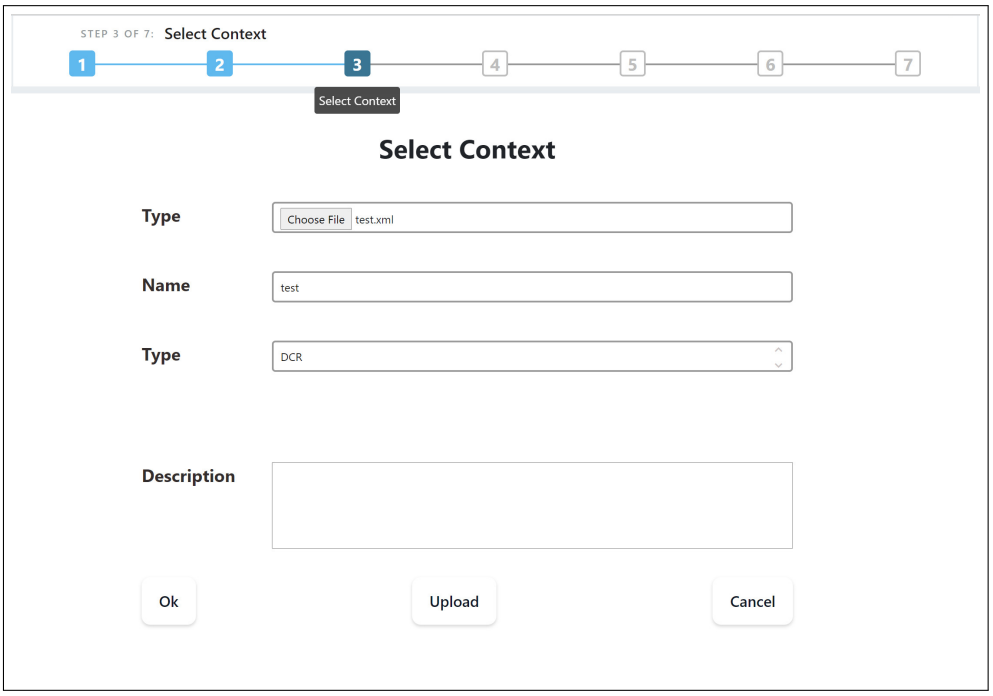

Figure 7.4: Context selection interface

- The user is then invited to select the type of property he would like to verify on his contracts. This property can be:
  - a property corresponding to a vulnerability. In this case, the tool offers different interfaces to guide the user to provide the necessary information for the selected vulnerability among the following six supported vulnerabilities (Fig. 7.5):
    1. integer underflow/overflow
    2. reentrency
    3. self-destruction
    4. timestamp-dependance
    5. skip empty string literal
    6. unintialiazed storage variable
  - a property specific to the contract(s) to be check (Fig. 7.6). In this case, the tool offers the user two possibilities:
    * expressing the property using predefined LTL templates for standard and common properties where the user has to only fill predefined fields with contract information (e.g., functions, variables, etc). We note that at the time of writing, this part has not been yet implemented.

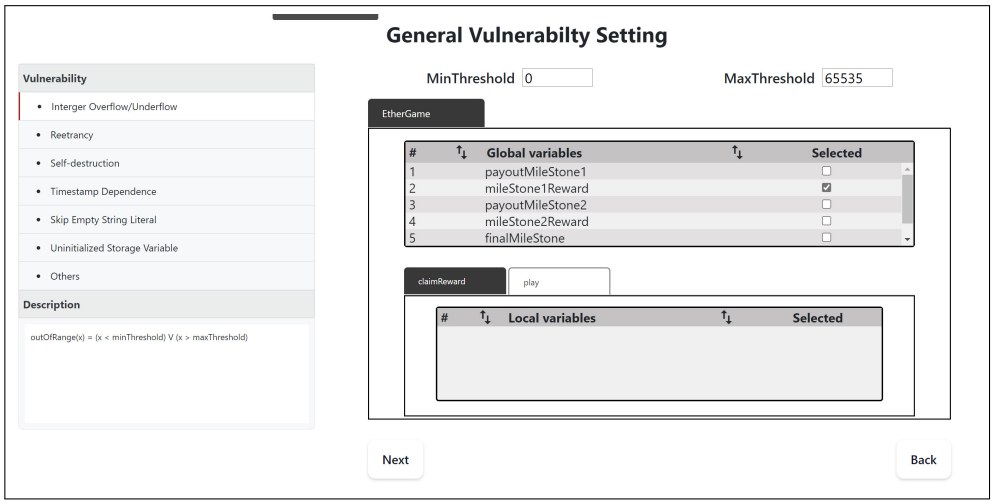

Figure 7.5: Vulnerability selection interface

∗ expressing the property freely (without template) in LTL. In this case, the user needs to be familiar with the syntax and semantics of LTL.

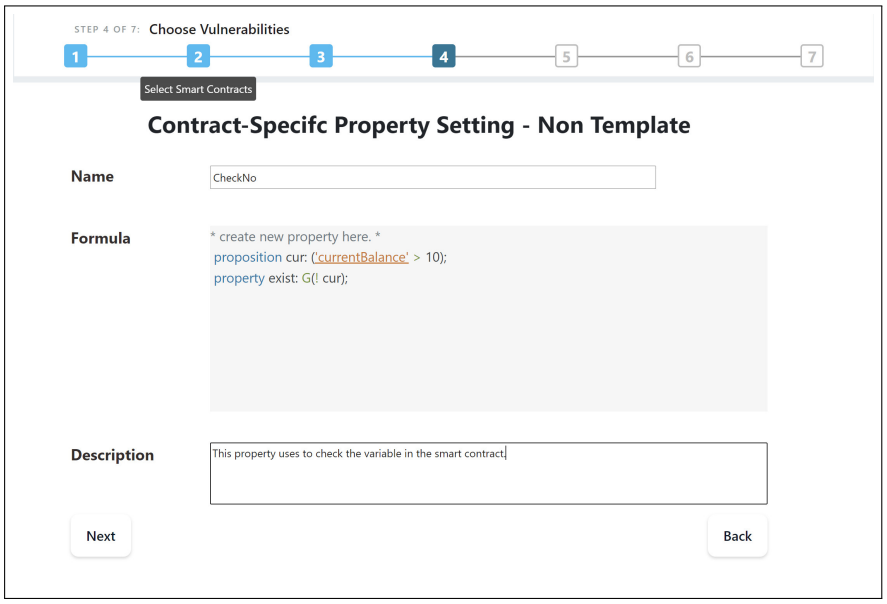

Figure 7.6: Contract-specific property setting interface (without a template)

- The user gives some additional information (e.g., the number of users of his contracts, their balances, etc.) to configure the final verification session (Fig. 7.7).

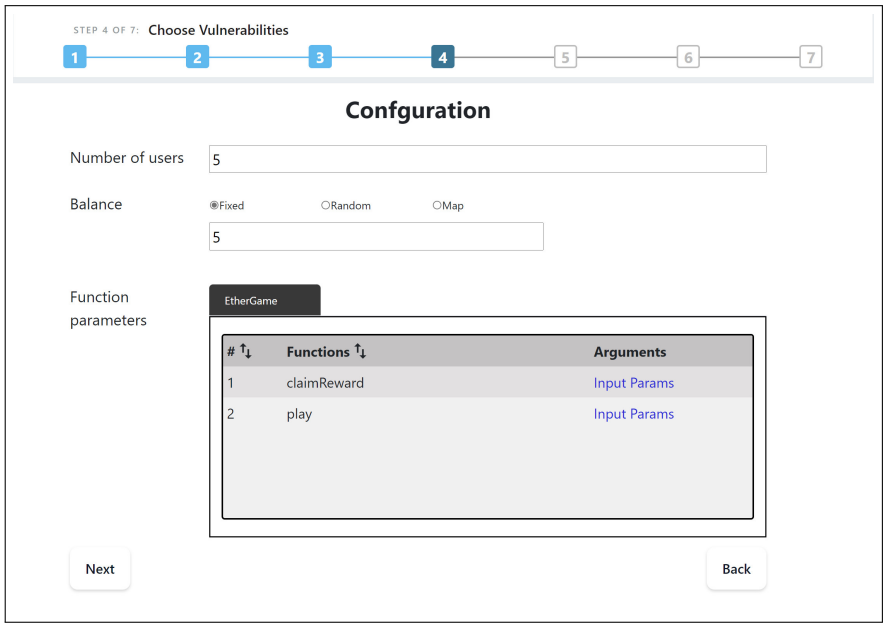

Figure 7.7: Configuration interface

- The user launches the generation of the target model (the final hierarchical CPN as well as the LTL property expressed in the specification language of *Helena*) (Fig. 7.8).

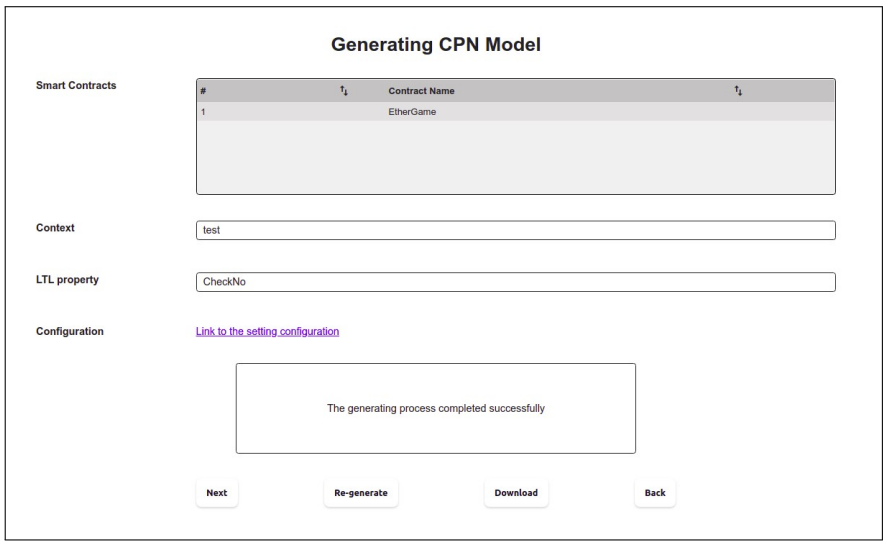

Figure 7.8: Generation interface

- The user launches the verification of his contract(s) (the tool invokes *Helena* with the artifacts generated in the previous step) (Fig. 7.9).

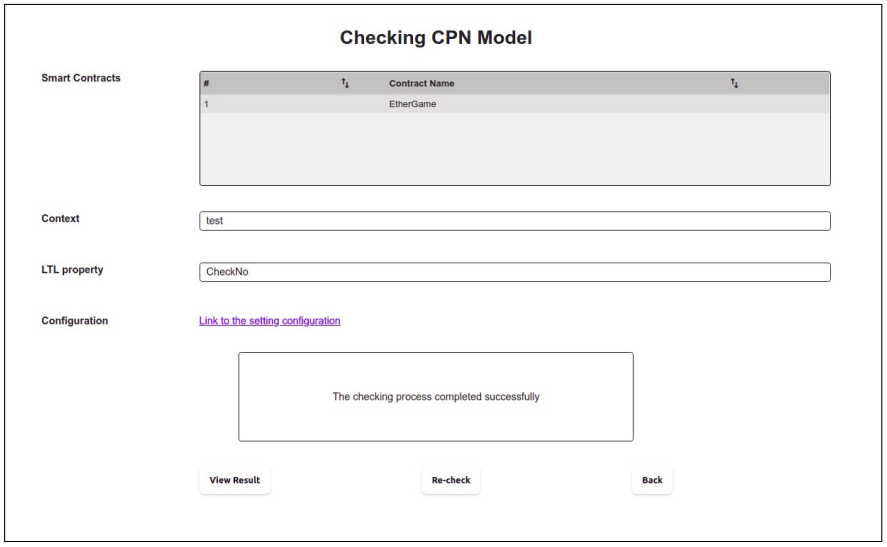

Figure 7.9: Verification interface

- The tool displays the verification result (Fig. 7.10).

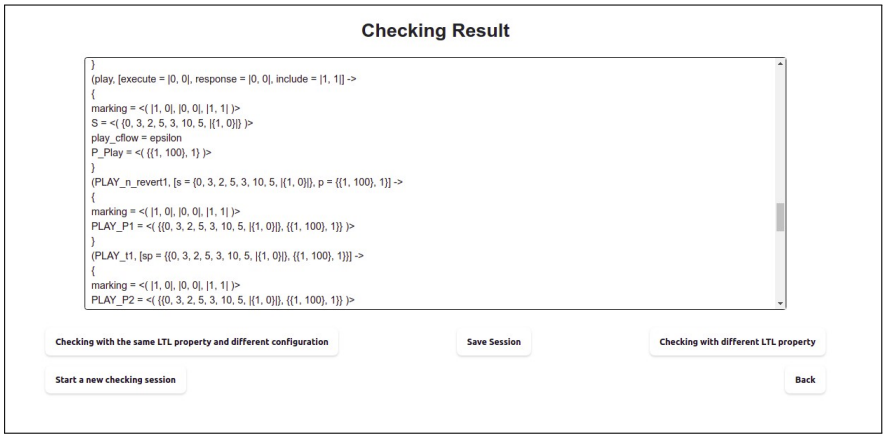

Figure 7.10: Results interface

- The user can thus initiate another verification session.

7.5 Conclusion

In this chapter, we achieved our objective ***Obj4*** mentioned in Section 1.3 which is providing a full approach to formally verify smart contracts while supporting the possibility of behavioural context specifications. To do so, we have presented how the generation of the final Hierarchichal CPN model is implemented through the *unfolding* mechanism for which we have proposed a generic algorithm. The final step that concludes our approach is then too apply model checking on the generated HCPN model w.r.t to the specified LTL property. We have also presented our graphical *Solidity2CPN* tool which automates our end-to-end approach and provides user-friendly interfaces to bring our approach closer to as a wide a range of users as possible.

Chapter 8

Conclusion an Future Work

Contents

Being an important pillar for Blockchain technology, smart contracts need to provide certain guarantees in terms of correctness to support its foundation built on trust. Formal approaches for the verification of Solidity smart contracts have been proposed, but they are generally designed to target specific vulnerabilities known in the literature (e.g., reentrancy) which have been reported to be the root of some attacks or malfunctions. Checking the absence of such vulnerabilities in a smart contract does not guarantee its correctness as a faulty behaviour may stem from a flaw specific to that contract. Moreover, the need to verify contract-specific properties has proven increasingly necessary in the light of the expanding reach of smart contracts in many application fields. In fact, the combination of the Blockchain technology and the BPM domain has been an evident step, especially considering the assets that the former brings to the latter. It is still crucial, however, to guarantee the correctness of the smart contracts involved in this association in order to ensure its safety. To the best of our knowledge, there are no studies that focused on the verification of smart contracts in the context of BPM despite the fact that Business processes can be intuitive representations to describe the context in which smart contracts are used.

Our proposed approach comes as an effort to bring a solution to this research issue of *formal verification of Solidity smart contracts in a BPM context* by providing a way to formally model contracts along with their behavioural context specification (while also considering the case where no such specification is provided). We do that by both checking for vulnerabilities in the code and offering the possibility to express additional contract-specific properties to check. Moreover, we take into account the context in which the smart contracts to be verified are executed as a behavior specification.

In this manuscript, we presented in details our contributions to tackle the problems specified in Section 1.2. In this concluding chapter, we summarize our work in section 8.1 and present our future research directions Section 8.2.

8.1 Fulfillment of Objectives

The ultimate goal of this PhD work was to provide a solution that would allow Solidity developers of the BPM community to make sure that their smart-contracts-based applications are correct. This correctness characterization can be in the form of vulnerabilities detection. In fact, the semantics of Solidity can be at times obscure even to domain experts, and many documented attacks on Blockchain platforms prove that seemingly-correct smart contracts can be susceptible to malicious manipulations. The correctness of smart contracts can also be defined as some contract-specific property that is specific to the use case at hand. This is especially important when the designers have some knowledge on the context in which their smart contracts will be used and are able to formulate properties that they would expect their application to satisfy/hold. Our solution, despite being oriented to business process designers, can also be used for the mere objective of verifying smart contracts in general (i.e., without having to provide a behavioural context).

To achieve such a goal, we proposed a model-checking-based approach using Coloured Petri Net as a formalism for the representation of smart contracts and their behavioural contexts and Linear Temporal Logic to express properties on the modeled Blockchain-based application. To do so, we had to divide our problem into a set of challenges and therefore touch on a number of milestones.

The *first contribution* aimed to provide a formal representation for Solidity smart contracts in the form of Coloured Petri Nets. This choice of formalism emanated from its capability to represent both control- and data-flows of systems which is an important aspect to consider when aiming to verify contract-specific properties. To achieve this aim, we proposed CPN patterns for each type of statement of the Solidity language along with well-defined algorithms that would serve for the automation of the transformation Solidity code into CPN.

The *second contribution* had as a goal the formalization of the behavioural specifications that may describe the context in which the smart contracts to be verified are to be used. Our approach being oriented towards smart contracts used in the BPM domain, we focused on declarative representations that Business process designers may use to express constraints on their Blockchain-based applications, especially considering the scarcity of studies on formal verification of such models. We singled out DCR graphs and choreographies as our main representation of choice for a *constrained context* as they are one the modeling techniques that are gaining popularity in the field of BPM, and proposed semantically-equivalent formal CPN models to integrate them in our ultimate solution. To keep our approach as generic as possible, we also proposed a CPN model, dubbed as a *free context* in our work, that would be used as a context in case none has been provided. This model is built on the elements of the smart contracts to be verified and makes no assumptions on their behaviour, therefore it can be used in our approach as a basis for the general verification of smart contracts.

The *third contribution* had in view the formalization of vulnerabilities. Having established that the detection of flaws in Solidity smart contracts is not to be neglected as part of checking for their correctness, we selected six of the commonly-present vulnerabilities in Solidity to target in our work. Since our approach is based on the verification of Linear Temporal Logic properties, we represented the selected vulnerabilities in the form of different LTL formulae, the verification of which would enable designers to detect the corresponding vulnerabilities in their code. We note that in our work, the six vulnerabilities that we considered are given as mere examples to prove that the expressiveness of LTL formulae can cover vulnerabilities from the literature even though our focus is on the verification of contract-specific properties.

The *last contribution* sought to put all the pieces of our approach together to achieve our final goal. This consisted in the generation of the final hierarchical CPN model by *1* generating the different CPN sub-models corresponding to the smart contracts functions as per the first contribution, *2* generating the level-0 CPN model depending on the provided behavioural context specification (or lack thereof) as per the second contribution, *3* the definition of the properties to be verified on the Blockchain-based application as per the third contribution, and *4* using the level-0 CPN model as the initial start and building a hierarchy upon it based on the property to verify and using the previously-generated sub-models.
To prove the feasibility of our approach we have developed a fully-fledged tool that provides users with a graphical interface that allows them to specify the property to be verified on the smart contracts, be it contract-specific or corresponding to a common vulnerability, as well as to potentially indicate the behaviour according to which the contracts are used as a BP specification. The transformation of the smart contracts and their behaviours are carried out in the background followed by the model checking phase which is performed by transparently invoking *Helena* and returning the obtained results. The users, oblivious to the running mechanisms behind the scenes, therefore benefit from a seamless experience that allows them to formally verify their smart contracts without any experience requirements on formal modelling nor model checking.

8.2 Future Work

Our work opens several research perspectives to accomplish in short and middle terms. The most immediate perspective is rather technical and concerns our Solidity2CPN tool. Then we intend to investigate the possibility of extending our work by focusing on the integration of the Blockchain technology with other domains, mainly that of the Internet of Things (IoT). Lastly, we will work on improving the performance of our approach by exploring the *Helena* model checker.

Finishing up the Solidity2CPN tool. As was mentioned in Chapter 7, there are still two parts that we are currently working on implementing for our tool, namely

adding the support of template-based LTL properties along with the necessary graphical interfaces for that, and the *Backtracking* module. For the first part, we need to identify standard and/or common properties that would be interesting to check for our target users and then define them in the form of LTL formulae that could be "configured" or "set up" with elements of the smart contracts in question. We will then design corresponding graphical interfaces to include such properties as templates in our tool. For the second part, we need to implement a module that would transform the counter example returned by *Helena* (in case of violated property), which is a sequence of states and fired transitions, into a result that the users can easily interpret as an execution trace of their smart contracts.

Supporting IoT as a behavioural context. Just like BPM, the integration of the Blockchain technology with the IoT has been a hot topic recently [37]. We intend to extend our current approach for the verification of Blockchain-based IoT applications by additionally supporting behavioural context specifications provided as an IoT design, in particular as a Node-Red [7] application. To do that, we will transform Node-Red applications that use smart contracts to handle and manipulate data from IoT devices into CPN models, and apply the same strategy presented in our work to verify the correctness of such applications.

Improving *Helena* To further improve the performance of our approach, we plan to work on *Helena*'s model checker by embedding it with an extension to an existing technique based on symbolic observation graphs (SOG) [60] previously developed to deal with the state space explosion problem in regular PNs and adapting it to CPNs.

Bibliography

[1] Blockchain technology history: Ultimate guide. https://101blockchains.com/history-of-blockchain-timeline/

[2] Ethereum — ethereum.org. https://ethereum.org/en/

[3] Ethereum bounty program. https://bounty.ethereum.org/

[4] Formal verification for solidity contracts — ethereum community forum. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts

[5] Language grammar — solidity 0.8.18 documentation. https://docs.soliditylang.org/en/latest/grammar.html

[6] Magic BlockchainQA: Blockchain qa services. https://www.magicblockchainqa.com/

[7] Node-red. https://nodered.org/

[8] Numbers about blockchain - fujitsu blockchain innovation center : Fujitsu belgium. https://www.fujitsu.com/be/microsite/blockchain/numbers-about-blockchain/

[9] Overflow incident, https://en.bitcoin.it/wiki/Value_overflow_incident

[10] Parity bug bounty program. https://www.parity.io/bug-bounty/

[11] Ropsten. https://ropsten.etherscan.io/

[12] Solidity documentation. https://solidity.readthedocs.io/en/latest/

[13] Top 10 blockchain statistics and facts that will make you think. https://yourtechdiet.com/blogs/blockchain-stats/

[14] Top 5 programming languages to build smart contracts - 101 blockchains. https://101blockchains.com/smart-contract-programming-languages/

[15] Ethlint: (formerly solium) linter to identify and fix style & security issues in solidity. https://github.com/duaraghav8/Ethlint (2019)

[16] Solhint: Solhint project. https://github.com/protofire/solhint (2019)

[17] van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: Yet another workflow language. Inf. Syst. 30(4), 245–275 (Jun 2005)

[18] van der Aalst, W.M.P., Pesic, M.: Decserflow: Towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4184, pp. 1–23. Springer (2006)

[19] van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Comput. Sci. Res. Dev. 23(2), 99–113 (2009)

[20] Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.: Validation, verification, and testing of computer software. ACM Comput. Surv. 14(2), 159–192 (1982)

[21] Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 66–77. New York, NY, USA (2018)

[22] Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. Int. J. Softw. Tools Technol. Transf. 11(1)

[23] Angelo, M.D., Salzer, G.: A survey of tools for analyzing ethereum smart contracts. 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON) pp. 69–78 (2019)

[24] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings

[25] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)

[26] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016. pp. 91–96 (2016)

[27] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings. pp. 193–207 (1999)

[28] Bond, F.: Solcheck: A solidity linter written in js. https://github.com/federicobond/solcheck (2017)

[29] Boubaker, S., Klai, K., Kortas, H., Gaaloul, W.: A formal model for business process configuration verification supporting or-join semantics. In: Panetto, H., Debruyne, C., Proper, H.A., Ardagna, C.A., Roman, D., Meersman, R. (eds.) On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11229, pp. 623–642. Springer (2018)

[30] BPM.com: What is bpm? https://bpm.com/what-is-bpm

[31] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. CoRR abs/1809.03981 (2018)

[32] Bryans, J.W., Wei, W.: Formal analysis of BPMN models using event-b. In: Kowalewski, S., Roveri, M. (eds.) Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6371, pp. 33–49. Springer (2010)

[33] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of VSL 2014, Austria.

[34] Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017. pp. 442–446 (2017)

[35] Coalition, T.W.M.: What is bpm? http://wfmc.org/what-is-bpm

[36] Dadam, P., Reichert, M., Rinderle, S., Jurisch, M., Acker, H., Göser, K., Kreher, U., Lauer, M.: Towards truly flexible and adaptive process-aware information systems. In: Kaschek, R.H., Kop, C., Steinberger, C., Fliedl, G. (eds.) Information Systems and e-Business Technologies, 2nd International United Information Systems Conference, UNISCON 2008, Klagenfurt, Austria, April 22-25, 2008, Proceedings. Lecture Notes in Business Information Processing, vol. 5, pp. 72–83. Springer (2008)

[37] Dai, H., Zheng, Z., Zhang, Y.: Blockchain for internet of things: A survey. IEEE Internet Things J. 6(5), 8076–8094 (2019)

[38] Dechsupa, C., Vatanawood, W., Thongtak, A.: Transformation of the BPMN design model into a colored petri net using the partitioning approach. IEEE Access 6, 38421–38436 (2018)

[39] Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. 50(12), 1281–1294 (2008)

[40] Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P.E., Deng, L.: Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. IJNDC 7(3), 121–132 (2019)

[41] Dodson, N.: Solint: A linting utility for ethereum solidity smart-contracts. https://github.com/SilentCicero/solint (2016)

[42] Duo, W., Huang, X., Ma, X.: Formal analysis of smart contract based on colored petri nets. IEEE Intell. Syst. 35(3), 19–30 (2020)

[43] Dwivedi, V., Deval, V., Dixit, A., Norta, A.: Formal-verification of smart-contract languages: A survey. In: Singh, M., Gupta, P., Tyagi, V., Flusser, J., Ören, T., Kashyap, R. (eds.) Advances in Computing and Data Sciences. pp. 738–747. Springer Singapore, Singapore (2019)

[44] Efanov, D., Roschin, P.: The all-pervasiveness of the blockchain technology. In: Samsonovich, A.V., Klimov, V.V. (eds.) 8th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2017, August 1-6, 2017, Moscow, Russia. Procedia Computer Science, vol. 123, pp. 116–121. Elsevier (2017)

[45] Evangelista, S.: High level petri nets analysis with helena. In: Applications and Theory of Petri Nets 2005. pp. 455–464. Berlin, Heidelberg (2005)

[46] Fahland, D., Lübke, D., Mendling, J., Reijers, H.A., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: Halpin, T.A., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings. Lecture Notes in Business Information Processing, vol. 29, pp. 353–366. Springer (2009)

[47] Gaaloul, W., Bhiri, S., Rouached, M.: Event-based design and runtime verification of composite service transactional behavior. IEEE Trans. Serv. Comput. 3(1), 32–45 (2010)

[48] Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Nickovic, D. (eds.) Formal Methods for Industrial

Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12327, pp. 3–69. Springer (2020)

[49] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 343–361 (2015)

[50] Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. EPTCS, vol. 69, pp. 59–73 (2010)

[51] Hildebrandt, T.T., Slaats, T., López, H.A., Debois, S., Carbone, M.: Declarative choreographies and liveness. In: Pérez, J.A., Yoshida, N. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11535, pp. 129–147. Springer (2019)

[52] Hirai, Y.: Ethereum vm for coq (v0.0.2). https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18 (March 2017)

[53] Holloway, C.: Why engineers should consider formal methods. In: 16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings. vol. 1, pp. 1.3–16 (1997)

[54] Houhou, S., Baarir, S., Poizat, P., Quéinnec, P., Kahloul, L.: A first-order logic verification framework for communication-parametric and time-aware BPMN collaborations. Inf. Syst. 104, 101765 (2022), https://doi.org/10.1016/j.is.2021.101765

[55] Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)

[56] Jiao, J., Kan, S., Lin, S., Sanán, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: Executable operational semantics of solidity. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1695–1712. IEEE (2020)

[57] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018 (2018)

[58] Kheldoun, A., Barkaoui, K., Ioualalen, M.: Formal verification of complex business processes based on high-level petri nets. Inf. Sci. 385, 39–54 (2017), https://doi.org/10.1016/j.ins.2016.12.044

[59] Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings. pp. 553–568 (2003)

[60] Klai, K., Poitrenaud, D.: MC-SOG: an LTL model checker based on symbolic observation graphs. In: Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings. pp. 288–306 (2008)

[61] OMG. Unified modelling language, v.: http://www.omg.org/spec/uml/2.3/, http://www.omg.org/spec/UML/2.3/

[62] Liu, Z., Liu, J.: Formal verification of blockchain smart contract based on colored petri net models. In: 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 2. pp. 555–560. IEEE (2019)

[63] López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: A business process execution engine on the ethereum blockchain. Softw. Pract. Exp. 49(7), 1162–1193 (2019)

[64] Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 254–269 (2016)

[65] Marmsoler, D., Brucker, A.D.: A denotational semantics of solidity in isabelle/hol. In: Calinescu, R., Pasareanu, C.S. (eds.) Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13085, pp. 403–422. Springer (2021), https://doi.org/10.1007/978-3-030-92124-8_23

[66] Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Financial Cryptography and Data Security

- 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018

[67] Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019

[68] McMillan, K.L.: Symbolic model checking. Kluwer (1993)

[69] Meghzili, S., Chaoui, A., Strecker, M., Kerkouche, E.: An approach for the transformation and verification of BPMN models to colored petri nets models. Int. J. Softw. Innov. 8(1), 17–49 (2020)

[70] Mendling, J., Weber, I.: Blockchains for business process management - challenges and opportunities. EMISA Forum 38(1), 22–23 (2018)

[71] Mendling, J., Weber, I., van der Aalst, W.M.P., vom Brocke, J., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S., Gal, A., García-Bañuelos, L., Governatori, G., Hull, R., Rosa, M.L., Leopold, H., Leymann, F., Recker, J., Reichert, M., Reijers, H.A., Rinderle-Ma, S., Solti, A., Rosemann, M., Schulte, S., Singh, M.P., Slaats, T., Staples, M., Weber, B., Weidlich, M., Weske, M., Xu, X., Zhu, L.: Blockchains for business process management - challenges and opportunities. ACM Trans. Manag. Inf. Syst. 9(1), 1–16 (2018)

[72] Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings. pp. 67–82 (2009)

[73] Morimoto, S.: A survey of formal verification for business process modeling. In: Bubak, M., van Albada, G.D., Dongarra, J.J., Sloot, P.M.A. (eds.) Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part II. Lecture Notes in Computer Science, vol. 5102, pp. 514–522. Springer (2008)

[74] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008.

[75] Muehlen, M.Z., Recker, J.: How much language is enough? theoretical and practical use of the business process modeling notation. In: Proceedings of the 20th International Conference on Advanced Information Systems Engineering. pp. 465–479. CAiSE '08, Springer-Verlag, Berlin, Heidelberg (2008)

[76] Mukkamala, R.R.: A Formal Model For Declarative Workflows Dynamic Condition Response Graphs. Ph.D. thesis (06 2012)

[77] Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)

[78] Nakamoto, S., Bitcoin, A.: A peer-to-peer electronic cash system. Bitcoin.– URL: https://bitcoin. org/bitcoin. pdf (2008)

[79] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 653–663 (2018)

[80] OMG: Business process model and notation (bpmn) 2.0. http://www.omg.org/spec/BPMN/2.0/ (2011)

[81] Ou-Yang, C., Lin, Y.D.: Bpmn-based business process model feasibility analysis: a petri net approach. International Journal of Production Research 46, 3763 – 3781 (2008)

[82] Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws, Vienna, Austria, September 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4103, pp. 169–180. Springer (2006)

[83] Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA. pp. 287–300. IEEE Computer Society (2007)

[84] Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011. vol. 99, pp. 383–394 (2011)

[85] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence. pp. 46–57. IEEE Computer Society (1977)

[86] Puhlmann, F., Weske, M.: Using the *pi*-calculus for formalizing workflow patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings. vol. 3649, pp. 153–168 (2005)

[87] Rakamaric, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014.

[88] Rea, A.: SolCover: Code coverage for solidity smart-contracts. https://github.com/sc-forks/solidity-coverage (2016-2020)

[89] Rozier, K.Y.: Linear temporal logic symbolic model checking. Comput. Sci. Rev. 5(2), 163–203 (2011)

[90] Scheer, A.: ARIS - vom Geschaftsprozess zum Anwendungssystem. Springer, 4., durchges. aufl. edn. (2002)

[91] Siegel, D., Yue, F., Keoun, B., Shen, M., Engler, A., Dale, B.: The dao attack: Understanding what happened (Dec 2020), https://www.coindesk.com/understanding-dao-hack-journalists

[92] Staples, M., Chen, S., Falamaki, S., Ponomarev, A., Rimba, P., Tran, A., Weber, I., Xu, S., Zhu, J.: Risks and opportunities for systems using blockchain and smart contracts. In: Data61 (CSIRO), Sydney (2017)

[93] Team, S.: Parity multi-sig wallets funds frozen (explained) (2021), https://www.springworks.in/blog/parity-multi-sig-wallets-funds-frozen-explained/

[94] Tinelli, C.: Smt-based model checking. In: NASA Formal Methods - 4th International Symposium, NFM 2012, USA, April 3-5, 2012. Proceedings (2012)

[95] Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 664–676 (2018)

[96] Tran, A.B., Lu, Q., Weber, I.: Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In: Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018, Sydney, Australia, September 9-14, 2018. vol. 2196, pp. 56–60 (2018)

[97] Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Canada, October 15-19 (2018)

[98] Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G.M. (eds.) Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings). Lecture Notes in Computer Science, vol. 1043, pp. 238–266. Springer (1995)

[99] Verbeek, E., van der Aalst, W.M.P.: Woflan 2.0: A petri-net-based workflow diagnosis tool. In: Nielsen, M., Simpson, D. (eds.) Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding. Lecture Notes in Computer Science, vol. 1825, pp. 475–484. Springer (2000), https://doi.org/10.1007/3-540-44988-4_28

[100] Von Rosing, M., Von Scheel, H., Scheer, A.W.: The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM, vol. 1. Morgan Kaufmann (2014)

[101] Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering workflow models from event-based data using little thumb. Integr. Comput.-Aided Eng. (2003)

[102] Ye, J., Sun, S., Song, W., Wen, L.: Formal semantics of bpmn process models using yawl. In: 2008 Second International Symposium on Intelligent Information Technology Application. vol. 2, pp. 70–74 (2008)

[103] Zhou, E., Hua, S., Pi, B., Sun, J., Nomura, Y., Yamashita, K., Kurihara, H.: Security assurance for smart contract. In: 9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018. pp. 1–5 (2018)